SICS Software-Intensive Cyber-Physical Systems (2020) 35:63–75
https://doi.org/10.1007/s00450-019-00412-x

SPECIAL ISSUE PAPER

The essential deployment metamodel: a systematic review of deployment automation technologies

Michael Wurster[1] · Uwe Breitenbücher[1] · Michael Falkenthal[1] · Christoph Krieger[1] · Frank Leymann[1] · Karoline Saatkamp[1] · Jacopo Soldani[2]

Published online: 26 August 2019

Abstract
In recent years, a plethora of deployment technologies evolved, many following a declarative approach to automate the delivery of software components. Even if such technologies share the same purpose, they differ in features and supported mechanisms. Thus, it is difficult to compare and select deployment automation technologies as well as to migrate from one technology to another. Hence, we present a systematic review of declarative deployment technologies and introduce the essential deployment metamodel (EDMM) by extracting the essential parts that are supported by all these technologies. Thereby, the EDMM enables a common understanding of declarative deployment models by facilitating the comparison, selection, and migration of technologies. Moreover, it provides a technology-independent baseline for further deployment automation research.

Keywords Deployment · Infrastructure as Code · Configuration Management · Metamodel · Review

1 Introduction

With the advent of DevOps [24] as a software development paradigm, the gap between *development* and *operations* is attempted to be eliminated by revising organizational and cultural challenges. One integral aspect of DevOps is to enable an efficient collaboration by establishing deployment processes that are highly automated [23] as manual deployments of services consisting of multiple units is complex, hard to repeat, and error-prone [33]. Key concepts like *configuration management* [16] and *infrastructure as code* [28] enable a continuous and automated delivery of software components over the entire lifecycle, e.g., to install, start, stop, or terminate components. By describing components and infrastructure of an application in maintainable and reusable *deployment models*, a repeatable end-to-end deployment automation can be established. Such deployment models can be of *declarative* or *imperative* nature [19]: Declarative models express the desired state into which an application or parts thereof are transferred. In contrast, imperative models describe the deployment steps in a procedural manner. In industry and research, declarative deployment models are widely accepted as the most appropriate approach for application deployment and configuration management [22]. As a result, a plethora of different technologies have been developed following this approach such as Chef, Puppet, AWS CloudFormation, Terraform, and Kubernetes.

All such technologies aim at automating the deployment of applications, but they differ in supported features and mechanisms. For example, Terraform supports the deployment across multiple cloud providers and it is able to target

✉ Michael Wurster
michael.wurster@iaas.uni-stuttgart.de

Uwe Breitenbücher
breitenbuecher@iaas.uni-stuttgart.de

Michael Falkenthal
falkenthal@iaas.uni-stuttgart.de

Christoph Krieger
krieger@iaas.uni-stuttgart.de

Frank Leymann
leymann@iaas.uni-stuttgart.de

Karoline Saatkamp
saatkamp@iaas.uni-stuttgart.de

Jacopo Soldani
soldani@di.unipi.it

[1] Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany

[2] Department of Computer Science, University of Pisa, Pisa, Italy

different cloud offerings as a service (XaaS). Whereas, there are cloud provider-specific technologies, such as AWS CloudFormation, allowing the deployment only on Amazon's cloud services. Moreover, there are platform-specific technologies, such as Kubernetes, that support only specific *deployment bundles* (container images) or cloud service offerings (e.g., restricted to PaaS). In addition, most technologies use their own modeling language with its own syntax and expressiveness.

As a result, it is difficult to compare technologies by their capabilities as there exists currently no systematic comparison of deployment features and mechanisms. Further, as application systems are in constant change, it is challenging to choose an appropriate technology upfront. Moreover, in cases applications need to be migrated, it is also required to migrate the associated deployment models to the target environment's technology. This quickly gets cumbersome and requires knowledge about how to translate the features and mechanisms from one provider's technology, e.g., AWS, to another one, e.g., Azure. In addition, the various technologies currently in use impede systematic deployment automation research as the practical feasibility of new approaches is typically only evaluated using a certain technology. However, this makes it hard for researchers to understand if proposed approaches can be mapped to their technologies in use.

