Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

# Deployment-Technology-agnostic Management of Running Applications

Tobias Mathony

| | |
|---|---|
| **Course of Study:** | Softwaretechnik M. Sc. |
| **Examiner:** | Prof. Dr. Dr. h. c. Frank Leymann |
| **Supervisor:** | Lukas Harzenetter, M. Sc.,<br>Michael Wurster, M. Sc. |
| **Commenced:** | March 30, 2020 |
| **Completed:** | September 30, 2020 |

## Abstract

In recent years, a plethora of technologies emerged to automate the deployment of applications, which is, if manually performed, a complex and error-prone process. Since deployment technologies heavily differ in their feature sets, mechanisms, modeling languages, and deployment models, committing to a technology may result in a lock-in. Further, enterprises often use multiple deployment technologies for their applications, each fitting the respective need. However, managing multiple applications deployed with different technologies is tedious. Due to the aforementioned reasons, it is desirable to provide a normalized representation of running applications, as well as to enable the management of applications regardless of the technology used to deploy them. To bridge this gap, EDMMi is introduced, a normalized model to represent running applications independently of their deployment technology. Further, this work proposes an approach to retrieve the instance information of running applications in an automated manner. A mapping between technology-specific instance data and EDMMi is then used to derive a normalized model of the retrieved application instance. To gain advantage from standardization, a further transformation from EDMMi to the TOSCA instance model is provided. Afterwards, the standardized TOSCA instance model is enriched with additional management functionalities that can be executed on the running application based on the existing *Management Feature Enrichment* approach. As a result, the concept presented in this work enables the enrichment of running applications with standards-based executable workflows for additional management functionality, regardless of the technology used to deploy them. With the transformation of technology-specific instance information to a standardized instance model, also the management of running applications in one single place is enabled, uncoupled from their deployment technologies. To prove the feasibility of the proposed concept, a prototypical implementation and an accompanying case study within the EDMM Transformation Framework and the OpenTOSCA ecosystem is provided.

## Kurzfassung

In den vergangenen Jahren entstanden eine Vielzahl von Technologien, um die Bereitstellung von Anwendungen zu automatisieren, da diese, wenn sie manuell durchgeführt wird, komplex und fehleranfällig ist. Da sich solche Technologien jedoch in ihren angebotenen Funktionen, Mechanismen, Modellierungssprachen und Modellen erheblich unterscheiden, kann die Wahl einer Technologie einen Lock-In verursachen. Außerdem setzen Unternehmen oft verschiedene Deployment Technologien ein für verschiedene Anwendungen, was das ganzheitliche Management dieser Anwendungen weiter erschwert. Um dem entgegen zu wirken, führt diese Arbeit mit EDMMi ein normalisiertes Modell ein um laufende Anwendungen, deren Komponenten und Konfigurationen abzubilden, unabhängig von der Deployment Technologie, die für die ursprüngliche Bereitstellung der Anwendung genutzt wurde. Zudem wird ein Ansatz präsentiert, wie Instanzinformationen von laufenden Anwendungen automatisiert bezogen und mittels EDMMi normalisiert dargestellt werden können. Mit einer Transformation von EDMMi zu dem TOSCA Instanzmodell wird des Weiteren demonstriert, wie das normalisierte Modell standardisiert werden kann. Auf dem standardisierten TOSCA Instanzmodell basierend wird daraufhin der *Management Feature Enrichment* Ansatz angewendet, um Anwendungen mit zusätzlichen Management-Funktionalitäten auszustatten. Daraus resultiert eine standardisierte Darstellung einer laufenden Anwendung in TOSCA, welche automatisiert mit Management-Workflows angereichert wird, unabhängig von der genutzten Deployment Technologie für die ursprüngliche Bereitstellung. Des Weiteren ermöglicht die standardisierte Darstellung von laufenden Anwendungen das Management dieser an einem einzigen Ort, auch wenn verschiedene Deployment Technologien für die Bereitstellung genutzt wurden. Um den vorgestellten Ansatz zu validieren, wird zudem eine prototypische Implementierung und ein Validierungsszenario anhand des EDMM Transformation Frameworks und des OpenTOSCA Ökosystems beschrieben.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**API** Application Programming Interface.

**AWS** Amazon Web Services.

**BPEL** Business Process Execution Language.

**BPMN** Business Process Model Notation.

**CLI** Command Line Interface.

**CSAR** Cloud Service ARchive.

**DB** Database.

**DBaaS** Database-as-a-Service.

**DBMS** Database Management System.

**DMMN** Declarative Application Management Modeling and Notation.

**EC** Elastic Compute.

**EDMM** Essential Deployment Metamodel.

**EDMMi** Essential Deployment Metamodel instance.

**ETG** Enterprise Topology Graph.

**HOT** Heat Orchestration Template.

**HTTP** Hypertext Transport Protocol.

**IP** Internet Protocol.

**IT** Information Technology.

**OASIS** Organization for the Advancement of Structured Information Standards.

**OS** Operating System.

**PaaS** Platform-as-a-Service.

**PuppetDB** Puppet database.

**RAM** Random Access Memory.

**RDS** Relational Database Service.

**REST** Representational State Transfer.

**SaaS** Software-as-a-Service.

**SSH** Secure Shell.

**TOSCA** Technology and Orchestration Specification for Cloud Applications.

**UI** User Interface.

**VM** Virtual Machine.

**WAR** Web Application Resource.

**XaaS** Everything-as-a-Service.

**YAML** YAML Ain't Markup Language.

# 1 Introduction

With the advent of Cloud Computing and the digital era, success of enterprises is often determined by the efficient operation of Information Technology (IT) applications. Such applications typically consist of multiple components, making their manual deployment complex, error-prone, and hardly reusable [Opp03]. Thus, many technologies, called *deployment technologies*, emerged to automate their deployment, e. g., Terraform [Has20], Chef[Che20], or AWS CloudFormation [Ama20a]. Such technologies may be bound to a specific provider like AWS CloudFormation to AWS; others depend on specific platforms, for example, Kubernetes [The20b] on Docker [Doc20a], and others, such as Chef [Che20], require a running infrastructure node with a Chef client installed. Despite their differences, deployment technologies typically employ declarative deployment models to automate the deployment of an application. Declarative deployment models describe the desired state of an application by specifying its structure, i. e., its components and dependencies [WBF+19]. For instance, a declarative deployment model may consist of a Java web application deployed on an Apache Tomcat [The20a] web server, which is hosted on a VM that is running on OpenStack [Ope20e]. To automatically deploy an application described in a declarative deployment model, deployment technologies provide an engine which is able to interpret such models. In this process, the engine analyzes the described application, its components, and their structure to derive and execute the steps required to instantiate the application as specified in an automated manner.

On the contrary, imperative deployment models explicitly define all necessary steps and their execution order to instantiate an application. Imperative deployment models are specified and executed as process or workflow [EBF+17] using standardized languages like Business Process Execution Language (BPEL) [Org07] or Business Process Model Notation (BPMN) [Obj11]. Although imperative deployment models allow for arbitrary customization, research and industry agree on declarative deployment models being the most appropriate approach for the automation of application deployment [BBF+18; HAW11]. Hence, currently popular deployment technologies employ a declarative model, e. g., Puppet [Pup20b] and OpenStack Heat [Ope20a]. However, typically each deployment technology builds upon a proprietary modeling language and deployment model [EBF+17; WBF+19]. Furthermore, the feature set and mechanisms between such technologies vary. For instance, AWS CloudFormation only allows the management of infrastructure resources, while Puppet also supports the deployment and configuration of arbitrary software. Due to the aforementioned reasons, committing to one deployment technology may result in a deployment technology lock-in. To reduce the risk of a lock-in, Wurster et al. [WBF+19] introduced the Essential Deployment Metamodel (EDMM), which adds an abstraction layer to deployment models to ease the migration between deployment technologies. EDMM is a normalized metamodel that unifies the essential parts of the deployment models of the 13 most popular deployment technologies [WBF+19]. Further, Wurster et al. [WBB+20]

presented accompanying tooling with the *EDMM Modeling and Transformation System* to create technology-agnostic deployment models based on EDMM, which can be transformed into each of the 13 investigated technologies in an automated manner. This eases the migration of deployment models, which is, if manually performed, a tedious, error-prone and complex process [WBB+20; WBF+19]. However, the concept and tooling presented by Wurster et al. [WBB+20; WBF+19] currently only supports the migration of deployment models of applications that are not yet deployed, i. e., it lacks a model and transformation system for running applications. This is required especially since enterprises typically operate a large set of applications that were already deployed using deployment technologies. Added to this, different deployment technologies may be used within a company depending on an application and its needs. For example, Puppet is suitable to deploy proprietary software, while it is reasonable to choose AWS CloudFormation when deploying applications that use AWS infrastructure. To manage such a plethora of applications deployed with different technologies, management activities must be performed separately for each application, which is inconvenient and time-consuming.

Another issue of deployment technologies is the limited support of automated management for applications, especially considering holistic management functionalities addressing multiple, possibly distributed components [HBL+19]. Tackling this, Harzenetter et al. [HBL+19] introduced the *Management Feature Enrichment* approach to automatically derive component-specific management operations based on reusable component types. To enable this, the declarative deployment model of an application is investigated for its component types. For each identified component type, a repository containing management operations related to such types is searched. If a matching type is found, the management functionality is added automatically to the respective component. Based on this, imperative workflows are generated that orchestrate the enriched management operations according to their functionality. For example, all management operations concerned with testing an application are assembled to one holistic *testing* workflow. Yet, the presented approach does not support the management feature enrichment of applications that are already running, but only for normalized deployment models.

To tackle the aforementioned issues, this work proposes a concept to enrich running applications deployed with arbitrary deployment technologies with management functionalities by utilizing the existing approach presented by Harzenetter et al. [HBL+19]. To enable this, EDMM is extended with the Essential Deployment Metamodel instance (EDMMi) to represent running applications in a normalized, technology-agnostic manner. The suitability of EDMMi is demonstrated by a mapping between the instance models of popular deployment technologies and EDMMi, e. g., for Puppet, AWS CloudFormation, and Kubernetes. Moreover, deployment-technology-specific components like an AWS EC2 server instance and its properties are mapped to normalized component types to define the semantics of components in a technology-independent fashion. For example, a normative type *Database* comprises all databases, like PostgreSQL or MySQL. Furthermore, this work presents an approach to automatically derive the normalized EDMMi from technology-specific instance information of running applications using the Application Programming Interface (API) of their managing deployment technology. To further decouple from specific deployment technologies, and to gain advantage from standardization, this work shows how EDMMi can be transformed to a standardized representation using the Technology and Orchestration Specification for Cloud Applications (TOSCA) standard which specifies the description of

Cloud applications in an interoperable manner [Org13b; Org20]. Based on the obtained standardized TOSCA instance model, the Management Feature Enrichment approach presented by Harzenetter et al. [HBL+19] is extended to support the enrichment of running applications with additional management functionalities. As a result, this work enables the addition of management features to running applications, regardless of their managing deployment technology. Since the presented concept derives a normalized and standardized instance model, this also enables the management of running applications in a single place, e. g., in a single dashboard, to ease the management of applications for enterprises that have different deployment technologies in operation. Concludingly, the feasibility of the presented approach is demonstrated via a prototypical implementation within the EDMM Transformation Framework [WBB+20] and the OpenTOSCA ecosystem [Bre+16]. The EDMM Transformation Framework is extended to automatically retrieve technology-specific instance information of a running application. The retrieved instance information is then transformed to the normalized EDMMi, which is then in turn transformed to a TOSCA instance model. Eventually, the obtained TOSCA instance model is enriched with executable management workflows using the OpenTOSCA ecosystem and an extension to the prototypical implementation of the Management Feature Enrichment approach. By executing the generated workflows, an application is manageable with additional management features, regardless of the technology that was used to deploy it.

## Outline

The remainder of this work is divided into the following Chapters:

**Chapter 2 – Foundations:** Fundamental knowledge for Cloud application deployment, its deployment models and deployment technologies is introduced.

**Chapter 3 – Related Work:** Here, research on related methodologies and their differences to the concept presented in this work are discussed.

**Chapter 4 – Approach:** This chapter is the main part of this work as it presents the approach.

**Chapter 5 – Implementation:** This chapter constitutes the prototypical implementation of the approach.

**Chapter 6 – Validation:** The prototypical implementation is validated in an experimental setting and threats to validity are discussed.

**Chapter 7 – Conclusion and Future Work:** A conclusion of the work is provided, followed by an outlook to future work.

# 2 Foundations

This chapter provides the foundations required to understand the remainder of this work. First, fundamental explanations on Cloud application deployment are provided. Further, deployment technologies that are used later on for the implementation and validation of the approach are briefly presented. As the concept of this work is built upon EDMM and TOSCA, the key aspects of both are outlined, including the EDMM Transformation Framework and the OpenTOSCA ecosystem. Concludingly, this chapter entails explanations on the Management Feature Enrichment approach utilized in the concept.

## 2.1 Cloud Application Deployment

Modern enterprise IT applications are typically characterized by large-scale, heterogeneous, distributed components, which impedes their management and operation [BBKL14a]. Oppenheimer et al. [Opp03] discovered that a key cause of the unavailability and failure of such large-scale applications is an operator error. To reduce such errors, Oppenheimer et al. [Opp03] propose to introduce and use tools that visualize configurations, data flow, and relationships among components. In addition, system misconfiguration could be prevented by automatically generating configuration files for components based on a high-level service architecture [Opp03]. Basically, Oppenheimer et al. [Opp03] describe the early idea of deployment technologies and their benefits.

Besides the need to eliminate operator errors, the necessity of automated management and deployment was further reinforced by the introduction and broad adaption of *Cloud Computing*. Cloud Computing changed the way how IT resources are used and managed: Instead of buying hardware of software upfront, resources are consumed in a utility-like fashion with a pay-per-use model [Ley09; MG+11]. To provide Cloud offerings in an economically feasible manner, Cloud provider pool such resources. However, since the management and operation of IT resources is one of the biggest cost factors nowadays, Cloud provider need to automate their deployment and management to enable the essential characteristics of Cloud Computing such as (i) on-demand self-service, and (ii) rapid provisioning [BBKL14a; MG+11]. Leveraging Cloud Computing and its essential characteristics is considered to be a key factor for success of enterprises, but also requires automated management [Gar10]. Especially since Cloud applications typically consist of a large amount of complex, composite, distributed components that interact with each other, their manual management and operation is a tedious, error-prone and complex process [HBL+19]. As a result, many technologies emerged that help to deploy Cloud applications in an automated manner to reduce costs and errors. Although the mechanisms, modeling languages, and features of such deployment technologies vary, their employed deployment automation strategy can be categorized into two main approaches: the (i) *declarative*, and (ii) *imperative* deployment

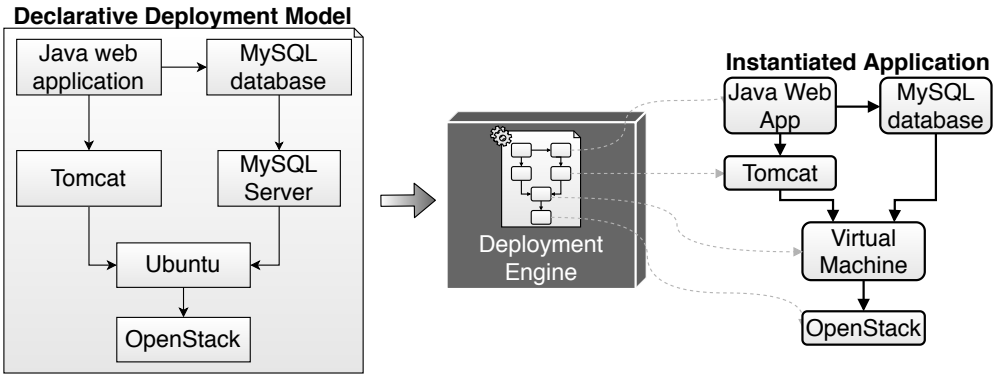


**Figure 2.1:** Declarative deployment approach.

modeling approach [EBF+17; WBF+19]. In a declarative deployment model, the desired state of an application is described by specifying its components, their configuration, and dependencies. Figure 2.1 depicts the approach of declarative deployment. On the left side, an exemplary deployment model is shown that describes the desired structure of an application to be deployed in a declarative manner by declaring its component and their relationships. The shown model expresses that a Java web application needs to be deployed on a Tomcat web server running on a Ubuntu VM which is provisioned on OpenStack. Further, it is modeled that the Java web application connects to a MySQL database that is hosted on the same Ubuntu VM. Another dependency in the illustrated declarative model is that the MySQL database requires a running MySQL server. To instantiate the described application, its declarative deployment model is fed into an engine that is able to interpret it, i. e., that derives the necessary steps required to instantiate the application as modeled. To determine the order of these steps, the structure is interpreted. In the depicted example, the Ubuntu VM needs to be deployed on OpenStack, before the Tomcat web server and the MySQL server can be installed. After the installation of the MySQL server, the MySQL database is instantiated. Then, the Java web application is deployed to the Tomcat web server as modeled. Once the database is up and running, the Java application connects to the database, for example, to retrieve or fetch data. Eventually, the desired state is achieved as described in the declarative model. Since the required steps to deploy an application are derived automatically, the declarative approach minimizes manual configuration and operation steps and thus, significantly eases the deployment. However, this approach is limited to common, well-known components since deployment engines cannot support an arbitrary amount of component types. Furthermore, to be able to interpret declarative models, deployment engines rely on a certain conformity of the model, thus, custom deployment logic is hardly applicable [BBK+14a; EBF+17]. Concludingly, declarative deployment is suitable for applications consisting of well-known components that require few or no individual customization [EBF+17].

In contrast, the imperative deployment approach builds upon a step-by-step specification of the necessary management tasks to instantiate an application. An imperative deployment model is a process model that defines (i) all activities to be executed, (ii) the order of execution, and (iii) the data flow between those activities to deploy an application in an automated manner [EBF+17]. Once specified, an imperative deployment model can be executed as process or workflow without involving any human tasks [EBF+17]. Since every

execution step is defined explicitly, it is possible to execute arbitrary deployment logic. Hence, the imperative approach is more suitable for complex applications that require application-specific customization [EBF+17]. However, such a precise and explicit definition also requires immense technical knowledge of management technologies [BBK+13].

### 2.1.1 Deployment Technologies

Deployment technologies aim to automate the deployment of applications. Since industry and research agree with declarative deployment models being the most appropriate approach for application deployment and configuration management, this work only focuses on declarative deployment in the following [HAW11; WBF+19].