To tackle these issues, we introduce the Essential Deployment Metamodel (EDMM), which we obtained through a systematic analysis aimed at distilling the essential parts of declarative deployment technologies. We also show how the analyzed technologies comply semantically with the EDMM and how it can be mapped to native constructs of each technology. Thereby, the EDMM provides a common denominator of the features of the most important deployment technologies. This enables (1) a common understanding of declarative deployment technologies. Further, it (2) eases the selection of a deployment technology for one's own use case as the EDMM mapping describes how this can be achieved based on a technology-independent model. In cases where it is required to migrate from one provider to another, it also requires to migrate the associated deployment models. The EDMM (3) supports and eases such migration processes by knowing the essential elements of deployment technologies. On top of that, the EDMM facilitates an automated transformation into specific deployment technologies in the context of model-driven architecture (MDA) in order to decide which specific technology to use as late as possible. Finally, our results give researchers (4) the possibility to evaluate their concepts in a technology-agnostic manner by knowing to which technologies an approach can be applied to without disruptive adaptations. The review of technologies revealed that there are general-purpose (GP), provider-specific (ProvS), and platform-specific (PlatS) deployment technologies that enable the description of components, relations, and respective types as main deployment model entities.

Hereafter, Sects. 2 and 3 present our review framework and technology classification. Section 4 defines the EDMM, while Sect. 5 discuss its mapping to selected technologies. Finally, Sects. 6 and 7 discuss related work and draw some concluding remarks.

2 Review framework

This section presents the procedure taken to identify the EDMM. This was done in three phases: First, we identified a list of deployment technologies using well-known search engines in research and industry. Each result was individually considered and searched for presented or used deployment technologies. In the second phase, we ranked the technologies by the amount of search results in a search engine. Lastly, the analysis of the highest ranked technologies based on a reference scenario led us to the essential elements of deployment models. The complete data of the review are available online.[1]

2.1 Phase 1: identify technologies

In the first phase, we used ACM Digital Library, IEEE Xplore, and Google to identify deployment technologies. We excluded the term "deployment" as the hundreds of results include unrelated research topics, i.e., covering organizational and build processes regarding deployment. Therefore, we refined the search using known terms to target the identification of deployment automation technologies. In each search engine, the following query structure was used: Results must contain the phrase "infrastructure as code" or "configuration management" and must match the keyword "cloud computing". The term "cloud computing" is chosen because we regard the support of cloud services as an elementary characteristic of future-proof deployment technologies.

The exact search queries and result numbers are shown in Table 1. Each result was individually considered and searched for whether a new technology is presented, the technology is used in comparative works, or is used to support evaluating a work or study. In total, 56 technologies were identified. We focused our review on open-source or community-licensed technologies and, therefore, excluded eight technologies in this phase since there are only commercial or enterprise licenses available. Further, we excluded Vagrant [21] because it focuses on managing local development environments. In addition, we excluded AWS OpsWorks [4] because it is a managed service by AWS pro-

[1] http://tinyurl.com/y2azrq3r.

Table 1 Search details for identifying deployment technologies

Engine	Query	Records
ACM	("infrastructure as code" "configuration management") AND keywords.author.keyword: ("cloud computing")[a]	19
IEEE	(("infrastructure as code" OR "configuration management") AND "IEEE Terms":"cloud computing")[b]	71
Google	"configuration management" "infrastructure as code" "cloud computing"	72.600[c]

[a]Used "Any field" and "Matches any" operators
[b]"Command Search" with "Metadata Only" operator
[c]The first 100 results were considered

viding Chef or Puppet master nodes and does not provide its own deployment technology.

2.2 Phase 2: technology selection

To select the most popular deployment technologies, a Google search for each technology was performed. Since most technologies are industry driven and not backed by scientific papers, the amount of citations is not usable as ranking method. Therefore, the magnitude of search results in Google was used, which also includes discussions in blogs or newsgroups, in order to derive the relevance of a certain technology. Even though the result is not precise, we are able to derive a trend in currently used deployment technologies. We used the following search pattern for ranking: "<technology> 'configuration management' OR 'infrastructure as code'". We used the American version of Google for the search. The search was performed using Google's Chrome browser in incognito mode on February 14, 2019. We considered the first 13 technologies for further analysis as shown in Table 2. The complete ranking is available online[1].

2.3 Phase 3: technology analysis

We analyzed the features, mechanisms, and capabilities of each selected technology based on a reference scenario, which is depicted in Fig. 1. The authors investigated how it can be realized using the native constructs or extension mechanisms of each technology.