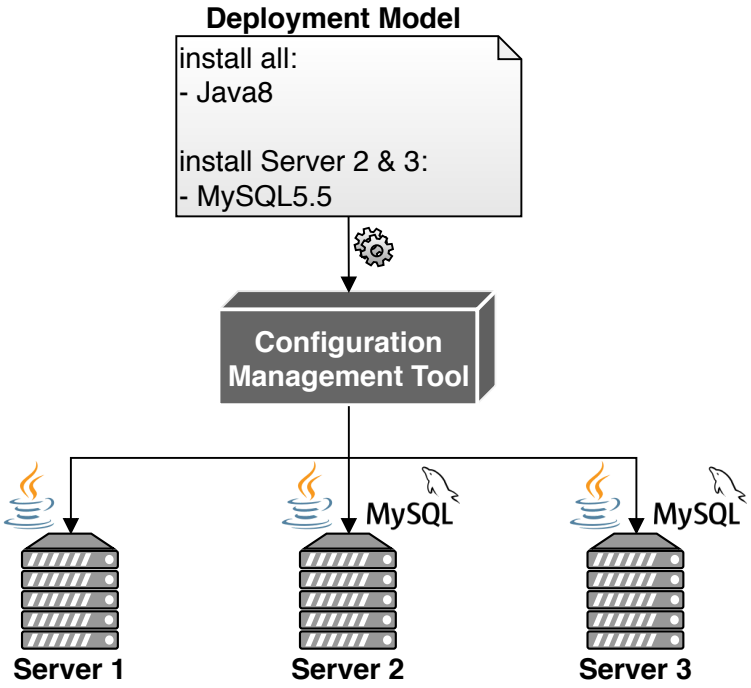


**Figure 2.2:** Configuration management approach.

Deployment technologies differ between (i) *orchestration technologies* and (ii) *configuration management tools* [MSK+18; WBF+19]. While orchestration technologies are used to provision application components such as infrastructure and software, configuration management tools are mainly used to configure already running components, for example, by installing or managing software on a server [MSK+18]. Figure 2.2 illustrates the approach of configuration management: A deployment model is defined that specifies the desired configuration of one or more servers in a declarative manner. In the depicted example, the deployment model declares to install *Java 8* on all managed servers, and further to install *MySQL 5.5* on two specific servers, i. e., *Server 2* and *Server 3*. The deployment model is then fed into a configuration management tool, which interprets the model to derive the

configuration on the servers as specified. As illustrated in Figure 2.2, after applying the configuration, Java 8 is installed on all servers, while MySQL 5.5 is installed on *Server 2* and *Server 3* as specified in the deployment model.

Besides the categorization into orchestration and management tools, Wurster et al. [WBF+19] observed during a systematic review that deployment technologies can be further divided into three categories, i. e., (i) general-purpose deployment technologies, (ii) provider-specific deployment technologies, and (iii) platform-specific deployment technologies. General-purpose deployment technologies are characterized by their support of single, multi, and hybrid Cloud deployments, as well as their support of Everything-as-a-Service (XaaS) offerings, for example, Software-as-a-Service (SaaS) offerings, and Platform-as-a-Service (PaaS) offerings [WBF+19]. On the contrary, provider-specific deployment technologies only support single Cloud deployments as they are offered by a specific provider, and thus only support services of the respective provider. However, different kinds of Cloud service offerings are supported [WBF+19]. Platform-specific deployment technologies are not restricted to a Cloud provider, but limited in the supported service offerings and platform bundles as they are bound to a specific platform. For instance, Kubernetes relies on a container runtime like Docker [Doc20a] to deploy applications. Thus, only artifacts supported by the specific platform can be deployed [WBF+19].

Based on the categorization of Wurster et al. [WBF+19], one representative deployment technology of each category is investigated closer in the following. To represent general-purpose deployment technologies, OpenStack Heat[1] was selected as it is open-source and one of the most popular Cloud computing platforms. Furthermore, it is well documented, has a large community, and offers a Representational State Transfer (REST) API for programmatic interaction. Regarding provider-specific deployment technologies, AWS CloudFormation is investigated closer, since it is bound to one of the most popular Cloud providers, i. e., AWS, and is largely used in industry. Kubernetes was chosen to cover platform-specific deployment technologies, since 78% of companies using containers in production are using it [Clo19]. Since the selected deployment technologies are solely orchestration tools, Puppet was included additionally to cover configuration management. In the following, the concepts of the four deployment technologies will be explained briefly.

#### OpenStack Heat

OpenStack Heat is a general-purpose, declarative deployment technology. It aims to automate the management of the whole lifecycle of infrastructure and applications within OpenStack Clouds, i. e., their deployment, management, and termination [Ope20a]. To enable this, OpenStack Heat employs a declarative deployment model called *Heat Orchestration Template (HOT)* that describes a Cloud application to be deployed by specifying its infrastructure resources, their configuration, and dependencies. A HOT is in a human-readable YAML format. Listing 2.1 shows a simple HOT file that deploys a single compute instance. As depicted, a HOT file specifies resources to be deployed in a declarative manner by stating their name, type, and properties [Ope20c]. For example, Listing 2.1 declares

[1] https://wiki.openstack.org/wiki/Heat

**Listing 2.1** A simple HOT file that deploys a single compute instance [Ope20c].

```
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      image: ubuntu-18.04
      flavor: m1.small
```

a single resource called *my_instance*, which is of type *OS::Nova::Server*, i. e., a server instance in an OpenStack Cloud. OpenStack Heat supports a wide range of infrastructure resource types which are available at the OpenStack documentation [Ope20d]. The type of a resource specifies its semantics, and its configuration is defined by properties [Ope20a; Ope20c]. In the shown example, the *OS::Nova::Server* resource features properties that specify its Operating System (OS) image, and its flavor. Moreover, it is also possible to define dependencies between resources by declaring that a resource is *required by* another resource. For example, to specify that a volume is connected to a server, the server may feature a *required by* attribute with the name of the volume as value [Ope20a].

Multiple resources that are logically cohesive and form an application are called *stack* in Openstack Heat [Ope20a]. To instantiate a stack, OpenStack Heat provides an orchestration engine [Ope20a]. To trigger the engine, it is possible to upload a HOT file that describes the stack either manually or in an automated manner using the provided REST API. Once the orchestration engine receives the declarative deployment model, the steps required to deploy the specified resources are derived automatically. For instance, the deployment model shown in Listing 2.1 triggers operations to instantiate an *OS::Nova::Server* instance according to the declared configuration, i. e., a Ubuntu VM with the flavor *m1.small*. Since OpenStack Heat allows the management of the whole lifecycle of an application, it is possible to modify the HOT file of a running application to update it [Ope20b]. Considering the example of Listing 2.1, it is possible to increase the RAM of the running server instance by updating the flavor property of the deployment model.

#### Puppet

Puppet is a general-purpose, declarative configuration management tool. It aims to automate the configuration and management of running infrastructure like servers [Pup20a]. To enable this, Puppet employs a master-agent architecture, where a master node controls one or more agent nodes [Pup20a]. The Puppet master stores the declarative deployment models that describe the desired configuration of its controlled agents. A deployment model in Puppet is a human-readable text file which is called *manifest*. A manifest describes resources to be configured in a declarative manner by specifying the (i) type, and (ii) properties of each resource [Pup20a]. Listing 2.2 depicts an exemplary Puppet manifest that copies a file from a Puppet master to a Puppet agent, where *file* is the

**Listing 2.2** Simple puppet manifest to copy a file from a master to an agent.

```
file { 'sample.war':
  path    => '/var/lib/tomcat8/webapps/sample.war',
  ensure  => file,
  source  => "puppet:///modules/copy_sample/sample.war",
}
```

resource type which defines the semantics of the resource to be configured [Pup20c]. For the shown example, the *path* property declares the target path at the Puppet agent the file is copied to, while the *source* property describes the source location of the file at the Puppet master [Pup20c]. Besides the configuration of files, Puppet supports the automated management of a plethora of resource types, e. g., *package*, or *exec*. For instance, the *package* type is used to install packages such as Java, while the *exec* type is used to execute commands on a Puppet agent, e. g., to unzip a copied file. To apply the resource configurations declared in a manifest, the Puppet master employs a *main manifest*, that specifies which manifest, i. e., configuration, is applied to which agent. For instance, it is possible to apply a manifest to all managed agents, or only to a dedicated agent. Based on the main manifest, the process of applying the configuration is initiated, which is visualized in Figure 2.3. First, a Puppet agent sends so-called *facts* about itself that are required by the Puppet master to apply a configuration, e. g., its hostname which is used to check whether a manifest applies to an agent. In the second step, the Puppet master compiles the manifests into a *catalog* that describes the desired state of the agent in a declarative manner [Pup20a]. Upon receiving the catalog, the agent translates it into specific commands to be executed to achieve the state specified in the catalog, e. g., to copy a file as of Listing 2.2. Per default, Puppet applies the resource declarations in the order in which they appear in a manifest, but it is also possible to declare relationships between resources using the *required* attribute. After the catalog has been applied, a report is generated indicating whether the desired state was achieved or not. Such reports, as well as facts and catalogs are stored in the Puppet database (PuppetDB) on the Puppet master. The Puppet master also exposes an API to query the PuppetDB.

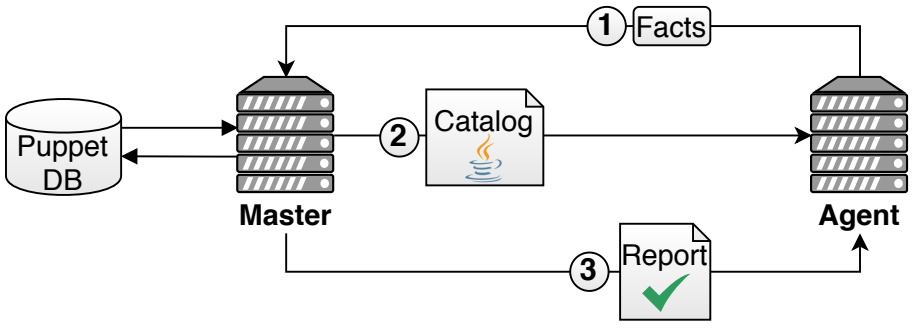


**Figure 2.3:** Architecture and control flow of Puppet.

**Listing 2.3** Sample CloudFormation template that deploys an AWS EC2 instance.

```
Resources:
  Ec2Instance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0ff8a91507f77f867
```

#### Amazon Web Services (AWS) CloudFormation

AWS CloudFormation is a provider-specific, declarative deployment technology bound to the Cloud provider AWS. It aims to automate the management of resources provided by AWS, e. g., EC2 servers, or AWS Relational Database Services (RDSs). To enable this, AWS CloudFormation employs a declarative deployment model called *AWS CloudFormation template*, or *template* in short. A template describes an application by specifying its AWS resources to be deployed and their configuration [Ama20d]. Listing 2.3 displays such a template that deploys an AWS EC2 server instance. As depicted, a template declares its resources, their type, and properties in a declarative manner. In the example, the resource *Ec2Instance* is of type *AWS::EC2::Instance*, which specifies the semantics of the resource [Ama20c]. Further, properties are defined for its configuration, i. e., an image referring to the OS of the server. Besides attributes to declare its configuration, a resource may express a dependency via a *Depends On* attribute, e. g., to describe that a *AWS::EC2::Instance* depends on a storage volume.

To instantiate such a template, it is fed to the AWS CloudFormation orchestration engine which interprets it to derive the steps required to deploy the infrastructure resources as specified. For instance, when instantiating the template of Listing 2.3, the orchestration engine derives operations to create a VM in the AWS Cloud with the specified OS image. Once instantiated, a template is called a *stack* [Ama20b]. A template can be fed to the orchestration engine manually via the AWS console, via its provided REST API, or by using the AWS Command Line Interface (CLI). Further, it is possible to perform updates on a stack by modifying its template, for example by allocating more RAM to the AWS EC2 instance to handle increased workload.

#### Kubernetes

Kubernetes is a platform-specific, declarative deployment technology that orchestrates the deployment of containerized applications and allows to manage their whole lifecycle [Kub20d]. Kubernetes aims for high availability and scalability by providing interfaces to easily scale up and down application components, and replicate them [Kub20f]. Similar to Puppet, Kubernetes follows a agent-master architecture where the master is responsible for the management of a cluster of nodes, i. e., scheduling applications, maintaining the desired state of applications, scaling, or updating them [Kub20b]. Such a cluster can be deployed on a virtual or physical machine [Kub20c]. A node of a cluster is a VM or a physical com-

puter that runs a *Kubelet*, which is an agent to communicate with the Kubernetes master and manage the node [Kub20c]. Since Kubernetes base concepts rely on containerized applications, each node needs to run a container runtime, e. g., Docker[2] [Kub20c].

To describe the desired state of a Kubernetes cluster, *Kubernetes objects* are used, including *pods*. Kubernetes objects build an abstraction layer to decouple containers from individual hosts. Pods are the most basic execution unit of Kubernetes and comprise containers that operate together to form an application [Kub20e]. Moreover, pods are declared in *pod templates*, which are text files in YAML format that describe the desired state of a node in a Kubernetes cluster. Containers are defined within the *spec* section of the *template* and, among others, contain properties that declare the image of a container, ports, or some commands to execute within a container.

To make use of the easy scalability of Kubernetes applications, deployments are used to bundle multiple pods and define their replication strategy. Deployments are defined in a similar, declarative manner as pods. By using deployments, Kubernetes enables the creation and management of replication sets of pods to easily adjust to changing workloads or replace failing pods automatically to ensure high availability. Whenever a pod fails, Kubernetes makes sure that a new one is started immediately to always have as much replicas as specified. It is also possible to change a running application, for example, by increasing or decreasing the number of replicas in the respective deployment model YAML file to start or destroy a pod.

## 2.2 Essential Deployment Metamodel (EDMM)

Wurster et al. [WBF+19] conducted a systematic review of the most popular deployment technologies to provide a basis for the comparison of features and mechanisms to ease the selection of the most suitable deployment technology. They identified following technologies as most relevant: (i) Kubernetes [Kub20a], (ii) Chef [Che20], (ii) Ansible [Red20], (iv) Puppet [Pup20b], (v) Terraform [Has20], (vi) Juju [Can20], (vii) Cloudify [Ltd20], (viii) OpenStack Heat [Ope20a], (ix) CFEngine [Nor20], (x) Docker Compose [Doc20b], (xi) Azure Resource Manager [Mic20], (xii) SaltStack [Sal20], and (xiii) AWS CloudFormation [Ama20a]. By extracting the essential parts that are supported by all aforementioned deployment technologies, Wurster et al. [WBF+19] derived the Essential Deployment Metamodel (EDMM), which is a normalized deployment metamodel. EDMM aims to provide a deployment-technology-independent model that eases the migration between deployment models of different deployment technologies [WBF+19] Figure 2.4 depicts EDMM in its entirety. The topmost entity of EDMM is a deployment model, which describes the desired state of an application in a declarative manner by the means of properties and model entities [WBF+19]. Model entities are composed of a model element, i. e., a component or relation, and its type. A component is defined as a physical, functional, or logical unit of an application, for example, a database that stores user data. The semantics of a component is defined by its component type, an entity that can be reused across deployment models, e. g., two database components in different deployment models may

[2] https://www.docker.com/

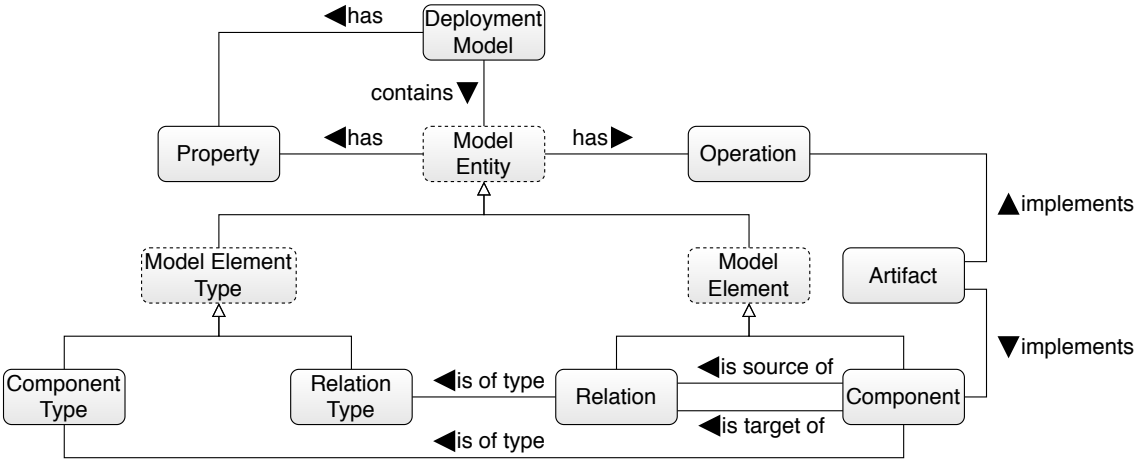


**Figure 2.4:** Essential Deployment Metamodel (EDMM) from Wurster et al. [WBF+19].

share the same component type MySQL, but store different data. To provide a normalized metamodel, Wurster et al. [WBF+19] introduced a set of normative component types for EDMM. Types included in this set are, for example, *Compute*, *Software_Component*, or *Database*. To model a connection between two components, for instance, between a component of type *Software_Component* and a component of type *Database*, relations are used. A relation is defined as directed physical, functional or logical dependency between exactly two components [WBF+19]. Similar to components, relations have types that specify their semantics. For instance, a software component that retrieves data from a database must establish a connection to the database to issue queries, i. e., the software component needs an outgoing relation of type *ConnectsTo* with the database component as target. To manage such components and relations, EDMM employs operations. An operation is defined as executable procedure performed to manage a component or relation [WBF+19]. An example for an operation is an installation script that installs a software component on a VM. Typically, operations require information about the components to be managed, for example the IP address of a VM where a software needs to be installed. To express such information, EDMM features properties. Properties are used to express the desired state, or the configuration of a component, relation or deployment [WBF+19].