Table 2 Deployment technology ranking by popularity

#	Technology	Search hits
1	Puppet	1.630.000
2	Chef	1.150.000
3	Ansible	989.000
4	Kubernetes	708.000
5	OpenStack HEAT	458.000
6	Terraform	348.000
7	AWS CloudFormation	156.000
8	SaltStack	130.000
9	Juju	53.200
10	CFEngine	48.500
11	Azure Resource Manager	47.100
12	Docker Compose	41.300
13	Cloudify	23.100

2.3.1 Deployment features and mechanisms

For the analysis, several aspects were taken into account in order to find commonalities. We expect the deployment technologies to be open-source or community-licensed, capable to express single-, multi-, or hybrid-cloud deployments, and provide support for cloud-native application components (PaaS and FaaS). We derived the following *deployment features and mechanisms* for analyzing the automated deployment capabilities, these are: (i) supporting multiple cloud providers and platforms, (ii) targeting different cloud offerings (XaaS), (iii) providing capabilities to structure a deployment into logical parts, (iv) supporting the creation of custom, fine-granular, reusable entities, (v) allowing to specify desired application state, and (vi) allowing to hook into or influence the deployment lifecycle. These deployment automation features and mechanisms were analyzed based on a reference scenario presented in the next subsection.

2.3.2 Reference scenario

We envision a simplified multi-cloud order management application containing cloud-native as well as classical and legacy components in order to identify the derived deployment features and mechanisms for each technology. This reference scenario is intended to cover typical deployment requirements in modern application systems. It covers exemplary different cloud providers, regarding multi-cloud deployments and different cloud offerings as a service (XaaS). Even though the reference scenario does not reflect a complex real-life scenario, it supports the identification of required deployment capabilities to cover modern application deployments. The left hand side of the figure depicts

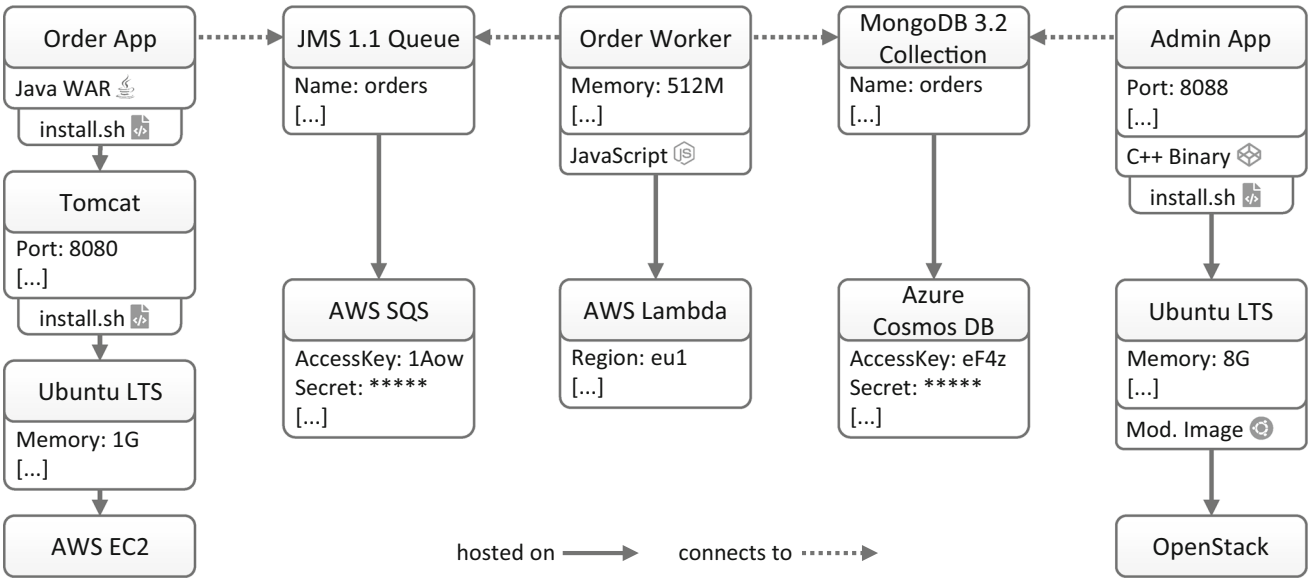


Fig. 1 Reference scenario: application topology containing cloud-native as well as on-premise components