Further, Wurster et al. [WBF+19] presented how the concepts of EDMM can be semantically mapped to the 13 identified deployment technologies. Also, a mapping of EDMM to TOSCA was provided as various technologies support the TOSCA standard [WBF+19]. With the introduction of the *EDMM Modeling and Transformation System*, Wurster et al. [WBB+20] presented tooling to graphically model applications based on EDMM and transform deployment models to each of the 13 deployment technologies by the means of a CLI. The EDMM Modeling and Transformation system consists of (i) a modeling tool and (ii) a transformation framework as depicted in Figure 2.5. The modeling tool is a web-based environment to graphically model technology-independent EDMM deployment models. As depicted, an EDMM model can be exported from Winery as a text file in YAML format. The exported EDMM model is then fed to the transformer, a CLI tool that is able to transform EDMM into concrete deployment technologies. The transformer mainly consists of two parts, (i) the *model importer*, and (ii) the *transformation logic*, as depicted

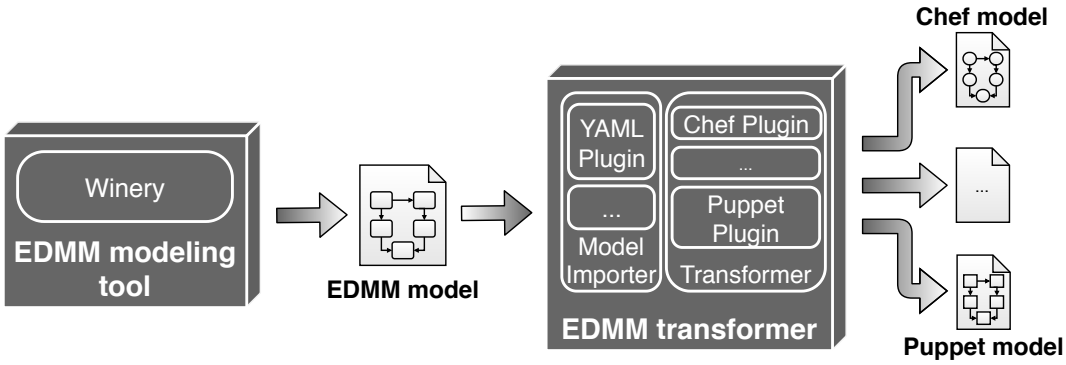


**Figure 2.5:** Architecture of the EDMM modeling and transformation system from Wurster et al. [WBB+20].

in Figure 2.5. To process and prepare the input EDMM YAML file for the transformation, the model importer implements a YAML plugin. In the following, the processed model is passed to the transformation logic, which is a plugin-based, easily extendable logical unit that implements the transformation of EDMM models to concrete deployment technologies. The output of the transformation is a fully-executable deployment model of the selected technology, which can be specified when calling the CLI [WBB+20].

## 2.3 Technology and Orchestration Specification for Cloud Applications (TOSCA)

Technology and Orchestration Specification for Cloud Applications (TOSCA) [Org13b] is a standard introduced by the Organization for the Advancement of Structured Information Standards (OASIS) to describe Cloud applications in a portable and interoperable manner [Org13b; Org20]. The goal of TOSCA is to represent Cloud applications in a platform and vendor agnostic way. To achieve this, TOSCA describes applications based on two main concepts: (i) The structure of an application by the means of a *topology* that describes an application's components and their interplay, and (ii) *plans* to define its management behavior [Org13b]. The structure of an application is described by the means of a *topology template*, which is, as illustrated in Figure 2.6, a graph that consists of nodes representing application components and directed edges that define their interplay. The nodes of a topology template are called *node templates*, while the directed edges are called *relationship templates*. To define the semantics of node templates and relationship templates respectively, TOSCA employs node types and relationship types. As visualized in Figure 2.6, types are equipped with properties and interfaces [Org13b]. Properties define the state or the characteristic of a node template or relationship template, for example, the port where a database node template listens to requests. Interfaces contain operations that offer management behavior for a node type or relation type, e. g., the interface of a software component may contain an operation that defines its installation procedure as an executable. For instance, Figure 2.7 shows a topology template of a simple application where a software component running on a web server connects to a database while both

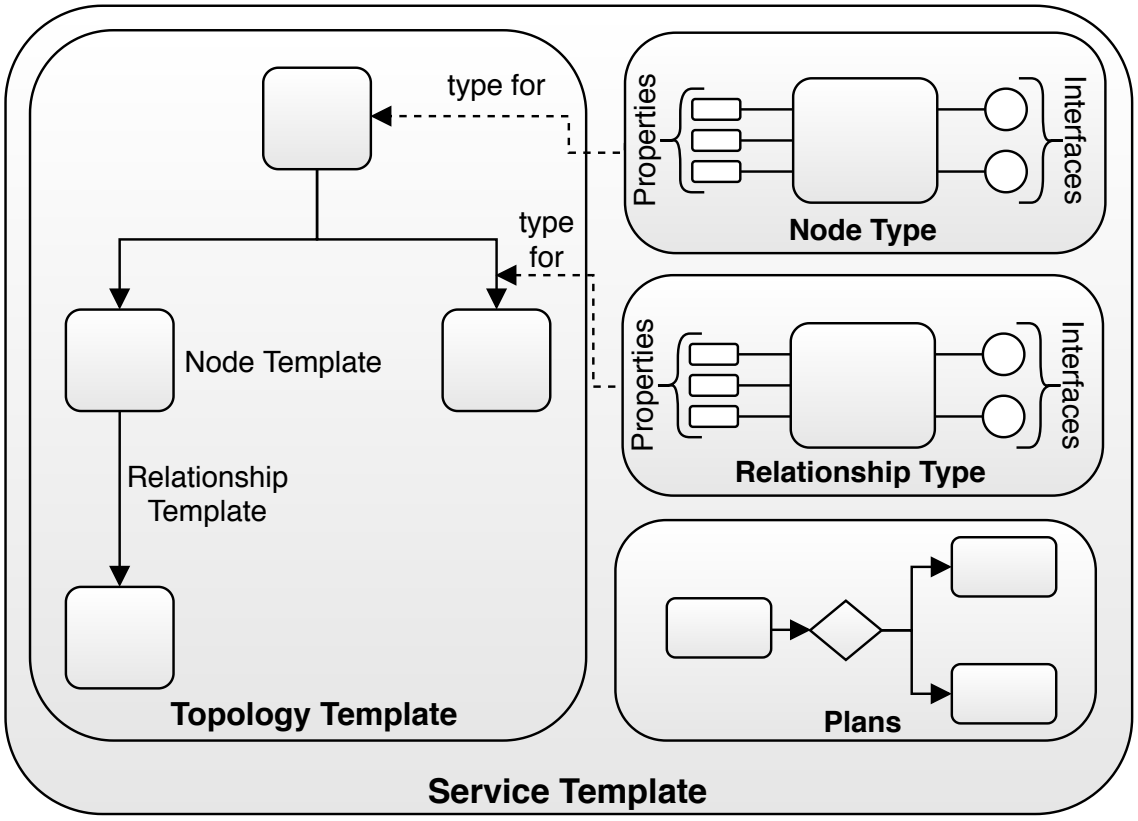


**Figure 2.6:** Composition of a Service Template [Org13b].

application components are hosted on a VM which is provisioned in an OpenStack Cloud. To represent this, the topology template contains a node template of type *JavaApp* that is connected with a relationship template of type *ConnectsTo* to a node template of type *MySQL-5.5*. Besides that, the software component has an outgoing *HostedOn* relationship to a node template of type *Tomcat8*, since the software component is served on a Tomcat web server. Further, the database node template has an outgoing *HostedOn* relationship template to a MySQL server which is required to run the database. Both the Tomcat web server, and the database server are hosted on a Ubuntu VM which is provisioned on an *OpenStack* node template. In this topology template, various properties are defined, for example, the port of the database node template, which indicates to which port the software component node template connects to for the retrieval of data. Besides properties, node templates may contain so-called *deployment artifacts*, which represent any artifact needed to deploy a component, for example, a configuration file, executable, or a script [Org13b]. In the scenario of Figure 2.7, the software component node template has such a deployment artifact; the executable Web Application Resource (WAR) file of the Java application. Additionally, node types may contain *implementation artifacts* that describe operations to make a component manageable [BBH+13]. For example, the Ubuntu 18.04 node type from Figure 2.7 may be equipped with implementation artifacts that respectively implement an executable management operation to (i) install, (ii) configure, (iii) start, and (iv) stop the VM. To execute such operations and make applications described as service template manageable, *plans* are used [Org13b]. Plans are defined as process models that describe

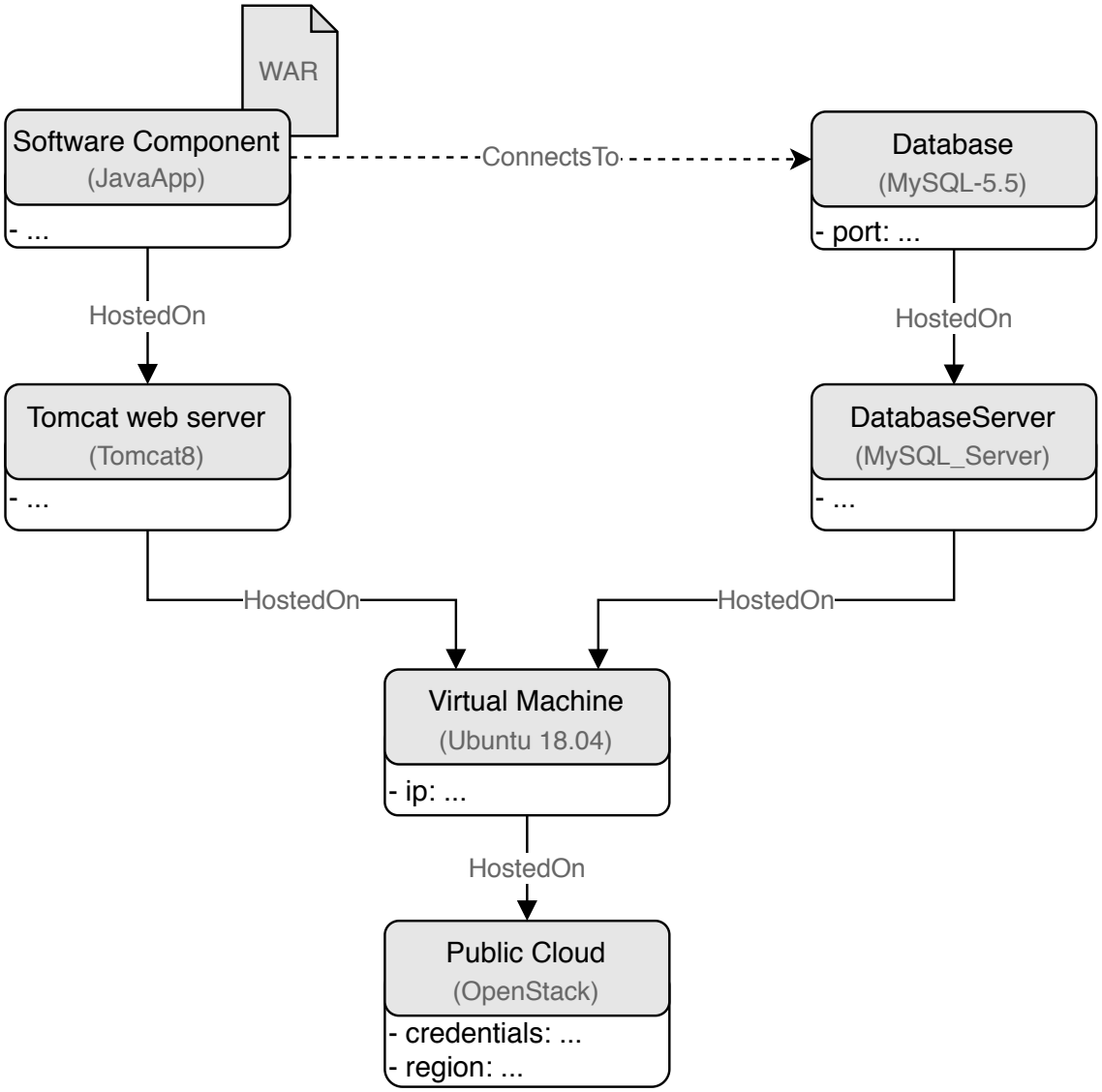


**Figure 2.7:** Exemplary Topology Template.

a workflow of one or more steps to realize management of a service template [Org13b]. For instance, a plan may describe every step required to install and start an application described as a service template in an automated manner. To ensure interoperability and portability of plans, TOSCA makes use of existing standardized process models such as BPEL [Org07] or BPMN [Obj11]. To bundle plans, files like deployment artifacts, and all TOSCA definitions belonging to a Cloud application described in TOSCA, a Cloud Service ARchive (CSAR) is used [Org13b]. TOSCA definitions refer to all elements needed to define a TOSCA service template, e. g., the service template itself and its node templates and relationship templates [Org13b]. To instantiate a CSAR, it is passed to a TOSCA runtime, which can process TOSCA service templates in an imperative or declarative manner. When using a runtime with imperative processing, the CSAR needs to contain the management plans for an application, while a runtime that supports declarative processing interprets

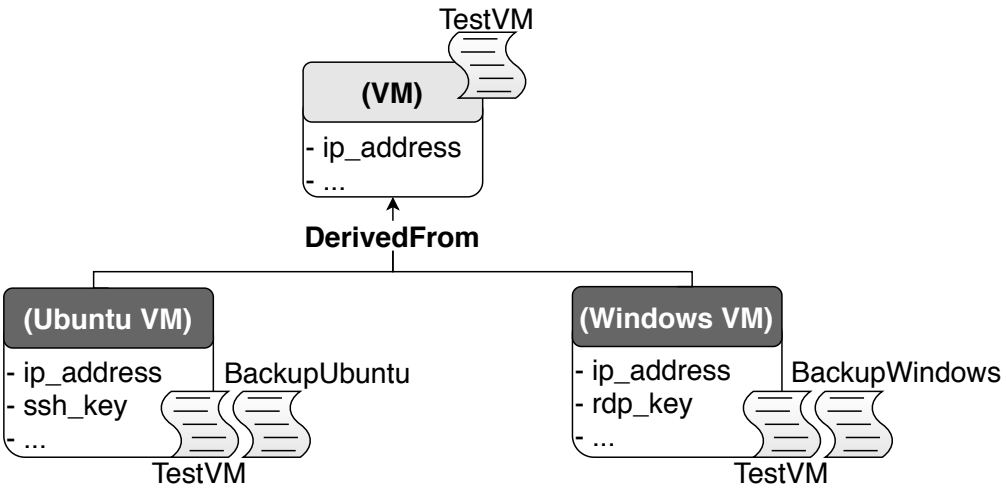


**Figure 2.8:** Exemplary TOSCA type hierarchy with a Ubuntu VM node type and a Windows VM node type deriving from a VM supertype.

the topology template to derive the necessary steps and their order to achieve the declared state. For instance, a runtime that supports declarative processing may generate a plan to install and start the described application in an automated manner. When executing such a plan, a service template is instantiated, i. e., all modeled node templates and relationship templates are installed, configured, and started as specified. To represent an instance of a service template, TOSCA describes an instance model [Org13a]. To clearly distinguish between models and instances of an application described in TOSCA, the suffix *instance* is appended to all template elements, e. g., an instantiated node template is called *node template instance*. When instantiated, property values of node and relationship templates may change to represent the current value. For example, the property of a node template of type VM specifying the IP address may update after instantiation to the real-world IP address of the resource. Moreover, states are assigned to node templates and relationship templates to describe the current step in the lifecycle of a resource [Org13a; Org20]. For a example, while executing the build plan of a service template, a node template may traverse several states, such as *creating*, *configuring*, or *started*. Once all node and relationship templates of a topology template are instantiated, i. e., are in *started* state, the represented Cloud application can be managed further by custom management plans, for example, with a process model that scales an application up to match a specific workload.

Furthermore, TOSCA features inheritance which allows node and relationship types to form type hierarchies. To enable this, each TOSCA type has a nested *DerivedFrom* element that allows to specify supertypes. For instance, a database supertype may define properties and operations applicable to different types of databases. Hence, a MySQL node type that specifies the database supertype in its DerivedFrom element inherits all elements defined by the supertype. This is especially useful to identify semantically similar types. Figure 2.8 illustrates an exemplary type hierarchy in TOSCA. As depicted, a VM super type defines properties like its IP address, and management operations, e. g., to test the availability of a VM. Both the Ubuntu VM node type, and the Windows VM node type are derived from the VM supertype. This means, that they inherit the properties, and management

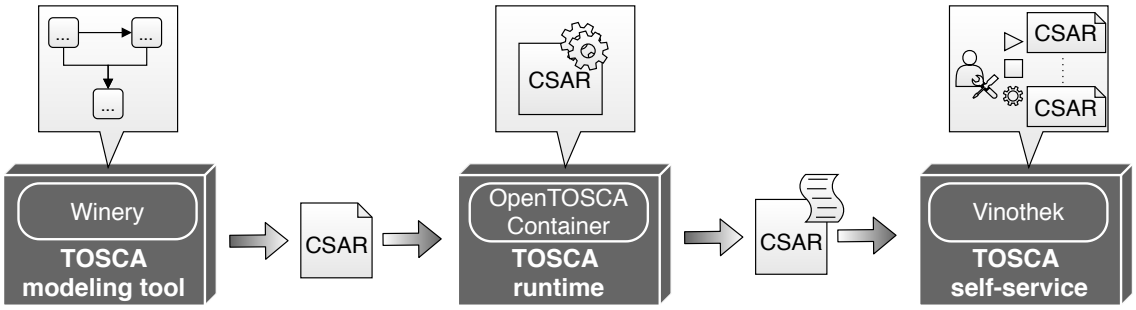


**Figure 2.9:** Overview of the architecture and workflow of the OpenTOSCA ecosystem.

operations of the VM node type. Further, a node type may extend its supertype with additional properties and management operations. In the depicted example, both the Ubuntu VM node type and the Windows VM node type extend the VM node type with a management operation to backup a VM of it OS respectively.

### 2.3.1 OpenTOSCA

OpenTOSCA [Bre+16] is an ecosystem that allows the modeling and automated deployment and management of Cloud applications according to the TOSCA specification described in Section 2.3 [Org13b]. The architecture and workflow of OpenTOSCA is three-fold as illustrated in Figure 2.9. (i) *Winery* [KBBL13], a web-based environment, is used to model the structure of a Cloud application according to the TOSCA specification. Once modeled, the user exports the application as a CSAR with all TOSCA definitions and artifacts required to enable automated management. The TOSCA definitions and artifacts are available via a repository within Winery. The repository contains, for example, TOSCA node and relationship types and their offered management operations. Then, the CSAR is imported into the (ii) *OpenTOSCA Container* [BBH+13], a TOSCA runtime to deploy and manage TOSCA applications. The OpenTOSCA Container generates executable management plans, for example, to instantiate or terminate an application. Management plans are generated by coordinating all related management operations of node templates to a holistic workflow, for example, by composing all operations of all node templates of a service template that relate to the installation of an application. In this process, the structure is interpreted to determine the execution order of the management operations. For instance, consider the topology template of Figure 2.7: The VM must be provisioned before installing the web server and database server, thus the OpenTOSCA Container interprets the topology accordingly to derive the correct execution order to satisfy all dependencies. The generated workflows can be executed via (iii) *Vinothek* [BBKL14d], a web-based self-service portal for the interaction with the OpenTOSCA Container.

### 2.3.2 Management Feature Enrichment

Although the automated deployment of Cloud applications is enabled by a plethora of different deployment technologies, the automated management of deployed applications is only covered partially by such technologies. Typically, Cloud provider or deployment

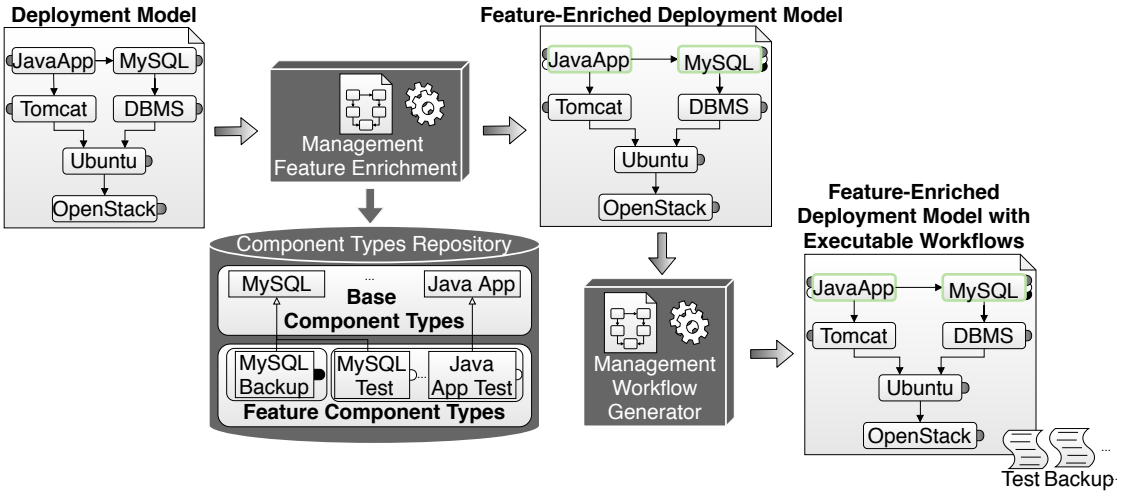


**Figure 2.10:** Overview of the Management Feature Enrichment approach of Harzenetter et al. [HBL+19].

technologies may support provider-specific management functionalities like scaling a VM. However, the support of holistic management functionalities affecting multiple components possibly distributed across different environments is limited [HBL+19]. Especially the holistic management of Cloud applications containing components that reside in different environments, i. e., a Multi-Cloud scenario, is not supported since each Cloud provider or provider-specific deployment technology only supports management within their own environment. Using imperative workflows, it would be possible to define custom logic that enables holistic management of such applications. However, such workflows may become outdated quickly as they are tightly coupled to the deployment model, which may change [HBL+19]. Besides that, defining imperative workflows is tedious, error-prone and requires deep technical knowledge [BBK+13; BBKL13b; KNK16].

To tackle mentioned issues, Harzenetter et al. [HBL+19] propose an approach to automatically generate executable workflows that enable holistic management functionalities for applications. Figure 2.10 depicts a general overview of the approach of Harzenetter et al. [HBL+19]. As illustrated, initially, a declarative deployment model is retrieved that describes the structure of an application that needs to be enriched with management functionalities. The components of the declarative model are based on component types that only define basic lifecycle operations like creating or deleting an application component. Such component types are called *base component types* [HBL+19]. The example scenario shows a deployment model where a Java application is deployed to a Tomcat web server running on a Ubuntu VM which is hosted on OpenStack. Moreover, the Java application connects to a MySQL database which is running on a database server hosted on the Ubuntu VM. The base component types of the application reside in a *Component Types Repository*, for example as depicted in Figure 2.10 a MySQL node type. Additionally to the base component types, the repository contains *feature component types*, that provide type-specific management functionalities by the means of operations, e. g., to test or back up a component. Feature component types extend base component types, and thus, form a type hierarchy as described in Section 2.3. The Component Types Repository is then

utilized by the *Management Feature Enrichment* component to find feature component types that extend the base component types used in the original deployment model. If the Management Feature Enrichment component finds a feature component type for a given base type, a new component type is generated that unifies all management functionalities of both node types, i. e., the basic lifecycle operations of the base component type, and the additional management features of the feature component type. In the following, the generated component type replaces the base component type in the declarative deployment model. This process emits a feature-enriched deployment model, which is passed to the *Management Workflow Generator* that generates imperative workflows based on the enriched management operations. The Management Workflow Generator is plugin-based, where a plugin implements logic that decides in which order management operations are executed. For example, a test plugin generates an imperative workflow that executes all management functionalities related to testing, i. e., all white operations of the feature-enriched deployment model in Figure 2.10 [HBL+19]. As a result, the deployment model is packaged with the generated management workflows, e. g., with a test, and a backup workflow. Once the deployment model is deployed, the resulting management workflows can be executed using a workflow engine.

This approach eliminates the manual implementation of imperative workflows and enables holistic management by orchestrating type-specific management operations to a fully-executable workflow. By providing a prototypical implementation within the OpenTOSCA ecosystem, Harzenetter et al. [HBL+19] proved the feasibility of the approach. To store base component types and their extending feature component types, the OpenTOSCA Container was extended with a *Container Repository*. Once a CSAR is uploaded to the OpenTOSCA Container, this repository is used to identify suitable feature component types based on its base types of the deployment model contained in the CSAR as described before. The resulting feature-enriched deployment model is then equipped with executable BPEL workflows that contain the enriched management operations. Afterwards, the generated enriching workflows can be triggered using the Vinothek [HBL+19].

# 3 Related Work

This chapter presents work that is related to the approach proposed in this thesis and discusses differences to our concept. In the first section, related concepts concerned with the retrieval of instance information of running applications are described. In the following, various work for the management of running applications are discussed.

## 3.1 Retrieval of Instance Information

A plethora of approaches investigated the retrieval of instance information of running applications. For instance, Binz [Bin15] and Binz et al. [BBKL13a] presented a plugin-based concept to identify components and configurations of a running application in an iterative manner. The plugins are used to determine the type of components to define their semantics. This is done via specific crawlers that identify components of a running application, and use the gathered information iteratively to further identify components, or refine already detected components based on the previous iteration.

Besides that, Holm et al. [HBLE14] showed a concept that uses network scanners to identify components of a running application, such as infrastructure and software. Similar to the concept of this work, Holm et al. [HBLE14] transform the retrieved information to a standardized model, i. e., an ArchiMate[1] model capturing an enterprise architecture.

Further, Farwick et al. [FAB+11] introduced a similar concept to automate the management of running application by identifying its components. Motivated by the discrepancy between enterprise architecture models and the actual real world representation, they aim to precisely gather information about components of running applications. Similar to this work, Farwick et al. [FAB+11] try to automate the maintenance of running applications especially as it is time-consuming when performed manually.

Contrary to the concept presented in this work, the aforementioned approaches use custom scanner and crawler software to identify components of a running application, while this work leverages the API of the deployment technology an application was deployed with. Besides following a more generic approach in retrieving instance information, the concept of this work currently relies on the data of the used deployment technology and is therefore not able to retrieve instance information of manually deployed applications. However, the approach is easily extendable to also support the retrieval of instance information of running applications that were manually deployed by combining it with the concepts of the presented related work [BBKL13a; Bin15; FAB+11; HBLE14].

[1] https://pubs.opengroup.org/architecture/archimate3-doc/

By introducing application auto-discovery, Machiraju et al. [MDW+00] introduced a more generic approach to generate models of running applications. As input for the proposed auto-discovery engine, models are used that specify what to discover, and how to discover it. However, this limits the approach since Machirau et al. [MDW+00] must specify the components to be discovered beforehand.

Brogi et al. [BCS17] proposed a concept to derive TOSCA node types from running Cloud services similar to the mapping from technology-specific component types to standardized component types presented in this work. However, they solely focus on the identification of component types, instead of retrieving instance information of such components as this work proposes. Similarly, Endres et al. [EBLW17] crawl repositories to transform well-known artifacts of different Cloud deployment technologies into a technology-agnostic model. Resulting from this, executable deployment models based on the TOSCA standard are generated. Since both Brogi et al. [BCS17] and Endres et al. [EBLW17] utilize TOSCA types to derive a standardized model, it is possible to combine the concepts with the approach of this work which also leverages a TOSCA repository and its types.

## 3.2 Management of Cloud Applications

Numerous work was done in the field of automated management for Cloud applications since it is an error-prone, complex, and time-consuming process when performed manually [BBK+13; BBKL13b; HBL+19; KNK16; Opp03]. For instance, EDMM is capable of providing a deployment-technology-agnostic declarative deployment model as 13 of the most relevant deployment technologies can be mapped to it [WBF+19]. With the EDMM modeling and transformation framework, Wurster et. al [WBB+20] presented an implementation to create EDMM models that can be transformed to a plethora of concrete deployment technologies by the means of a plugin-based CLI. This eliminates the fear of a possible deployment technology lock-in and enables the migration between deployment technologies [WBF+19]. However, this approach is only feasible if an application is modeled with EDMM. Existing deployment models of a concrete deployment technology, e. g., AWS CloudFormation, are currently not convertible to EDMM. Further, EDMM is only able to describe the deployment model of an application, but does not provide any mechanisms to manage an already deployed application. Especially Cloud applications that are already deployed and running cannot be represented in EDMM. This is a problem since the deployment of most enterprise applications is already automated with a declarative deployment model of a concrete technology, and the manual transformation to EDMM may be tedious and error-prone [WBB+20; WBF+19]. Moreover, EDMM is only able to express a deployment model, but is missing a model for deployed applications to enable the deployment technology independent management of running Cloud applications.

Further, Binz et al. [BBKL14b; BFL+12; Bin15] presented an approach to describe enterprise applications to ease management of entire IT infrastructures and decrease operational cost. By introducing Enterprise Topology Graph (ETG), Binz et al. proposed a means to capture multi-environment IT landscapes based on a formal graph describing components and their interplay [BFL+12]. The resulting ETG consolidates IT resources from multiple environments, including Multi-Cloud or Hybrid-Cloud scenarios. However,

ETG is meant to represent enterprise applications that are running whereas TOSCA describes the deployment model of an application [BBKL14b; BFL+12]. Despite that, the work proposed by Binz et al. [BBKL14b; BFL+12; Bin15] is focusing on migrating existing applications between environments, while the approach of this thesis aims at transforming the instance model without changing the actual environment of it. However, the representation of an application instance by the means of a graph is similar to the notation of the instance metamodel proposed in this work.

Breitenbücher et al. [BBK+14b; BBKL13b; BBKL14c; Bre16] introduced the generation of executable workflows based on patterns and generic declarative models to enable automated management of Cloud applications. Similar to the approach of this work, additional management features are applied to running applications using its instance information. While the concept of this work uses the types of application components to derive additional management operations, Breitenbücher et al. [BBK+14b; BBKL13b; BBKL14c; Bre16] explicitly define management patterns to be applied to an application. Based on such patterns, a declarative model is generated that describes the new desired state, e. g., by inserting additional components to scale out an application [BBKL13b]. In this new model, each component that employs new management behavior needs to be annotated. Then, a plan generator interprets the model and its annotations to generate fully-executable management workflows by orchestrating so-called *management planlets*. Management planlets are small reusable workflows comprising multiple management operations that form a logical unit. The execution order of the planlets within the management workflow is defined by the generated declarative model. This is similar to the approach of this work where multiple management operations are orchestrated to a holistic workflow to enrich a running application with additional management features.

# 4 Approach

This chapter constitutes the main part of this work by presenting the approach. At first, a conceptual overview of the individual steps that make up the concept is given, which is illustrated in Figure 4.1. As depicted, the concept consists of six steps.

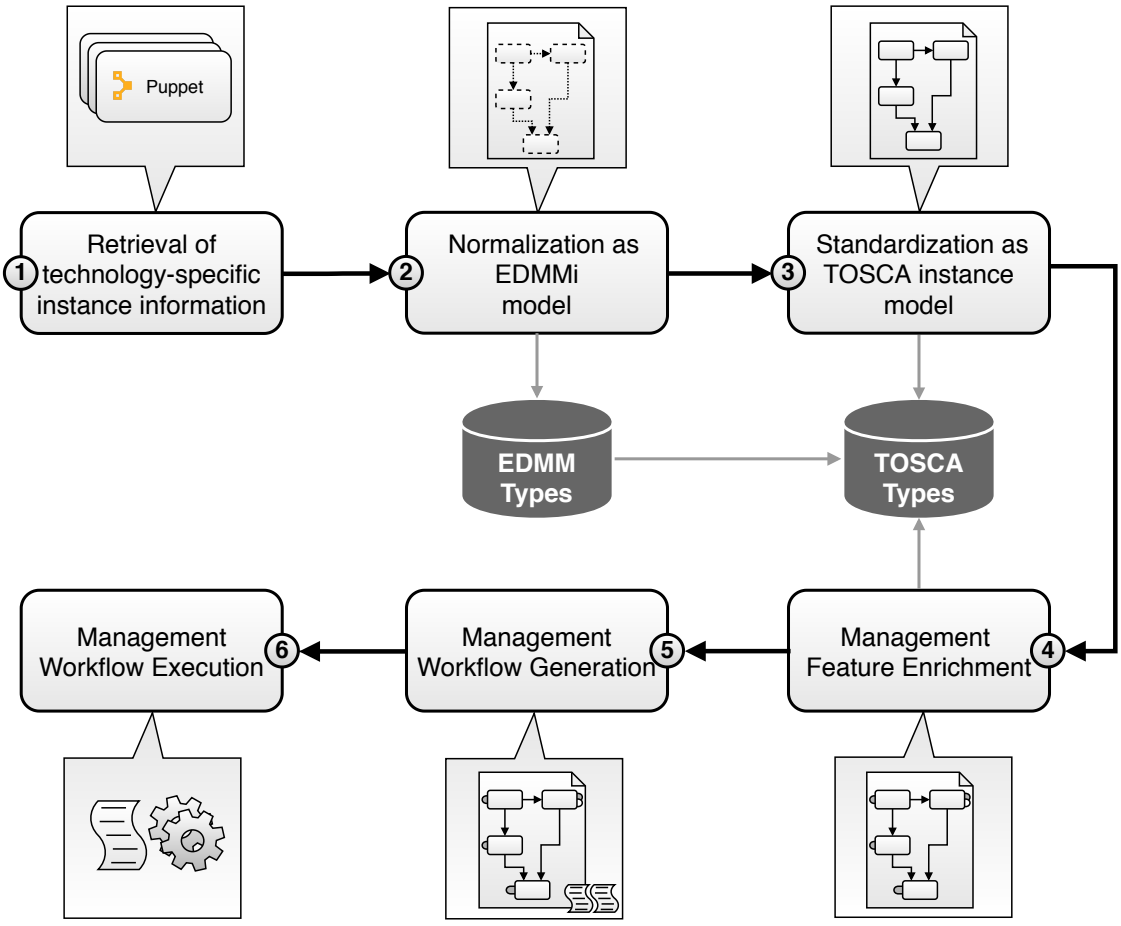


**Figure 4.1:** Overview of the approach.

In the first step, technology-specific instance data from a running application that was deployed using a deployment technology is retrieved in an automated manner. This is achieved by requesting data from the API of the respective deployment technology about a running application, for example, about its components, their types, and instance data like the current public IP address of a VM component. The retrieval process heavily differs

between deployment technologies and requires deep technical knowledge and understanding of the respective technology. In the second step, the retrieved technology-specific instance information is used to derive a normalized instance model, i. e., EDMMi. The normalized model entails components instances, their properties and relations, as well as a global state for the application and each component instance respectively. To obtain a normalized model, deployment-technology-specific types are mapped to normative, reusable types provided by the EDMM types repository. Such normative types are e. g., *compute*, *software component*, or *database*. For instance, a MySQL database maps to the normative type database, while a VM maps to the normative type compute. To further decouple the instance model from its vendor-specific representation, and to make use of standardization, the normalized EDMMi is transformed to a standardized TOSCA instance model in the third step. In this process, the normative EDMM types are mapped to TOSCA types to retain their semantics. In the fourth step, the TOSCA instance model is analyzed for its component types, and, based on that, the instance model is enriched with management features using the Management Feature Enrichment approach of Harzenetter et al. [HBL+19], cf. Section 2.3.2. Next, the identified management features of each component instance are orchestrated to executable, holistic management workflows. In the last step, a user can execute the generated workflows to manage the running application, for example to test its availability.

The detailed description of the approach is structured as follows. Section 4.1 introduces the Essential Deployment Metamodel instance (EDMMi) and its definitions to provide a means to represent running applications in a normalized manner. In Section 4.2, a mapping between selected popular deployment technologies and EDMMi is described to transform deployment-technology-specific instance information to a normalized representation. In the following, a mapping between EDMMi and the TOSCA standard instance model is established [Org13a] in Section 4.3 to gain advantage from standardization. Section 4.4 describes how the Management Feature Enrichment approach is extended to enrich the previously obtained TOSCA instance model with management workflows. Concludingly, the concept enables the enrichment of running applications with additional management functionalities, regardless of their managing deployment technologies.

## 4.1 Essential Deployment Metamodel instance (EDMMi)

This section introduces the Essential Deployment Metamodel instance (EDMMi), which extends the EDMM by an instance model to represent running applications in a technology-agnostic manner. EDMM was used as a base since the deployment model of most deployment technologies can be mapped to it [WBF+19]. By normalizing the instance information of a running application, EDMMi provides a technology-independent, normalized representation of an application that is already deployed. Furthermore, EDMMi aims to ease the transformation of running applications between deployment technologies.

Since EDMMi is an extension of EDMM, it was derived by an analysis of the same 13 deployment technologies that were used by Wurster et al. [WBF+19] to derive EDMM, cf. Section 2.2. More precisely, EDMMi was derived mostly by investigating the official documentation, the APIs, and the provided entities of the instance model of the selected deployment technologies. Figure 4.2 depicts EDMMi which was obtained eventually. To

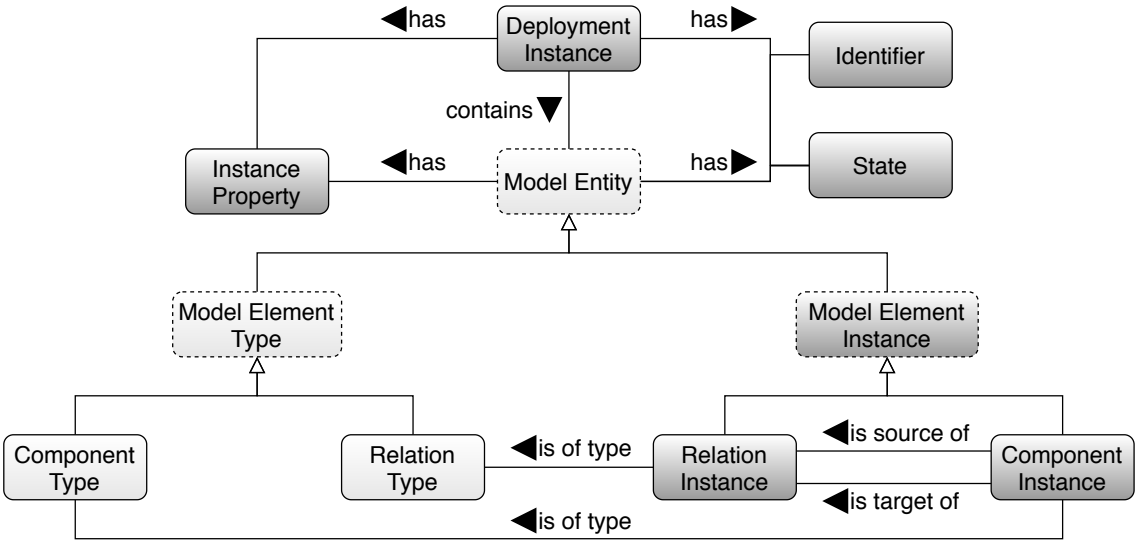


**Figure 4.2:** The Essential Deployment Metamodel instance (EDMMi).

visualize that EDMMi builds upon EDMM, new or changed entities compared to EDMM are highlighted in dark gray, while the unchanged entities are light gray. The graph-based nature, as well as the notation of the entities are inspired by the TOSCA standard [Org13b] and the Declarative Application Management Modeling and Notation (DMMN) [Bre16], analogous to EDMM [WBF+19]. To clearly distinguish from EDMM and to indicate that EDMMi is concerned with running applications, several entities were renamed. However, the following definitions are built upon the definition of EDMM [WBF+19].