a Java-based *Order* application deployed onto an Apache Tomcat server, which is depicted to be installed on a virtual machine (VM) provided by Amazon EC2. The order application is able to push new orders to a queue hosted and managed by Amazon's Simple Queue Service (SQS). Orders in this queue are processed by a *Order Worker* implemented as an ephemeral and stateless function using Amazon Lambda, Amazon's Function as a Service (FaaS) offering. In this case, each message put to the queue acts as event and triggers the worker function to process the respective order. The worker function updates respective values in a database, e.g., allocates the purchase order items for shipping. To cover a multi-cloud scenario, the orders are stored using a MongoDB Collection hosted and managed by Azure's Cosmos DB service offering. On the right hand side of the figure, a rather traditional, or non-cloud-native, application component is depicted. It is envisioned that a kind of administration application is used internally to track and manage the received orders by the *Order App*. In our scenario, this application is implemented using `C++`, requiring custom installation routines (cf. `install.sh` shell script), and is hosted on-premise using OpenStack. In summary, our reference scenario uses *application-specific components*, such as the *Order App* implementing some business logic. Further, it contains *middleware components*, such as the *Tomcat* server, and *computing components*, i.e., virtual machines. Lastly, it contains so-called *cloud service components*, such as *AWS EC2* and *Azure Cosmos DB*, which are hosting leaf nodes (considering the notion of a graph) and in full control of the respective cloud provider. Anyhow, there can be hosting leaf nodes representing traditional VM hypervisors or even bare metal servers.

2.3.3 Remarks

To perform a sound analysis of the selected technologies, we classified the technologies independently. To ensure avoidance of observer bias, the analysis was executed in parallel over on-third splits of the selected technologies. Afterwards, the observations and interpretations were discussed and double-checked in several joint sessions for reconciliation. Based on this analysis and the found commonalities of the technologies, we present a categorization of technologies in the next section.

3 Categorization of technologies

During the analysis of the selected technologies (cf. Table 2), we observed that they can be divided into three categories. Before presenting the categories in detail, we briefly introduce the selected deployment technologies. Puppet, Chef, and Ansible are configuration management systems. **Puppet** enables to write reusable configuration definitions describing system resources and their state for multiple providers and services using their own domain-specific language (DSL) [36]. **Chef** uses a Ruby-based DSL. Based on a server-client architecture, it can be used to maintain and configure systems on various platforms or cloud providers [34]. **Ansible** uses a declarative YAML-based DSL to describe system configurations for various platforms and cloud services. In contrast to Chef, Ansible uses an *agentless* architecture [37]. **Kubernetes** is a platform for automating the orchestration of containerized, multi-service applications. It automatically deploys the specified application onto a cluster and ensures

that its desired configuration is reached and maintained [15]. **OpenStack Heat** is an orchestration engine that enables the description of XaaS-based applications using a YAML syntax. It manages the whole lifecycle of an deployment and provides interfaces for custom extensions [32]. **Terraform** is an orchestrator providing plugin interfaces for custom extensions. It uses its own DSL and primarily targets multi-cloud application deployments [20]. In contrast, **AWS CloudFormation** uses a DSL (JSON and YAML) to describe, deploy, and manage all infrastructure resources across Amazon's cloud services [3]. **SaltStack** is an orchestration and configuration management system using its own DSL to deploy and manage all kinds of application stacks targeting different cloud providers and services [39]. **Juju** is a topology-based application orchestration tool. It enables the modeling of application deployments using a YAML-based DSL and supports multiple cloud offerings and services [12]. **CFEngine** is an open-source configuration management tool providing enterprise functionalities by a commercial version. Its primary function is to provide automated configuration and management on top of existing computing resources [11]. Similar to AWS CloudFormation, **Azure Resource Manager** is the deployment and management service by Azure to manage resources in Microsoft's cloud environment [27]. **Docker Compose** is a framework for defining and running multi-container Docker applications. It provides a YAML-based language to specify the containers forming an application and their configuration [18]. **Cloudify** is an open-source application orchestration framework. It supports hybrid-cloud deployments for all kinds of cloud services based on a customized YAML syntax inspired by the TOSCA standard [14].

During phase three (cf. Sect. 2.3), we figured out that some technologies support multi-cloud deployments, such as Ansible or Terraform, and others are restricted to certain cloud providers, such as AWS CloudFormation and Azure Resource Manager. Further, we concluded that there are technologies suitable to deploy applications targeting XaaS. Technologies, such as AWS CloudFormation, Terraform, and Ansible, support different kinds of services for deploying components, whereas technologies, like Kubernetes and Docker Compose, are restricted to use specific *platform bundles*, container images in their cases. Therefore, we categorize the technologies in (i) general-purpose, (ii) provider-specific, and (iii) platform-specific deployment technologies. Figure 2 indicates the overlap of deployment technologies regarding the derived *deployment features and mechanisms*.