The entities that form a running application are *component instances*, whose semantics are defined by component types [WBF+19]:

**Definition 1** "Component Instance": A *component instance* is an instantiated, and running physical, functional, or logical unit of an application instance.

**Definition 2** "Component Type": A *component type* is a reusable entity that specifies the semantics of a component instance that has this type assigned [WBF+19].

Component instances often depend on each other, e. g., a Tomcat web server requires that Java is running. To capture such interplay and dependencies between component instances, relation instances are used. Analogous to component instances, the semantics of relation instances are defined by relation types.

**Definition 3** "Relation Instance": A *relation instance* is an instantiated, and directed physical, functional, or logical dependency between exactly two component instances.

**Definition 4** "Relation Type": A *relation type* is a reusable entity that specifies the semantics of a relation instance that has this type assigned [WBF+19].

To make component and relation instances manageable, it is necessary to identify the configuration of these entities, for example, to connect to the port where a database is listening to requests. Such information is stored in *instance properties*.

**Definition 5** "Instance Property": An *instance property* describes the current state or configuration of a component instance, relation instance or deployment instance.

To bundle all component instances and relation instances that logically form an application, a *deployment instance* is used. A *deployment instance* describes an instance of a deployment model that was successfully deployed, i. e., a running application that was derived by executing its deployment model. It contains all model entities of a deployment model as instantiated entities, i. e., component instances and relation instances, as well as their types. Moreover, a deployment instance may have instance properties, e. g., to globally set the Java version that is used by all component instances.

**Definition 6** "Deployment Instance": A *deployment instance* describes an application which was successfully instantiated by executing its declarative deployment model.

Since components, relations, and deployment models can be instantiated arbitrary often, component instances and relation instances need an identifier to make them addressable.

**Definition 7** "Identifier": An *identifier* uniquely identifies a component instance, relation instance, or deployment instance within a deployment technology.

Besides that, to enable management of component instances, relation instances, and deployment instances, a means to describe their state is required. For example, it is not possible to successfully execute management workflows that replicate a component instance, if the component instance is not yet created.

**Definition 8** "State": A *state* describes the current lifecycle phase of a component instance, relation instance, or deployment instance.

The lifecycle phases are inspired by the lifecycle definition of the TOSCA standard [Org13b] specified in the TOSCA Primer [Org13a]: (i) *install*, (ii) *configure*, (iii) *start*, (iv) *stop*, and (v) *uninstall*. Hence, the values that describe the state model of a component instance, relation instance, or deployment instance are as follows: (i) *Creating*, (ii) *created*, (iii) *starting*, (iv) *started*, (v) *stopping*, (vi) *stopped*, (vii) *deleting*, and (viii) *deleted*. In addition to that, the states *updating* and *updated* are added to describe a component instance, relation instance, or deployment instance that was changed or is changing currently. To describe failure, the *error* state is introduced. Typically a deployment instance is considered to be successfully instantiated if none of its component instances and relation instances is in an error state.

## 4.2 Mapping from Deployment Technologies to EDMMi

Section 2.1.1 described the core principles of deployment technologies, and presented four deployment technologies in particular: (i) OpenStack Heat, (ii) Puppet, (iii) AWS CloudFormation, and (iv) Kubernetes. These technologies were selected respectively as representative for each category of deployment technologies as identified by Wurster et al. [WBF+19], i. e., (i) general-purpose deployment technologies, (ii) provider-specific deployment technologies, and (iii) platform-specific deployment technologies. Since the selected deployment technologies are solely orchestration technologies, Puppet was included additionally to cover configuration management tools. To prove the suitability of EDMMi, this section explains how the instance models of the chosen deployment technologies (i) OpenStack Heat, (ii) Puppet, (iii) AWS CloudFormation, and (iv) Kubernetes map to it. The mapping is necessary for the approach of this work to transform running applications and its technology-specific instance data to a normalized representation. For each of the aforementioned deployment technologies, the following sections will describe how the instance data of the respective deployment technology is retrieved, structured, and how the instance data is mapped semantically to EDMMi.

### 4.2.1 OpenStack Heat

The instance information of running applications deployed with OpenStack Heat can be retrieved using its built-in API. To retrieve the instance model of a particular application, its identifier within OpenStack is required. By mapping the instance data to EDMMi as visualized in Figure 4.3, a normalized EDMMi representation of a running application deployed with OpenStack Heat is derived. The bold black text describes the entities of OpenStack Heat, while the white text indicates to which entity the respective OpenStack heat entity maps in EDMMi, cf. Section 4.1. The topmost entity in OpenStack Heat is called a *stack*, which bundles all information of an application. Hence, a stack is mapped to a deployment instance in EDMMi. Since a stack is formed by its resources, i. e., logical, physical, or functional units, stack resources map to EDMMi component instances. The semantics of a stack resource is defined by a resource type. Since a stack resource maps to a component instance, its resource type maps to a component type in EDMMi. However, to derive a normalized EDMMi, deployment-technology-specific resource types, such as *Nova::Server*, must be mapped to normative EDMM component types. To enable this, it is necessary to carefully identify the semantics of each resource type available for OpenStack Heat resources. An excerpt of the mapping between OpenStack Heat resource types [Ope20d] and EDMM normative types is shown in Table 4.1. For instance, a *Nova::Server* in OpenStack Heat is mapped to the normative type compute.

To express a dependency between two resources, OpenStack Heat provides a *required by* attribute, which specifies that a resource is required by another resource. Hence, a dependency maps to a relation instance in EDMMi, while the relation type is represented by the *required by* type in OpenStack Heat. Besides specifying dependencies, the configuration of resources and stacks is described with key-value pairs, e. g., to define the OS of a Nova::Server instance. Therefore, such a key-value pair maps to an instance property in EDMMi. The availability of key-value pairs is bound to resource types, i. e., the attributes

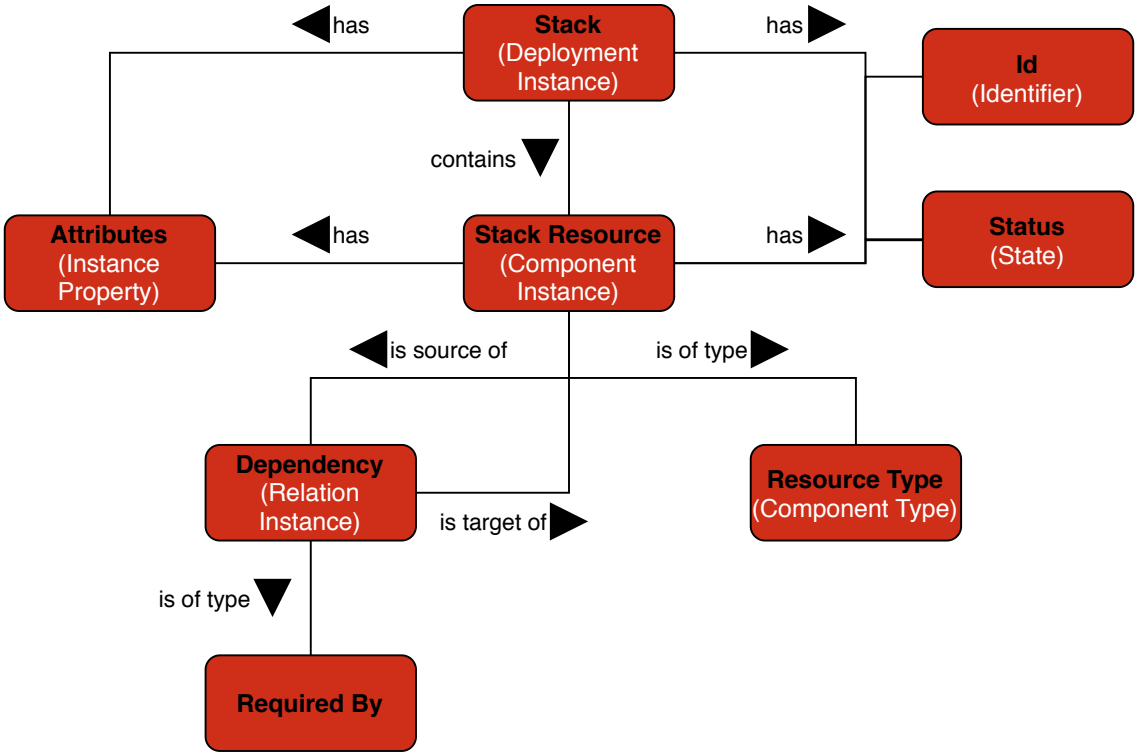


**Figure 4.3:** Mapping of the OpenStack Heat instance metamodel to EDMMi.

| OpenStack Heat resource type | EDMM normative type |
|---|---|
| Heat::SoftwareComponent | Software_Component |
| Nova::Server | Compute |
| Octavia::LoadBalancer | Compute |
| Trove::Instance | DBaaS |
| ... | ... |

**Table 4.1:** Excerpt of the mapping of OpenStack Heat resource types to EDMM normative types.

to be configured differs between, for example, a Nova::Server and a Trove::Instance. Thus, it is also necessary to map the keys of the attribute to a respective EDMM property key when mapping resource types to EDMM normative types. For instance, a resource of type Nova::Server specifies its OS in an attribute with the key *image*. When the resource type Nova Server is mapped to the EDMM normative type compute, the *image* attribute must be semantically mapped to an instance property of the compute type to retain the configuration of the resource. Since the compute type stores the information about its OS in the *os_family* property, the image key of an Nova::Server attribute is mapped to it.

Similar to deployment instances and component instances, stacks and its resources have a unique identifier to properly identify a specific running deployment. Also, a stack and its resources respectively feature a status attribute that describes their current state.

### 4.2.2 Puppet

Since Puppet follows a master-agent architecture, required instance information of an application deployed and configured with Puppet must be requested from *PuppetDB*. PuppetDB is a database running on a Puppet master that exposes an API to retrieve data about all managed agents and their configuration. As Puppet does not employ a holistic deployment model that explicitly specifies all of its components, it is assumed that all agents managed by one Puppet master form an application. This also includes software running on Puppet agents, e. g., a Tomcat web server or a MySQL database. Thus, to map an application deployed and managed with Puppet, the PuppetDB of the managing Puppet master is queried for its instance information, i. e., at least the IP address of the Puppet master that manages the application that must be transformed is required. The retrieved data is then mapped to EDMMi as depicted in Figure 4.4 to derive a normalized representation of such an application. Analogous to the mapping of OpenStack Heat to EDMMi, the bold black text describes the entities in Puppet, and the white text indicates the EDMMi entity it maps to. Since it is assumed that all agents managed by the same Puppet master form an application, the Puppet master maps to a deployment instance because it bundles all information about an application, its components and configurations. As an agent managed by a Puppet master is the basic building block of an application deployed and managed with puppet, it maps to a component instance in EDMMi. System variables of Puppet agents and their master are stored in so-called *facts*, for example the public IP address of the VM a Puppet agent is running on, or the OS of the VM. Hence, a fact in Puppet maps to an instance property in EDMMi. Further, to identify software running on a Puppet agent, reports are analyzed. Reports are generated by an agent when a software is installed or configured via its master, e. g., when Java is installed, or a MySQL database is configured. For example, a configuration at a Puppet master may declare that a Tomcat web server has to be installed on a certain Puppet agent. After the configuration is applied, the Puppet agent generates a report that specifies whether the execution was successfully, i. e., if the Tomcat web server was installed. Any resource identified as running on a Puppet agent in this process also maps to a component instance in EDMMi as they are software components. Furthermore, the aforementioned reports are used to identify the type of the detected resources. For instance, a report may be generated after installing a software, stating that the type of the configured resource is *software package*. Since a resource type defines the semantics of a resource to be managed, it maps to a component type in EDMMi. A Puppet agent itself does not explicitly specify a resource type in Puppet, however, since an OS is required to run it, the resource type of a Puppet agent is mapped to the EDMM normative type compute with a property indicating its OS family. Similar to the property mapping described in Section 4.2.1, the technology-specific fact keys must be mapped to the respective property keys when mapping Puppet agents to compute types. For instance, the fact that stores the public IP address is called *IPAddress* in Puppet, while the semantically equal instance property of the compute type is stored in the key *public_address*. Furthermore, a Puppet agent does not explicitly specify a global state, thus it is assumed that a Puppet agent is not in an error state if it is accessible Moreover, the deployment of a Puppet master does not define an identifier, but such identifier can be easily generated when retrieving the instance information from Puppet, e. g., by hashing the IP address of the managing Puppet master.

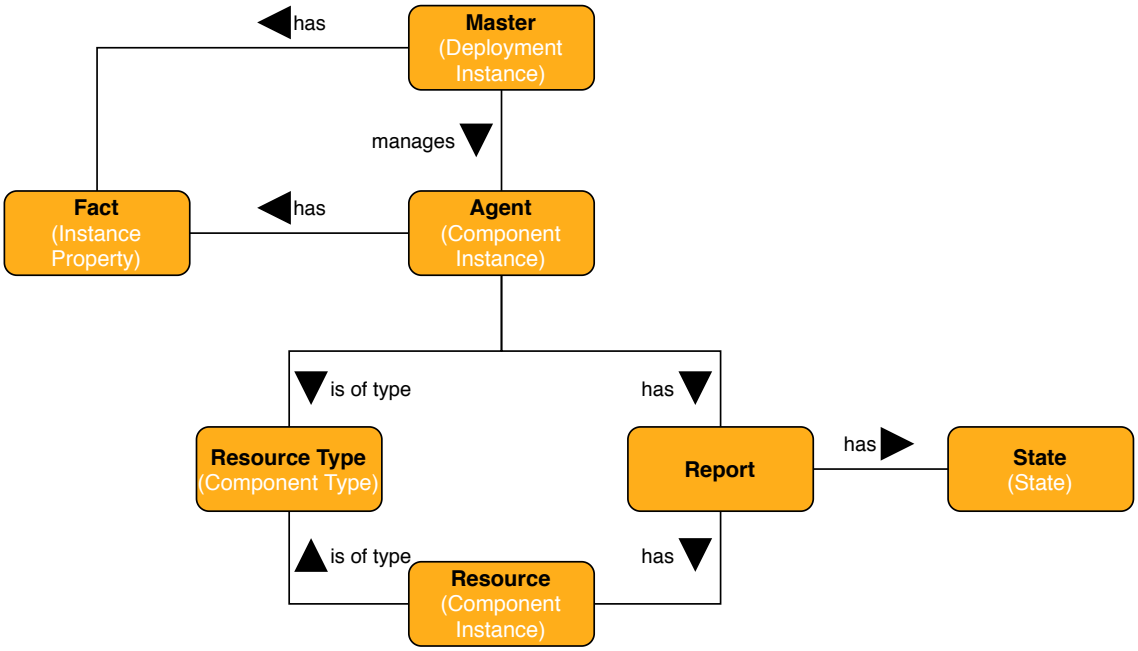


**Figure 4.4:** Mapping of the Puppet instance metamodel to EDMMi.

### 4.2.3 Amazon Web Services (AWS) CloudFormation

AWS CloudFormation provides various ways to retrieve instance information of deployed applications, e. g., its built-in API or the web console. To retrieve the instance information of a specific application in an automated manner, it is required to provide the identifier of the respective stack within AWS CloudFormation. By mapping the retrieved instance data to EDMMi as depicted in Figure 4.5, a normalized representation of an application deployed with AWS CloudFormation is derived. As before, the bold black text defines the entities of AWS CloudFormation, while the white text indicates the mapping to an EDMMi entity. A running application deployed with AWS CloudFormation is called *stack*. A stack bundles all information about such an application, e. g., its components, properties, and state. Hence, a stack maps to a deployment instance in EDMMi. As illustrated in Figure 4.5, such a stack has parameters that specifies information about it and its resources. Thus, a parameter maps to an instance property in EDMMi. The resources of a stack, e. g., a VM, or a database, are called *stack resources*. Since a stack resource is the main building block of an application in AWS CloudFormation, it maps to a component instance. The semantics of a stack resource is defined by its resource type, which maps to a component type in EDMMi. To derive a normalized EDMMi, the specific AWS CloudFormation resource types, such as DynamoDB, or EC2 instance must be mapped to normative EDMM types. Due to the plethora of resource types in AWS CloudFormation, only an excerpt of such a mapping is provided in Table 4.2. For instance, since an EC2 instance is a server, it maps to the normative type compute, while the Database-as-a-Service (DBaaS) offering RDS maps to the normative component type DBaaS. To describe dependencies between such resources in AWS CloudFormation, the *Depends On* attribute is used. Hence, a dependency maps to a relation instance in EDMMi, and its relation type is *Depends On*.

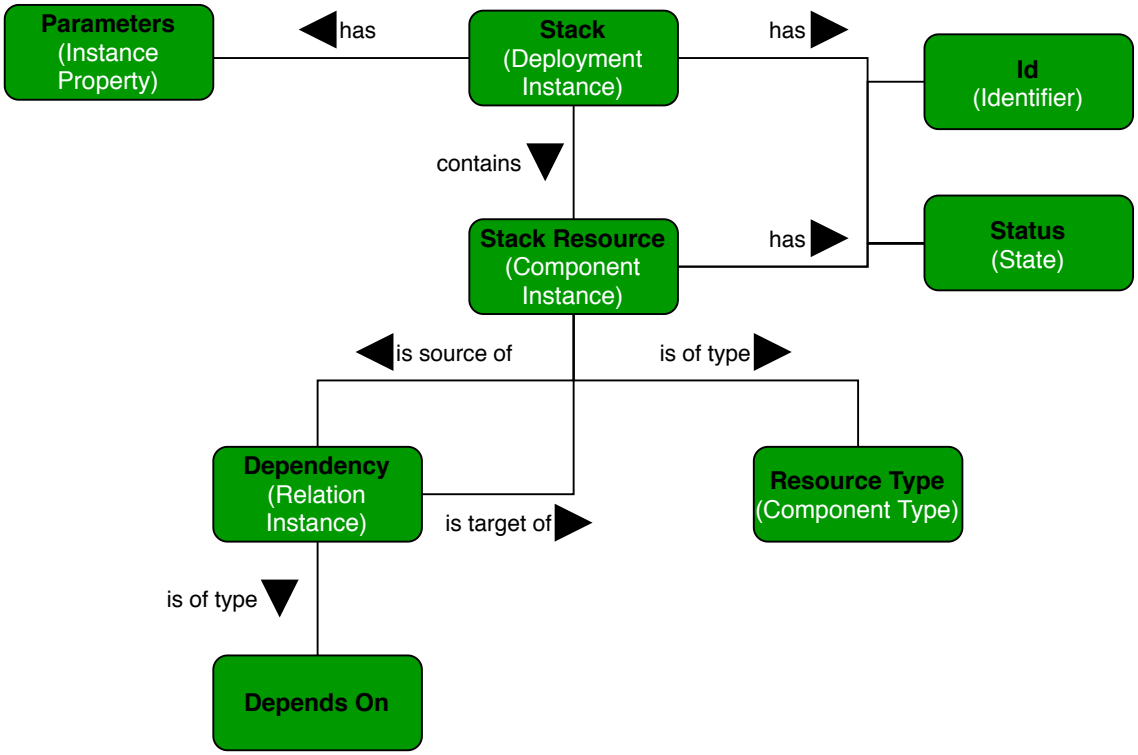


**Figure 4.5:** Mapping of the AWS CloudFormation instance metamodel to EDMMi.

| AWS CloudFormation resource type | EDMM normative type |
|---|---|
| EC2::Instance | Compute |
| Lambda::Function | PaaS |
| RDS::DBInstance | DBaaS |
| Redshift::Cluster | DBMS |
| ... | ... |

**Table 4.2:** Excerpt of mapping of AWS CloudFormation resource types to EDMM normative types.

Moreover, AWS CloudFormation provides a unique identifier for both the stack and each of its resources to make it addressable. Thus, this maps to an identifier in EDMMi. Furthermore, a stack and each of its resources feature a status attribute to capture the current lifecycle phase, e. g., to determine if a EC2 server is up and running. Such a status attribute maps to the state attribute in EDMMi.

### 4.2.4 Kubernetes

Similar to Puppet, Kubernetes follows a master-agent architecture. Hence, to retrieve instance information of an application deployed with Kubernetes, the respective data must be requested from the HTTP API exposed by the Kubernetes master's control plane [Kub20b; Kub20d]. To retrieve the instance information of a specific application

deployed with Kubernetes, its unique identifier is required. Then, to derive a normalized representation, the retrieved instance information is mapped to EDMMi as illustrated in Figure 4.6. Again, the white text indicates to which EDMMi entity the Kubernetes entity in the black text maps to. Similar to EDMMi, a running application is called a deployment in Kubernetes, and hence, maps to a deployment instance. A deployment contains one or more pods, which represent the smallest deployable unit of an application in Kubernetes [Kub20e]. A pod itself contains one or more containers, which map semantically to component instances in EDMMi. Since the semantics of a container is defined by its image, an image maps to a component type. For instance, an image may be a MySQL database, a Java installation, or a Tomcat web server. To derive a normalized EDMMi representation of an application deployed with Kubernetes, it is required to map the images to EDMM normative types. For example, the Tomcat web server image maps to the normative type *Web_Server*. To describe runtime information or the current configuration of a container, pod, and deployment, Kubernetes employs properties, for example to describe the IP address of a pod. Therefore, a property in Kubernetes maps to an instance property in EDMMi. As described before, it is necessary to map the deployment-technology-specific property keys when mapping properties from Kubernetes to EDMMi to retain their values and semantics. For instance, a pod exposes a property *podIP*, which maps to the *public_address* key of the EDMM normative type compute. Despite properties that describe the configuration of containers, pods and deployments, Kubernetes features a status attribute that describe the current lifecycle phase, for example to determine if all containers in a pod were started successfully. Since containers in a pod are tightly coupled, Kubernetes only offers a status attribute for a pod, but not for each of its containers individually [Kub20e]. Hence, the status attribute maps to the state entity in EDMMi. Moreover, a deployment and a pod feature a unique identifier to make them addressable, which maps to an identifier in EDMMi.

## 4.3 Mapping from EDMMi to TOSCA

In Section 4.2, mappings between the instance model of different deployment technologies and EDMMi were provided to show how applications deployed with such technologies can be represented in a technology-agnostic, normalized manner. To further decouple the representation of such applications from vendors and deployment technologies, this section presents a mapping from EDMMi to the standardized TOSCA instance model [Org13a; Org20]. This entails the advantage of freedom of choice between providers conforming with the TOSCA standard, e. g., OpenTOSCA [Bre+16], ALIEN 4 Cloud [Ali20], Cloudify [Ltd20], and Tosker [BRS18] [FS85; WBF+19]. Moreover, the mapping to TOSCA allows the utilization of the Management Feature Enrichment approach [HBL+19] which was implemented prototypically in OpenTOSCA.

The mapping between EDMMi and the TOSCA instance model is depicted in Figure 4.7. EDMMi entities are in bold black text, while the respective TOSCA entity is in white text. The topmost entity in EDMMi is a deployment instance, which contains all entities that define a running application. Analogous to this, a *service template instance* in TOSCA is an instantiation of a service template which contains all TOSCA definitions, files, and entities that make up an application. Hence, a deployment instance is mapped to a service

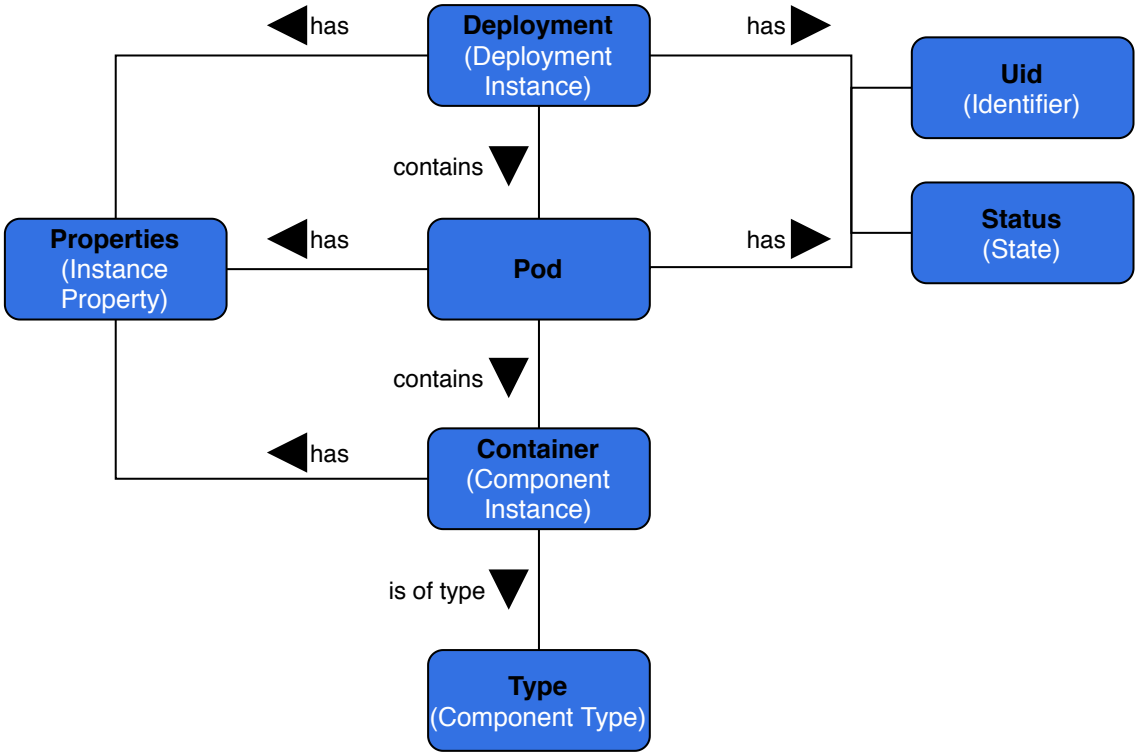


**Figure 4.6:** Mapping of the Kubernetes instance metamodel to EDMMi.

template instance. The building blocks of a deployment instance are component instances, which represent running application components, similar to node template instances in TOSCA. The interplay between two component instances is specified by a relation instance, which maps to a relationship template instance since it is semantically equivalent for node template instances in TOSCA. Since component instances map to node template instances, and relation instances to relationship template instances, the mapping of their semantic-defining types is trivial, i. e., component types in EDMMi map to node types, and relation types to relationship types. The configuration of a component instance, relation instance, and deployment instance is specified via instance properties, which map to instance properties in TOSCA. Besides its configuration, a component instance, relation instance, and deployment instance respectively features a state attribute. This is equivalent to the state attribute of a template instance in TOSCA which describes its current lifecycle phase [Org13a]. Moreover, the identifier of a deployment instance, component instance, and relation instance is provided by a *TOSCA runtime instance id* that uniquely identifies a template instance within a TOSCA runtime.

Since EDMMi extends EDMM, which is inspired by the TOSCA standard, the mapping between EDMMi and TOSCA is trivial. By utilizing this, it is possible to transform any running application that can be represented in EDMMi to a TOSCA instance model.

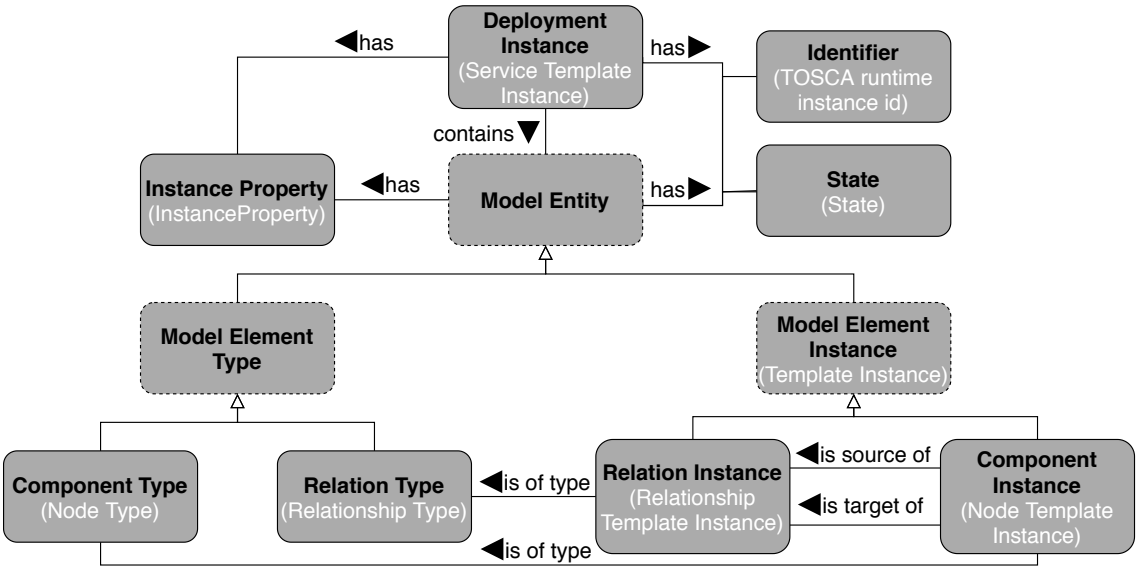


**Figure 4.7:** Mapping of EDMMi to the TOSCA instance metamodel.

## 4.4 Applying Management Feature Enrichment

In the first step of the approach, i. e., Section 4.2, it was shown how the instance model of different deployment technologies can be transformed to a normalized EDMMi representation. Subsequently, Section 4.3 provided a mapping between EDMMi and TOSCA to show how applications represented in EDMMi can be transformed to the standardized TOSCA instance model. Based on the derived TOSCA instance model, this section describes how the existing Management Feature Enrichment approach [HBL+19] is extended to enrich a running application with additional management functionalities, regardless of its managing deployment technology. However, the Management Feature Enrichment approach [HBL+19] currently only works on deployment models, i. e., applications which have not been deployed yet. Therefore, it is not possible to apply the Management Feature Enrichment approach [HBL+19] to our concept without further modification, since the approach presented in this work retrieves instance information of an application, but not its deployment model. Furthermore, a running application may differ from the model that was used to deploy it. Thus, it is required to modify the Management Feature Enrichment approach [HBL+19] to be able to enrich *running* applications with management functionalities. To do so, the fact that a service template instance is a derivation of a service template is leveraged [Org20]. More precisely, a service template instance extends a service template with instance information like the current state, or the current values of properties like the IP address of a VM. Thus, a service template instance also comprises information about the node types of the contained node template instances [Org20]. These node types of the node template instances are then used to identify possible management features as described in Section 2.3.2. However, to be able to properly identify management features based on node types, it is required to determine the node types as exact as possible when transforming from (i) a deployment technology to EDMMi, and from (ii) EDMMi to a TOSCA instance model. Section 4.2 already described how deployment-technology-specific types map to normative EDMM component types. Now, it is required to map

| EDMM normative type | TOSCA normative type |
|---|---|
| Compute | Compute |
| DBMS | DBMS |
| DBaaS | DBMS |
| Database | Database |
| PaaS | ContainerRuntime |
| Platform | ContainerRuntime |
| SaaS | SoftwareComponent |
| Software_Component | SoftwareComponent |
| Web_Application | WebApplication |
| Web_Server | WebServer |

**Table 4.3:** Mapping between EDMM and TOSCA normative types [Org13a].

the normative EDMM component types to TOSCA node types. Similar to EDMMi and EDMM, TOSCA defined a set of normative types in the TOSCA Primer [Org13a]. The mapping between EDMM normative types and TOSCA normative types is depicted in Table 4.3. Due to the similarity between EDMM and TOSCA, the mapping is trivial. However, it is difficult to enable proper management based on normative node types since the implementation of management operations heavily differs, even between semantically similar components. For example, it is hardly possible to define a management operation that performs a backup on all components of type compute, as the operation may depend on the underlying OS. Hence, node types need to be determined as exact as possible when transforming from EDMMi to TOSCA i. e., as close as achievable to the real-world representation. For instance, consider the exemplary application shown in Figure 2.1: When transforming this application to EDMMi, the Ubuntu VM is mapped to a component instance of type compute for a normalized, technology-agnostic representation. To be able to identify the type of a component as exact as possible when transforming from EDMMi to TOSCA, an additional property that states the deployment-technology-specific type is stored in the instance properties of a component instance in the transformation step from a deployment technology to EDMMi. For instance, when transforming the Ubuntu VM to an EDMMi component instance, its OSăis stored in the instance properties of the transformed component instance. Now, when transforming from EDMMi to TOSCA, the process to determine the node type of a node template instance is two-fold: (i) First, the normative EDMM component type of the component instance to be transformed to a node template instance is mapped to a TOSCA normative type as shown in Table 4.3. (ii) Next, to properly determine applicable management operations based on the node type, the normative TOSCA node type is refined using the property that records the original type that was stored when transforming from a deployment technology to EDMMi. To refine a node type, the TOSCA type hierarchy as shown in Figure 2.8 is leveraged. More precisely, an existing repository that defines TOSCA types that derive from normative TOSCA types is utilized. For instance, such a repository may contain a Ubuntu VM node type that specifies the normative TOSCA type compute in its *DerivedFrom* element. Now, based on the normative node type of a node template instance that was derived by transforming from EDMMi to TOSCA, the repository is searched for a matching type that derives from

the normative type using the additional instance property that stores its original type. For example, a Ubuntu VM deployed with OpenStack Heat is transformed to a component instance of type *compute* with the key-value property *original_type: Ubuntu VM*, which is then used to search such a TOSCA type hierarchy for the proper node type of a node template instance when transforming from EDMMi to TOSCA. If no matching type is found in the refinement process, its normative node type is used in the following process. Further, when mapping a type from EDMMi to TOSCA, it is also necessary to map its property keys accordingly to retain its semantic.

Based on the identified TOSCA types when applying the refinement approach while transforming from EDMMi to the TOSCA instance model, the Management Feature Enrichment approach [HBL+19] is applied to generate executable management workflows, cf. Section 2.3.2. In this process, the workflow generation distinguishes between (i) state-preserving, and (ii) state-changing management operations. State-preserving operations do not change the state of an application but only interact with it, e. g., by testing if a component is up and running, or by backing up a database. On the contrary, state-changing operations modify an application, for example, by replicating a component or changing its configuration. While state-preserving management operations can be executed straightforwardly, the execution of state-changing operations requires further processing. This is required since deployment technologies may monitor if the application state diverges from its deployment model, and thereupon perform steps to restore the application to the state declared in the model. Thus, it is necessary to notify or update the managing deployment technology when executing state-changing workflows to avoid interference. To enable this, the normalized, and standardized instance model needs to be annotated with the deployment technology that was used to deploy the transformed application. Further, state-changing management operations are annotated with the deployment technologies they support. A state-changing management operation *supports* a deployment technology if it propagates the changed state to the technology via its API to avoid interference. Then, when applying the Management Feature Enrichment, available state-changing management operations are filtered to match the deployment technology that manages the application.

Concludingly, this concept enables the enrichment of a running application with additional management features, regardless of the technology that was originally used to deploy it. Since the generated workflows are standards-based, it is possible to execute them using any engine that conforms with the respective standard, e. g., by using Apache ODE to execute BPEL workflows. Further, the approach enables the management of multiple running applications deployed with different deployment technologies in a single place, since a unifying standardized representation is derived.