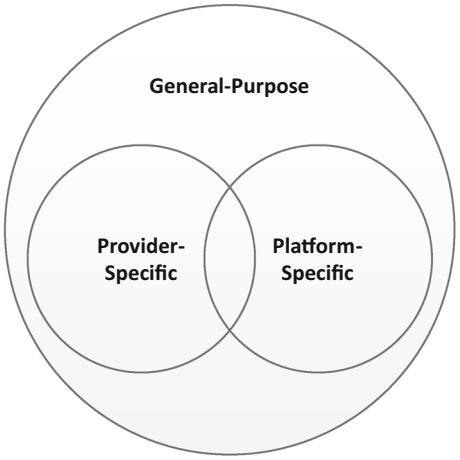


Fig. 2 Deployment automation technology categorization

General-Purpose (GP) GP technologies support all *deployment features and mechanisms* (cf. Sect. 2.3). They support single-, hybrid-, and multi-cloud deployments as well as different kinds of cloud services (XaaS). In addition, they can be extended by reusable and customized components for further providers or services. Thereby, it is possible to hook into or influence the component's lifecycle by defining custom actions. This group encompasses the following technologies: Puppet, Chef, Ansible, OpenStack Heat, Terraform, SaltStack, Juju, and Cloudify.

Provider-Specific (ProvS) ProvS deployment technologies support XaaS deployments, provide capabilities to create reusable entities, and can control a component's lifecycle (cf. deployment features and mechanisms presented in Sect. 2.3). In contrast to GP, ProvS technologies only support single-cloud deployments since they are offered by specific cloud providers, hence only supporting the cloud services offered by the respective provider. The EDMM is restricted to provider-supported component types for the cloud service components (cf. Fig. 1). This group encompasses the following deployment technologies: AWS CloudFormation and Azure Resource Manager.

Platform-Specific (PlatS) PlatS deployment technologies support multiple cloud providers, the creation of reusable entities, and influencing a component's lifecycle (cf. deployment features and mechanisms presented in Sect. 2.3). In contrast to GP, these are restricted regarding the cloud delivery model and regarding the use of specific *platform bundles* for realizing components. Considering the reference scenario (cf. Fig. 1), the EDMM is restricted such that only types and artifacts can be used that are supported by the underlying platform, e.g., Kubernetes only supports the deployment of container images. This group encompasses the following deployment technologies: Kubernetes, CFEngine, and Docker Compose.

All three groups provide deployment features and mechanisms to cover all extracted elements covered by the EDMM presented in the next section. However, the ProvS and PlatS groups are restricted in the power to express a deployment, which is explained in detail in the technology mapping section (cf. Sect. 5).

4 The essential deployment metamodel

The EDMM encompasses the essential parts of declarative deployment models. Declarative deployment models focus on the "what" and describe the structure of an application to be deployed including all components, their configuration, and relationships. The EDMM represents the result of the analysis of 13 selected declarative deployment technologies applied in industry and research. The EDMM is illustrated in Fig. 3, where the structure and names of the entities are inspired by the TOSCA standard [29] and the Declarative Application Management Modeling and Notation (DMMN) with its graph-based nature [8]. The first entities to be defined are the components forming an application, as well as the component types allowing to distinguish them and giving them semantics.

Definition 1 *(Component)* A *component* is a physical, functional, or logical unit of an application.

Definition 2 *(Component Type)* A *component type* is a reusable entity that specifies the semantics of a component that has this type assigned.

For example, a deployment model implementing the reference scenario contains several *components* (i.e., *Order App*, *Tomcat*, *JSM 1.1 Queue*, *Order Worker*, *Ubuntu LTS*, or *Azure Cosmos DB*), of different *component types* (e.g., *Order App* is a Java-based web application, while *Tomcat* is a Tomcat server). The semantics and actions required to install or terminate a component are provided by its type. While the component represents a certain functionality for a specific application, the component type can be used in different deployment models. To work as intended or to provide a higher level service, components often depends on other components. This is specified by relations between components.

Definition 3 *(Relation)* A *relation* is a directed physical, functional, or logical dependency between exactly two components.

Definition 4 *(Relation Type)* A *relation type* is a reusable entity that specifies the semantics of a relation that has this type assigned.

Concrete typed relations are also proposed by Weerasiri et al. [41] and examples of them can be found in our reference scenario (cf. Fig. 1). For instance, the relation from *Order App* to *JMS 1.1 Queue* is of type *connects to*, and it specifies that a network connection is to be set to allow the two components to communicate. The relation from *Order App* to *Tomcat* of type *hosted on*, and it indicates that the *Order App* is to be installed on the *Tomcat* server.

The actions and information required to realize the installation or termination of components and relations must be provided. The EDMM encapsulate this in operations and properties.

Definition 5 *(Operation)* An *operation* is an executable procedure performed to manage a component or relation described in the deployment model.

Definition 6 *(Property)* A *property* describes the current state or prescribes the desired target state or configuration of a component or relation.

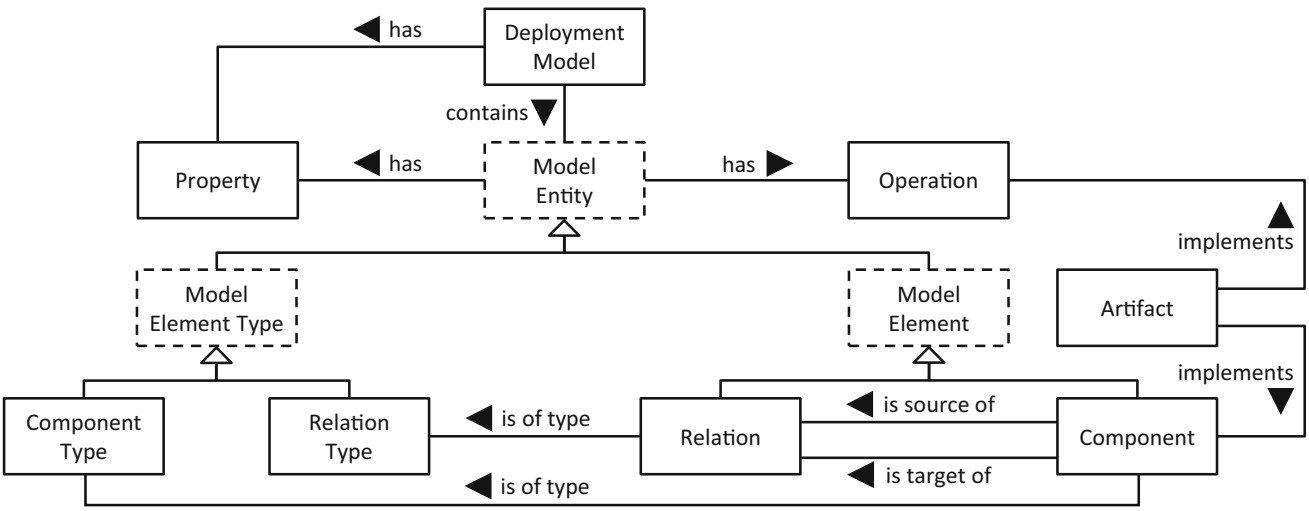


Fig. 3 The essential deployment metamodel

Consider, for instance, the *Tomcat* component in our reference application (cf. Fig. 1) which represents a Tomcat server installed on an Ubuntu VM. It can be associated with a property indicating that the exposed port of the web server must be `8080`. Further, the *Tomcat* component is also associated with the operation *install* (cf. `install.sh` script) that denotes the logic required to install the component on the corresponding VM. Components and operations are implemented through so-called artifacts. According to UML, an artifact is a physical piece of information that is created to be used for deployment and operation of a system [31]. On the one hand, there are artifacts representing an executable entity that implements an operation, e.g., a file containing the logic required to install and start a certain application component. On the other hand, there are artifacts used for the operation of a component to carry out the business logic and intended functionality, i.e., in the form container images, compiled binaries, or compressed source code.

Definition 7 *(Artifact)* An *artifact* implements a component or operation and are required for their execution.

For instance, consider the *Admin App* component in the reference scenario (cf. Fig. 1). The shell script to install the corresponding component represents an artifact that is associated with the *install* operation. In contrast, the compiled binary file of the *Admin App* also represents an artifact but this file is needed to materialize an instance of this component and is required for the operation.

Finally, note that all model entities, such as components, relations, their types as well as properties and operations are contained in a so-called deployment model. Component and relation types are usually defined across deployment models to enable reuse including their operations as artifacts, whereas artifacts implementing a component are typically referenced in the deployment model itself. Further, deployment models define properties, which can be referenced and used by contained elements. A deployment model does not have to consist of a single file, instead, it can be a set of files, which semantically belong together. For example, several technologies allow to reference or import component types from different sources, others require a self-contained deployment model. At the time of deployment, all entities must be available for the underlying runtime that are referenced or contained in the deployment model.

Definition 8 *(Deployment Model)* A *deployment model* describes declaratively the desired target state of an application including all necessary model entities.

The target state is the completed deployment of all components and relations according to the specified properties, as described in the deployment model.

The formalized essential entities of a declarative deployment model are not only resulting from our analysis. Similar elements have already been discussed in other research studies [5,8,9], even if not based on a systematic review of existing and well-established declarative deployment automation technologies.

5 EDMM technology mapping

In this section we show how the concepts of EDMM can be semantically mapped to the selected deployment technologies. The mapping is structured into three subsections according to the categories presented in Sect. 3.

5.1 EDMM to GP-technology mapping

We hereby show how the EDMM can be semantically mapped to Puppet, Chef, Ansible, OpenStack Heat, Terraform, SaltStack, Juju, and Cloudify. All of these technologies support the required features and mechanisms as outlined in Sect. 2.3.

5.1.1 Mapping to Puppet

In Puppet, *resources* are the main building blocks describing certain aspects of the system. For example, in our reference application, the deployment of a component can be described by a *resource*. The semantic and structure of a resource is defined by its assigned resource type (cf. Definition 2). Puppet provides a set of built-in resource types which can be extended by custom resource types written in Ruby. Resources can be bundled to *modules* enabling the encapsulation of logical parts into reusable entities (cf. Definition 2), e.g., the stack of the *Order App* in our reference scenario can be encapsulated using a *module*. Resources or modules can be used in the resulting *deployment model* and can be mapped to *components* encompassing certain semantics. Both, modules and resources utilize *properties*, e.g., to define a port a web server is listening on. In addition, *artifacts* can be defined that implement the component, e.g., by defining in a module to use a compressed version of the Tomcat web server. Modules contain a set of *classes* expressing the logic to converge components into a certain state and can be mapped to *operations* in EDMM. In Puppet, *relations* between components can be expressed on different levels and are limited to a predefined set of *relation types*. By including another module the semantic of a "depends on" relation type can be expressed. On the level of classes, a "depends on" relation can be defined by using a predefined `require` function.

5.1.2 Mapping to chef

Chef uses *cookbooks* to structure and encapsulate logical parts of a *deployment model*. Cookbooks can be mapped

to *components* as well as to *component types* in our metamodel. Using cookbooks, reusable entities can be defined expressing certain semantics, which map to *component types*. Further, cookbooks can be imported into other cookbooks, which maps semantically to *components* in this context. Our reference application (Fig. 1) can be expressed in a single cookbook by importing existing ones from the Chef supermarket, e.g., a cookbook to install a Tomcat web server. In Chef, all kinds of *artifact* are supported as long as they can be referenced and packaged using the provided mechanisms. The actual *operations* to be executed to install or configure a component are encapsulated in so called *recipes* written in Ruby. Cookbooks can have dependencies to other cookbooks by using the `depends` attribute in the respective meta data file. With `include_recipe`, operations from dependent cookbooks can be integrated in the sequence of operations required to reach a desired state. Thus, *relations* of one *relation type* can be expressed. By using an attribute file that, for example, defines the desired port of a web server, inputs for *properties* in the recipes can be represented in Chef.

5.1.3 Mapping to Ansible

Ansible and Chef are similar in their concepts. Like in Chef, playbooks are the elements used to create *deployment models*. For example, the complete reference scenario can be expressed in a single playbook. As playbooks can also encapsulate logical and reusable parts, they can be mapped to *components* and *component types* due to their recursive aggregation behavior. There can be a generic "Tomcat" playbook that enables the deployment and configuration of an Tomcat web server. Playbooks can also define *variables* that are mapped to *properties* in EDMM. In Ansible, *relations*, such as a web application is hosted on a web server, are implicitly defined by importing other Playbooks and can only expressed a "depends on" *relation type*. Ansible uses the concept of "roles", which contain "tasks" to converge a system to a desired state. Roles are part of playbooks and mapped to *operations* and, therein, all kinds of *artifacts* are supported that implement a component.

5.1.4 Mapping to OpenStack Heat

The *deployment model* for OpenStack Heat is called Heat Orchestration Template (HOT). The logical parts of an application, i.e., its components are modeled as *resources*. Several resource types are provided by Heat and further plugins for other resources are already available (e.g., Docker or AWS) or can be created. These resource types are reusable entities that specify the *properties* and *operations* that can be executed on a resource of this type. They form the *component types* in Heat. For deployments on a VM (such as for the Admin App in Fig. 1) an infrastructure, a `Heat::SoftwareConfig`, and a `Heat::SoftwareDeployment` resource are used. The supported *artifacts* that implement a component depend on the resource type. With a `SoftwareConfig` resource (restricted to be used only with other IaaS resources) arbitrary *operations* can be specified and linked to *implementation artifact*, e.g., in the form of executable scripts. To express dependencies between components, the `dependsOn` attribute can be used where one or more other components can be referenced. Further types of *relations* and, thus, *relation types* can be expressed using certain properties of a component.

5.1.5 Mapping to Terraform

In Terraform, *resources* are used to describe elements of a *deployment model*, e.g., compute instances, virtual networks, or software components. Each resource is assigned to a *resource type* that determines the kind of element that is managed and specifies *properties* in the form of so-called *attributes*. Several resource types are provided by Terraform and custom resource types can be written in Go. *Resources* and *resource types* are mapped to *components* and *component types* respectively. The supported *artifacts* that implement a component depend on the *resource type*. In addition, *provisioners* can be defined that are executed as part of the creation or destruction of a resource. For example, the *remote-exec provisioner* can be used to define arbitrary *operations* that are executed on a resource after its creation, e.g., downloading and installing an Apache Tomcat server on a provisioned EC2 instance. Explicit dependencies between resources can be expressed by the use of the *depends_on* attribute. Further, *modules* can be used to create logical, reusable groups of resources. *Input Variables* on modules are mapped to *properties* and are used to parameterize and customize modules. For example, each stack of the reference scenario can be expressed by a Terraform module specifying the required resources. Hence, *modules* are mapped to *components* and *component types* in EDMM.

5.1.6 Mapping to SaltStack

SaltStack is a flexible orchestration and configuration management tool, which uses so-called *top files* to create a *deployment model*. Top files contain *formulas* and *states*. Formulas are independent and reusable entities and map to *component types* in EDMM. For example, one can create a Tomcat formula that encapsulates the logic to install and start a Tomcat web server. Further, a formula can define a set of configuration values that map to *properties* in EDMM. Moreover, a logical group of states defined in formulas are expressing *operations* which in turn relate to *artifacts* that implement these. Using their DSL syntax, arbitrary logic in

the form of states can be supplied to reach a desired state. By using a formula in a deployment model a *component* is created according to our metamodel. *Relations* in SaltStack are, on the one hand, derived by the sequence of used formulas in a deployment model. On the other hand, states can *depend on* other stats defined by certain formulas, which results in a certain execution order.

5.1.7 Mapping to Juju

Juju is a topology-based application modeling tool based on a declarative YAML DSL. All instructions and artifacts necessary for deploying and configuring application components are defined in *charms*, which map to *component types* in the EDMM as they are reusable entities having a certain semantic. Each charm provides a set of configuration values that can be set during deployment, which are mapped to *properties*. Further, a charm defines actions, implemented as scripts, that are triggered by the runtime during deployment. These are respectively mapped to *operations* and *artifacts*, whereas artifacts that implement components are carried out by these operations. Charms can be used in *bundles*, which implies that a *component* of a certain *component type* inside a *deployment model* is used. In Juju, there can be *relations* following a "depends on" semantic by expressing requirements and capabilities on charms. For example, charm defines that it requires a database and, correspondingly, a database charm is capable of satisfying this requirements. This can be expressed using the `relations` keyword in the model. A compound deployment including multiple charms, their configuration, and relations can be described in a Juju bundle.

5.1.8 Mapping to Cloudify

The DSL defined and used by Cloudify is based on the TOSCA YAML profile [30]. However, the standard is not completely met. Similar to OpenStack Heat, built-in types encompassing Cloudify basic types can be used to model *components*. Further *components types* can be made available using plugins. In Cloudify, *node types* and *node templates* are mapped to *component types* and *components* respectively according to the EDMM definitions. Using Cloudify's lifecycle interface, i.e., *operations* allow to create, start, stop, and terminate physical resources. Node types define *properties*, implement operations, and define deployment as well as implementation *artifacts*. Built-in relation types, for example, define a `depends_on` or `connected_to` relation between components. In contrast to all other considered technologies, further relation types can be defined. To realize the reference scenario from Fig. 1, the AWS and Azure plugins are required. These plugins provide all types, operations, and artifacts to model the required components as well as to interact with the respective cloud provider application-programming-interfaces (API).

5.2 EDMM to ProvS-technology mapping

We hereby show how the EDMM can be semantically mapped to AWS CloudFormation and Azure Resource Manager. In contrast to GP, these technologies only support single-cloud deployments as they only support the services of the respective cloud provider.

5.2.1 Mapping to AWS CloudFormation

AWS CloudFormation is the deployment and management service by AWS and uses a JSON or YAML templates to create *deployment models*. In the template, *resources* are used to express *components*. AWS provides a