# 5 Implementation

To prove the feasibility of the presented approach, a prototypical implementation based on the open-source OpenTOSCA ecosystem [Bre+16] and the EDMM transformation framework [WBB+20] is provided. This chapter explains the implementation step by step and provides insight on the integration into the existing workflows and ecosystems as illustrated in Figure 5.1. New or adapted components are highlighted in dark gray, while existing components are visualized in light gray. The EDMM Transformation Framework was extended by plugins that implement logic to retrieve information about running applications deployed with different deployment technologies. To prove the feasibility of the concept, plugins for Puppet, Kubernetes, OpenStack Heat, and AWS CloudFormation were implemented. As illustrated in Figure 5.1, each plugin implements methods to (i) retrieve instance information of a running application, and (ii) transform this retrieved

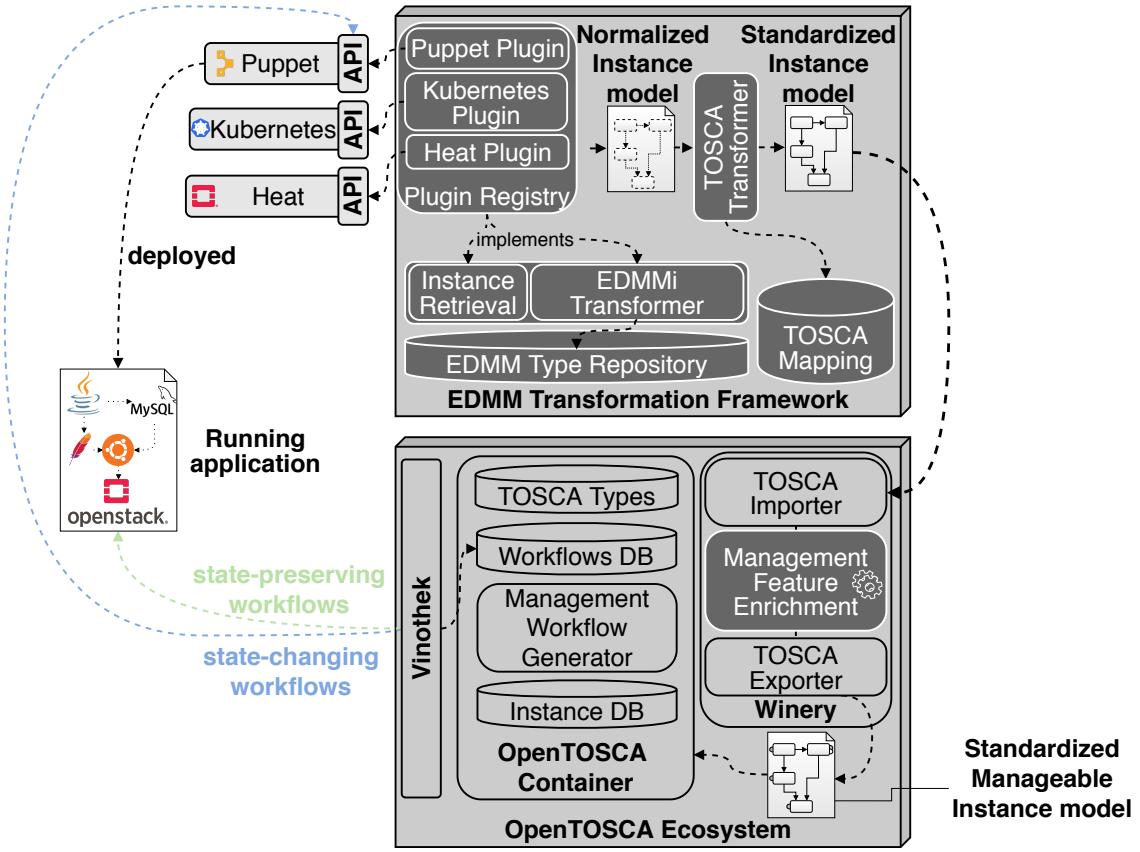


**Figure 5.1:** Architecture of the extended EDMM transformation framework and the OpenTOSCA ecosystem.

information to a normalized EDMMi representation The retrieval of instance information is performed via API request to the respective technology that was used to deploy the application. Then, the deployment-technology-specific instance information is transformed to an EDMMi representation in the *EDMMi Transformer*. The EDMM type repository is used in this step to map technology-specific types to normalized EDMM types, cf. Section 4.2. Therefore, this steps emits a normalized instance model according to the EDMMi specification. Next, the *TOSCA Transformer* implements logic to transform the derived EDMMi to a TOSCA instance model with support of the *TOSCA Mapping* Database (DB), which contains the mapping between EDMM and TOSCA types and property keys as described in Section 4.3. In this step, a standardized TOSCA instance model is derived. Moreover, the TOSCA Transformer interacts with the Winery API of the OpenTOSCA ecosystem to create a service template of the TOSCA instance model, and triggers the Management Feature Enrichment which is implemented in Winery with support of the *TOSCA Type Repository*. As a result, a standardized TOSCA instance model with additional management functionality is obtained. Then, this TOSCA instance model is imported into the OpenTOSCA Container, and registered as running application with the respective instance information that was retrieved. While importing the TOSCA instance model into the OpenTOSCA Container, the *Management Workflow Generator* generates holistic management workflows from the enriched operations, which are stored in the *Workflows DB*. Via Vinothek, the OpenTOSCA management User Interface (UI), a user is then able to trigger the generated management workflows on the running application.

In the following, every step of the implementation will be explained in depth. To retrieve the instance information of a running application deployed with a deployment technology, instance information retrieval plugins were implemented within the EDMM transformation framework. Such plugins traverse six phases in their execution lifecycle. To ease the implementation and extension with further instance information plugins, an interface for their implementation was specified as illustrated in Listing 5.1. The interface was created based on the *Template Method Pattern* [GHJV93] that defines the skeleton of each lifecycle phase. As illustrated, the six lifecycle phases are ordered as follows: (i) *Preparation Phase*, (ii) *Instance Information Retrieval Phase*, (iii) *EDMMi Transformation Phase*, (iv) *TOSCA Transformation Phase*, (v) *YAML Creation Phase*, and (vi) *Cleanup Phase*. The (i) Preparation Phase contains any steps required to retrieve instance information of an application deployed with a deployment technology, e. g., the authentication with the respective API. Next, the instance information of a running application deployed with a deployment technology is retrieved in the (ii) Instance Information Retrieval Phase via API requests. In the (iii) EDMMi Transformation Phase, the retrieved application is transformed to a normalized EDMMi representation, which is used in the (iv) TOSCA Transformation Phase to derive a standardized TOSCA instance model thereof. Further, the TOSCA Transformation Phase enriches the derived standardized TOSCA instance model with executable management workflows using the Management Feature Enrichment approach [HBL+19] which is implemented in the OpenTOSCA ecosystem. Subsequently, a YAML file is created that captures the derived EDMMi model in the (v) YAML Creation Phase. Eventually, the (vi) Cleanup Phase is concerned with measures to be executed before the plugin terminates, e. g., the removal of temporary files, or the release of a Secure Shell (SSH) connection. This phase is optional though, as it is not required by all plugins. Subsequently, an explanation of each phase is following.

**Listing 5.1** Interface for the implementation of instance information retrieval plugins.

```
package io.github.edmm.core.plugin;

import io.github.edmm.core.plugin.support.InstanceLifecyclePhaseAccess;

public interface InstancePluginLifecycle {

    void prepare();

    void retrieveInstanceInformation();

    void transformToEDMMi();

    void transformToTOSCA();

    void createYAML();

    void cleanup();
}
```

#### Preparation Phase

To retrieve instance information of a running application deployed with a deployment technology, the required data must be requested from the respective deployment technology's API. To do this, an authentication with the API is typically necessary. Thus, the *Preparation Phase* executes the *prepare()* method that implements logic related to anything required to start the retrieval of instance information. For example, the AWS CloudFormation plugin's *prepare()* method comprises logic to retrieve credentials for the authentication with the AWS API. Typically, an authentication with the API is required for all plugins, however its implementation differs between deployment technologies.

#### Instance Information Retrieval Phase

Once the preparation phase is finished, everything is set to request data from the API of a deployment technology to retrieve information about a running application. Thus, the *Instance Information Retrieval Phase* follows, which contains any logic concerned with the retrieval of instance data of a running application deployed with a deployment technology. Such information may include, for example, data about the components of an application, their state, and their properties. For instance, the Puppet plugin retrieves all agents that are currently managed by a certain Puppet master, as well as their configuration, e. g., a public IP address, but also additional software running on an agent, e. g., an Apache Tomcat web server as illustrated in Figure 5.1. The retrieval of such instance information is performed using the API of a deployment technology. Hence, the implementation of the *retrieveInstanceInformation()* method differs between plugins of different deployment technologies. Generally, this phase retrieves all information required to transform a running application to a normalized EDMMi representation.

#### EDMMi Transformation Phase

Once all instance information required for further processing is retrieved, it is transformed to a normalized EDMMi representation as defined in Section 4.1. The mapping presented in Section 4.2 between deployment-technology-specific data and EDMMi is utilized to implement the transformation logic for each plugin. Further, the mapping between deployment-technology-specific component types and normative EDMM component types is used to derive a normalized representation of a running application. Since the mapping to EDMMi is specific for each deployment technology, the implementation of the *transformToEDMMi()* method differs between deployment technologies.

#### TOSCA Transformation Phase

After the EDMMi Transformation Phase, a normalized EDMMi representation of a running application is derived. Next, the *TOSCA Transformation Phase* transforms the derived EDMMi to a standardized TOSCA instance model. The mapping between EDMMi and TOSCA described in Section 4.3 is utilized in this step. Since the previous phase emitted a normalized EDMMi representation of the application to be transformed, the *transformToTOSCA()* method implements the same logic, regardless of the plugin. Further, the TOSCA Transformation Phase comprises the steps to enrich the derived TOSCA instance model with additional management functionalities using the OpenTOSCA ecosystem. To enable this, the service template of the TOSCA instance model is imported into Winery using its API to apply the Management Feature Enrichment approach [HBL+19], which was implemented there prototypically. As described in Section 4.4, the Management Feature Enrichment approach distinguishes between (i) state-preserving, and (ii) state-changing management operations. While state-preserving operations can be executed straightforwardly, state-changing operations alter the state of an application, and thus, need to be handled differently. Since deployment technologies typically monitor the state of a deployed application, and restore its state if it diverges from the used deployment model, it is required to notify the deployment technology when performing state-changing operations. To achieve this, the imported service template is annotated with the deployment technology that was used to deploy the application. Further, the available state-changing management operations stored in the OpenTOSCA repository are annotated with the deployment technology they support. A state-changing operation *supports* a deployment technology if it propagates the changed state to the respective deployment technology to avoid interference. Now, when applying the Management Feature Enrichment approach [HBL+19], the OpenTOSCA repository is searched for possible enriching management operations based on the node types of the node templates of the service template as described in Section 2.3.2 and Section 4.4. To enrich a node template with *state-changing* management operations, the repository is filtered for operations that support the deployment technology the service template is annotated with. If an enriching management operation is identified, it is appended to the respective node template of the service template. As a result, a service template with enriched management operations is obtained. This service template is then imported via an API request into the OpenTOSCA Container, i. e., the TOSCA runtime of the OpenTOSCA ecosystem. During the import, the Management Workflow Generator generates holistic workflows of the individual management operations of the

node templates, e. g., by concatenating all management operations of a service template that are tagged as *backup* operation. The resulting management workflows are then stored in the Workflows DB in the OpenTOSCA Container. Since the application to be enriched is already running, the imported service template is registered as a running application, i. e., as service template instance. This is achieved by performing a further API request to the OpenTOSCA Container with the imported service template as target. In that request, the body of the request contains the instance data, i. e., the state of the node templates and service template, and their instance properties. At the end of this phase, a standardized TOSCA instance model is derived, enriched with additional, executable management workflows that can be triggered via the Vinothek [BBKL14d].

#### YAML Creation Phase

Subsequently, a YAML file is created that captures the EDMMi representation derived in the EDMMi Transformation Phase. This file holds the current state of the running application to be transformed and enriched in EDMMi and can be used, for example, for later re-import into the EDMM Transformation Framework or to gain an insight on the application. Listing 5.2 depicts an exemplary YAML file created in this phase that captures the EDMMi representation of a simple application deployed with Puppet. As visualized, the top level contains general information about the application, for example, the name, id, and creation date, followed by the component instances of the EDMMi model. Since the YAML file in Listing 5.2 is an excerpt, it only depicts two component instances, i. e., the VM the Puppet agent is running on, and a Tomcat hosted on the VM. As shown, a component instance comprises information about its state, type, and instance properties, like a SSH key pair for access, and the IP address of the VM. Also, the additional attribute that states the technology-specific type is stored, which is used for the type refinement when transforming from EDMMi to TOSCA.

#### Cleanup Phase

Concludingly, the last phase of a plugin's execution removes any track of its execution, e. g., temporarily created files. Further, it releases any established connections, e. g., to an API or a SSH connection to a VM. The *cleanup()* method is the last method executed before a plugin terminates. Typically, most plugins do not require this phase since connections are released automatically, hence its implementation is optional.

**Listing 5.2** Excerpt of an EDMMi YAML file as result of the YAML creation phase.

```
name: puppet.thesis.com
state: CREATED
id: '1321657185'
createdAt: '1590669238'
instanceProperties: null
componentInstances:
  - id: '1000217263'
    name: puppetagent
    type: Compute
    state: CREATED
    instanceProperties:
    - key: VMIP
      type: String
      instanceValue: ...
    - key: VMPrivateKey
      type: String
      instanceValue:...
    - key: VMPublicKey
      type: String
      instanceValue: ...
    - key: original_type
      type: String
      instanceValue: Ubuntu18.04
  - id: '1000483338'
    name: tomcat
    type: WebServer
    state: CREATED
    instanceProperties:
    - ...
  - id: ...
```

# 6 Validation

In this chapter, the feasibility of the presented approach and its prototypical implementation is validated in a case study, cf. Chapter 4, and Chapter 5. Further, the proposed concept is critically discussed, including its challenges and limitations.

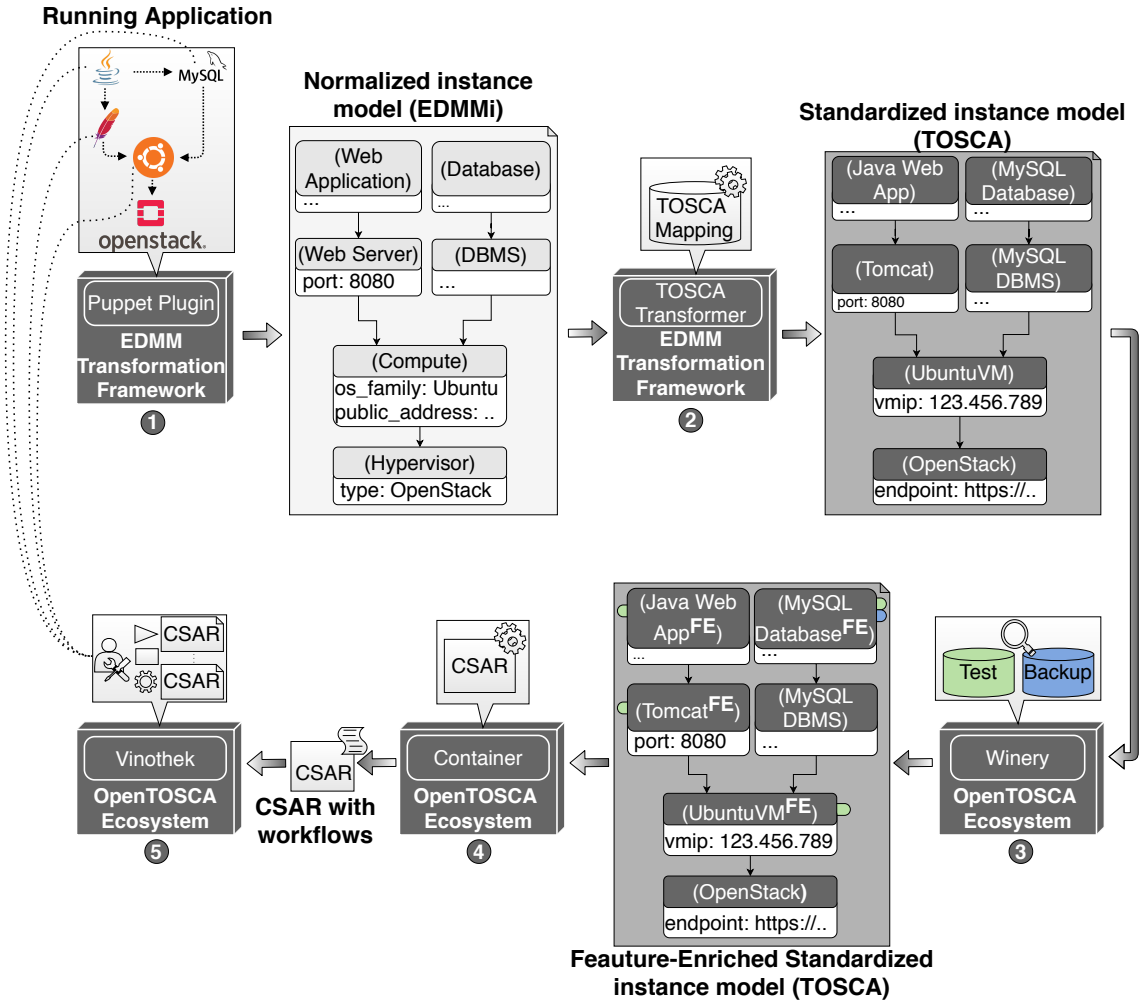


**Figure 6.1:** Validation scenario: Enriching a running application deployed with Puppet with management functionalities.

## 6.1 Case Study

Figure 6.1 depicts a validation scenario with a reference application that was deployed and configured with Puppet as visualized on the top left side, i. e., the technology-specific instance model. The application consists of a Java web application which is hosted on a Tomcat web server. Further, the Java web application connects to a MySQL database to store and retrieve data. Both the Tomcat web server and the database are running on a Ubuntu VM hosted on OpenStack. To enable the management of the described application in an automated manner, the prototypical implementation described in Chapter 5 was used to transform the technology-specific instance model to a standardized, feature-enriched, manageable instance model in TOSCA. Since Puppet follows a master-agent approach as outlined in Section 2.1.1, it is assumed that all agents managed by one Puppet master form one cohesive application. To start the transformation, the CLI of the extended EDMM Transformation Framework is called with an argument that specifies the technology used to deploy the application that must be transformed and enriched, i. e., in this case Puppet. Depending on the deployment technology, the framework expects further input arguments to specify the exact application to transform, e. g., the unique identifier of the application within its deployment technology. Further, parameters for the authentication with the API of the respective deployment technology must be provided typically. In the case of Puppet, following information about the Puppet master that manages the application must be provided: (i) Its public IP address, and (ii) a private SSH key of the master to access it. When started, the framework executes the Puppet plugin as illustrated in Figure 6.1. First, it retrieves all agents connected to the master, i. e., the compute nodes, and their properties like the public IP address, and the underlying OS. Further, the Puppet plugin examines the reports generated by the identified Puppet agents to determine if any additional software is running on it. Once all building blocks of the application and their configuration are identified, the plugin, which implements the mapping from Puppet to EDMMi as described in Section 4.2.2, transforms the technology-specific instance information to a normalized instance model, i. e., to EDMMi. To enable this, the concrete component types are mapped to normative types as depicted in Figure 6.1 within the braces. For instance, the VM that hosts the Tomcat web server and the database is mapped to the EDMM normative type *Compute*, while the Tomcat web server maps to the normative type *Web_Server*. With this, a technology-agnostic normalized instance model is derived that represents the illustrated application in EDMMi. As depicted in Figure 6.1, a close representation of the actual running application is obtained. Yet, the connection between the Java web application and the MySQL database is lost in this transformation step since Puppet does not hold such information in its instance data. By extending the approach presented in this work with the concepts discussed in Chapter 3, it would be possible to gather such information, e. g., by using network scanners [FAB+11].

In the second step, the framework transforms the derived EDMMi to a TOSCA instance model by executing the TOSCA transformer, which implements the mapping between EDMMi and TOSCA as specified in Section 4.3. First, the normative component types are transformed to normative TOSCA types using the mapping depicted in Table 4.3. In the following, the TOSCA transformer tries to refine the normative TOSCA types to concrete types using the TOSCA type repository of Winery to allow fine-grained management of these components in OpenTOSCA. In this process, the TOSCA transformer uses the

properties of the EDMMi components. For example, the Compute component holds information about its OS in the *os_family* property. This information is used to search the TOSCA repository for a matching subtype. In this case, the TOSCA transformer finds the node type *UbuntuVM* in the TOSCA repository. In this step, the framework also transforms the property keys respectively to retain their information and semantics. For instance, the key *public_address* of the Compute component is mapped to *vmip* when transforming to TOSCA. As a result, a standardized TOSCA instance model of the running application is derived as illustrated in Figure 6.1.

Using the standardized TOSCA instance model retrieved in the second step, the TOSCA transformer uses the Winery API to exploit the Management Feature Enrichment [HBL+19] implementation in Winery. In this process, the node types of the TOSCA instance model are used to identify enriching management operations, e. g., operations to test or backup a component. If a feature enrichment is found, the respective base node type is replaced by a feature-enriched node type which comprises the additional management operations, as well as the basic lifecycle operations of the original base node type. The resulting instance model of this step is visualized in Figure 6.1. In this example, all enriched management operations related to testing a component are highlighted in green, while backup related operations are illustrated in blue. For example, the generated $Ubuntu^{FE}$ node type offers a management operation that tests if an Ubuntu VM is up and running [HBL+19]. By importing this feature-enriched instance model in the OpenTOSCA Container, fully-executable workflows are generated that orchestrate the individual management operations based on annotations [HBL+19]. For instance, all test operations are orchestrated to one holistic test workflow. Consequently, a feature-enriched manageable instance model in OpenTOSCA is obtained. The enriched TOSCA model is then registered with its instance data and workflows as running application in the OpenTOSCA Container by the EDMM Transformation Framework. Using Vinothek, the management workflows can be invoked. Hence, the enrichment of an already deployed, running application with fully-executable management workflows in an automated manner is achieved.

## 6.2 Discussion

The described validation scenario showed how running applications deployed with technologies like Puppet can be enriched with management functionalities. Since the presented approach relies on deployment-technology-specific instance information, this reveals various challenges: (i) First, the presented approach depends on the instance information exposed by deployment technologies. However, the available instance information differs between deployment technologies. For example, connections and dependencies between application components may not be retrievable with Puppet as seen in the validation scenario. Further, some deployment technologies like AWS CloudFormation and OpenStack Heat only provide information about infrastructure components, e. g., servers, but not about software components. Therefore, the amount of retrievable instance information about applications deployed with such technologies is limited, i. e., the actual application architecture may deviate from the derived EDMMi representation. (ii) Furthermore, the mapping between technology-specific instance information and EDMMi requires deep technical knowledge

and understanding of the respective deployment technology. Especially when mapping properties and types from a deployment technology to normalized properties and types, immense technical expertise is necessary. Also, component type must be determined as exact as possible when transforming from the normalized EDMMi to TOSCA to enable proper management. If no exact mapping is possible, generic normative types are used which may impede management functionality.

(iii) Moreover, especially configuration management technologies may not explicitly express which nodes logically belong to one application. Thus, it may be necessary to meet assumptions about an application's structure and its components, e. g., that all agents managed by one Puppet master form one application.

(iv) As of now, the presented approach is limited to running applications deployed with deployment technologies, i. e., it is currently not possible to enrich manually deployed applications with management functionalities. However, it is possible to extend the presented concept using existing work from Binz [Bin15], Binz et al. [BBKL13a], [FAB+11], or [HBLE14] which use custom crawler, network scanner and service discovery to reliably retrieve information about running applications regardless of their deployment strategy.

(v) Another challenge of the presented concept brings the execution of state-changing management operations. Deployment technologies, especially configuration management tools like Puppet, constantly monitor managed agents and their configuration to check whether the state diverges from the declared model. If it differs, Puppet tries to restore the desired state based on the current configuration declaration at the Puppet master. Thus, if state-changing management workflows are invoked, it is required to ensure that the deployment technology that manages the application is updated about the changed state, such that it does not interfere with the executed workflow. More precisely, an interaction with the API of the respective deployment technology is necessary to match the internally stored state of the technology with the state obtained by executing a state-changing workflow. However, state-preserving management workflows, e. g., to test an application, can be executed straightforwardly since they do not alter the application and its state.

# 7 Conclusion and Future Work

This work presented a concept how running applications deployed with deployment technologies can be enriched with additional management functionality. To enable this, the approach introduced a normalized representation for running applications, as well as a transformation to a standardized representation using the instance model of the TOSCA standard. Further, the approach enables the management of applications deployed with different deployment technologies in a single place. Although the retrieval of instance information, as well as the semantic mapping between deployment-technology-specific and normative types and their properties revealed challenges, it was demonstrated that the information exposed by deployment technologies is sufficient to enable the automated management of such applications using the presented approach.

## Future Work

Currently, the presented approach is limited to applications deployed with deployment technologies, and further only to such technologies that provide access to instance information. Therefore, it is planned to extend the approach with the concepts presented by Binz et al. [BBKL13a; Bin15] to crawl instance information for applications that were either (i) deployed manually, or (ii) deployed with a technology that does not allow the access of instance information. Furthermore, improving the retrieval of instance information is planned such that the derived EDMMi model is closer to the real-world architecture, e. g., by providing a means to detect connections and dependencies between components more reliably. Since the presented approach enables the management of applications deployed with different deployment technologies in a single place, it may also be possible to generate high-level management workflows that coordinate the management of multiple applications, e. g., to backup all applications of an enterprise in a single workflow. Additionally, in future work the prototypical implementation may be extended to support popular deployment technologies that were not covered in this work, such as Chef, or Ansible. To enable this, it is also required to provide a conceptual mapping between the instance model of such deployment technologies and EDMMi similar to Section 4.2 in future work.

# Bibliography


[Ali20] Alien 4 Cloud. *Official Website*. 2020. URL: http://alien4cloud.github.io/ (cit. on p. 48).

[Ama20a] Amazon Web Services. *AWS CloudFormation*. 2020. URL: https://aws.amazon.com/cloudformation/ (cit. on pp. 15, 26).

[Ama20b] Amazon Web Services. *AWS CloudFormation Concepts*. 2020. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-concepts.html (cit. on p. 25).

[Ama20c] Amazon Web Services. *AWS resource and property types reference*. 2020. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html (cit. on p. 25).

[Ama20d] Amazon Web Services. *What is AWS CloudFormation?* 2020. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html (cit. on p. 25).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. "A Systematic Review of Cloud Modeling Languages". In: *ACM Comput. Surv.* 51.1 (Feb. 2018) (cit. on p. 15).

[BBH+13] 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 (ICSOC 2013)*. Springer, Dec. 2013, pp. 692–695 (cit. on pp. 29, 32).

[BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. "Integrated cloud application provisioning: Interconnecting service-centric and script-centric management technologies". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8185 LNCS (2013), pp. 130–148 (cit. on pp. 21, 33, 36).

[BBK+14a] 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 (IEEE IC2E 2014)*. IEEE Computer Society, Mar. 2014, pp. 87–96 (cit. on p. 20).

[BBK+14b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. "Context-Aware Cloud Application Management". In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*. SciTePress, Apr. 2014, pp. 499–509 (cit. on p. 37).

[BBKL13a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “Automated Discovery and Maintenance of Enterprise Topology Graphs”. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2013)*. IEEE, Dec. 2013, pp. 126–134 (cit. on pp. 35, 62, 63).

[BBKL13b] U. Breitenb, T. Binz, O. Kopp, F. Leymann. “Pattern-based Runtime Management of Composite Cloud Applications”. In: (2013), pp. 475–482 (cit. on pp. 33, 36, 37).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “TOSCA: Portable automated deployment and management of cloud applications”. In: *Advanced Web Services* 9781461475 (2014), pp. 527–549 (cit. on p. 19).

[BBKL14b] T. Binz, U. Breitenübcher, O. Kopp, F. Leymann. “Migration of enterprise applications to the cloud”. In: *it - Information Technology* 56.3 (2014), pp. 106–111 (cit. on pp. 36, 37).

[BBKL14c] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Automating Cloud Application Management Using Management Idioms”. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, May 2014, pp. 60–69 (cit. on p. 37).

[BBKL14d] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Vinothek - A Self-Service Portal for TOSCA”. In: *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*. CEUR-WS.org, Feb. 2014, pp. 69–72 (cit. on pp. 32, 57).

[BCS17] A. Brogi, P. Cifariello, J. Soldani. “DrACO: Discovering available cloud offerings”. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 269–279 (cit. on p. 36).

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. “Formalizing the Cloud through Enterprise Topology Graphs”. In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 742–749 (cit. on pp. 36, 37).

[Bin15] T. Binz. “Crawling von Enterprise Topologien zur automatisierten Migration von Anwendungen: eine Cloud-Perspektive”. Dissertation. University of Stuttgart, 2015 (cit. on pp. 35–37, 62, 63).

[Bre+16] U. Breitenbücher et al. “The OpenTOSCA Ecosystem - Concepts & Tools”. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016,* SciTePress, 2016, pp. 112–130 (cit. on pp. 17, 32, 48, 53).

[Bre16] U. Breitenbücher. “Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements”. Dissertation. University of Stuttgart, 2016 (cit. on pp. 37, 41).

[BRS18] A. Brogi, L. Rinaldi, J. Soldani. “TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications”. In: *Software: Practice and Experience* 48.11 (2018), pp. 2061–2079 (cit. on p. 48).

[Can20] Canonical Ltd. *Juju as a Service*. 2020. URL: https://jaas.ai/ (cit. on p. 26).

[Che20] Chef. *Chef*. 2020. URL: https://www.chef.io/ (cit. on pp. 15, 26).

[Clo19] Cloud Native Computing Foundation. *CNCF Survey 2019*. 2019. URL: https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf (cit. on p. 22).

[Doc20a] Docker Inc. *Docker*. 2020. URL: https://www.docker.com/ (cit. on pp. 15, 22).

[Doc20b] Docker Inc. *Overview of Docker Compose*. 2020. URL: https://docs.docker.com/compose/ (cit. on p. 26).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the 9th International Conferences on Pervasive Patterns and Applications* (2017), pp. 22–27 (cit. on pp. 15, 20, 21).

[EBLW17] C. Endres, U. Breitenbücher, F. Leymann, J. Wettinger. "Anything to Topology - A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts". In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal*. SCITEPRESS, Apr. 2017, pp. 180–190 (cit. on p. 36).

[FAB+11] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, I. Hanschke. "Automation Processes for Enterprise Architecture Management". In: *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*. 2011, pp. 340–349 (cit. on pp. 35, 60, 62).

[FS85] J. Farrell, G. Saloner. "Standardization, Compatibility, and Innovation". In: *The RAND Journal of Economics* 16.1 (1985), pp. 70–83 (cit. on p. 48).

[Gar10] Gartner. *Gartner Identifies the Top 10 Strategic Technologies for 2011*. 2010. URL: https://www.businesswire.com/news/home/20101019007069/en/Gartner-Identifies-Top-10-Strategic-Technologies-2011 (cit. on p. 19).

[GHJV93] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns: Abstraction and Reuse of Object-Oriented Design". In: *ECOOP' 93 --- Object-Oriented Programming*. Ed. by O. M. Nierstrasz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 406–431 (cit. on p. 54).

[Has20] HashiCorp. *Terraform*. 2020. URL: https://www.terraform.io/ (cit. on pp. 15, 26).

[HAW11] H. Herry, P. Anderson, G. Wickler. "Automated Planning for Configuration Changes". In: *Proceedings of the 25th International Conference on Large Installation System Administration*. LISA11. Boston, MA: USENIX Association, 2011, p. 5 (cit. on pp. 15, 21).

[HBL+19] L. Harzenetter, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder, M. Wurster. "Automated Generation of Management Workflows for Applications Based on Deployment Models". In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. 2019, pp. 216–225 (cit. on pp. 16, 17, 19, 33, 34, 36, 40, 48, 50, 52, 54, 56, 61).

[HBLE14] H. Holm, M. Buschle, R. Lagerström, M. Ekstedt. "Automatic data collection for enterprise architecture models". In: *Software & Systems Modeling* 13.2 (2014), pp. 825–841 (cit. on pp. 35, 62).

[KBBL13] 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 (ICSOC 2013)*. Springer, Dec. 2013, pp. 700–704 (cit. on p. 32).

[KNK16] T. Kuroda, M. Nakanoya, A. Kitano. "The Configuration-Oriented Planning for Fully Declarative IT System Provisioning Automation". In: Noms (2016), pp. 808–811 (cit. on pp. 33, 36).

[Kub20a] Kubernetes. *Pods Documentation*. 2020. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/ (cit. on p. 26).

[Kub20b] Kubernetes). *Kubernetes Components*. 2020. URL: https://kubernetes.io/docs/concepts/overview/components/ (cit. on pp. 25, 47).

[Kub20c] Kubernetes). *Nodes*. 2020. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (cit. on pp. 25, 26).

[Kub20d] Kubernetes). *Overview*. 2020. URL: https://kubernetes.io/docs/concepts/overview/ (cit. on pp. 25, 47).

[Kub20e] Kubernetes). *Pods*. 2020. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (cit. on pp. 26, 48).

[Kub20f] Kubernetes). *What is Kubernetes?* 2020. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (cit. on p. 25).

[Ley09] F. Leymann. "Cloud Computing: The Next Revolution in IT". In: *Photogrammetric Week '09*. Wichmann Verlag, 2009, pp. 3–12 (cit. on p. 19).

[Ltd20] C. P. Ltd. *Cloudify*. 2020. URL: https://cloudify.co/ (cit. on pp. 26, 48).

[MDW+00] V. Machiraju, M. Dekhil, K. Wurster, P. K. Garg, M. L. Griss, J. Holland. "Towards Generic Application Auto-Discovery". In: *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*. IEEE, Apr. 2000, pp. 75–87 (cit. on p. 36).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011) (cit. on p. 19).

[Mic20] Microsoft. *Azure Resource Manager documentation*. 2020. URL: https://docs.microsoft.com/en-us/azure/azure-resource-manager/ (cit. on p. 26).

[MSK+18] P. Masek, M. Stusek, J. Krejci, K. Zeman, J. Pokorny, M. Kudlacek. "Unleashing Full Potential of Ansible Framework: University Labs Administration". In: *Proceedings of the 22nd Conference of Open Innovations Association FRUCT*. FRUCT22. Jyvaskyla, Finland: FRUCT Oy, 2018 (cit. on p. 21).

[Nor20] Northern.tech, Inc. *CFEngine*. 2020. URL: https://cfengine.com/ (cit. on p. 26).

[Obj11] Object Management Group. *Business Process Model and Notation (BPMN) Version 2.0*. 2011 (cit. on pp. 15, 30).

[Ope20a] OpenStack. *Heat*. 2020. URL: https://wiki.openstack.org/wiki/Heat (cit. on pp. 15, 22, 23, 26).

[Ope20b] OpenStack. *Heat Orchestration Template (HOT) specification*. 2020. URL: https://docs.openstack.org/heat/latest/template_guide/hot_spec.html (cit. on p. 23).

[Ope20c] OpenStack. *Heat Template Guide*. 2020. URL: https://docs.openstack.org/heat/rocky/template_guide/ (cit. on pp. 22, 23).

[Ope20d] OpenStack. *OpenStack Resource Types*. 2020. URL: https://docs.openstack.org/heat/latest/template_guide/openstack.html (cit. on pp. 23, 43).

[Ope20e] OpenStack. *The OpenStack project*. 2020. URL: https://www.openstack.org (cit. on p. 15).

[Opp03] D. Oppenheimer. "The importance of understanding distributed system configuration". In: *Proceedings of the 2003 Conference on Human Factors in Computer Systems Workshop. CHI 2003*. Apr. 2003 (cit. on pp. 15, 19, 36).

[Org07] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language Version 2.0*. 2007 (cit. on pp. 15, 30).

[Org13a] Organization for the Advancement of Structured Information Standards (OASIS). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. 2013 (cit. on pp. 31, 40, 42, 48, 49, 51).

[Org13b] Organization for the Advancement of Structured Information Standards (OASIS). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. 2013 (cit. on pp. 17, 28–30, 32, 41, 42).

[Org20] Organization for the Advancement of Structured Information Standards (OASIS). *TOSCA Simple Profile in YAML Version 1.3*. 2020 (cit. on pp. 17, 28, 31, 48, 50).

[Pup20a] Puppet. *Introduction to Puppet*. 2020. URL: https://puppet.com/docs/puppet/6.18/puppet_overview.html (cit. on pp. 23, 24).

[Pup20b] Puppet. *Puppet*. 2020. URL: https://puppet.com/ (cit. on pp. 15, 26).

[Pup20c] Puppet. *Resource Type Reference*. 2020. URL: https://puppet.com/docs/puppet/5.5/type.html (cit. on p. 24).

[Red20] Red Hat. *Ansible*. 2020. URL: https://www.ansible.com/ (cit. on p. 26).

[Sal20] SaltStack, Inc. *SaltStack*. 2020. URL: https://www.saltstack.com/ (cit. on p. 26).

[The20a] The Apache Software Foundation. *Apache Tomcat*. 2020. URL: http://tomcat.apache.org/ (cit. on p. 15).

[The20b] The Kubernetes Authors. *Kubernetes*. 2020. URL: https://kubernetes.io/ (cit. on p. 15).

[WBB+20] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “The EDMM Modeling and Transformation System”. In: *Service-Oriented Computing -- ICSOC 2019 Workshops*. Ed. by S. Yangui, A. Bouguettaya, X. Xue, N. Faci, W. Gaaloul, Q. Yu, Z. Zhou, N. Hernandez, E. Y. Nakagawa. Cham: Springer International Publishing, 2020, pp. 294–298 (cit. on pp. 15–17, 27, 28, 36, 53).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The essential deployment metamodel: a systematic review of deployment automation technologies”. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Aug. 2019), pp. 63–75 (cit. on pp. 15, 16, 20–22, 26, 27, 36, 40–43, 48).

All links were last followed on September 30, 2020.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature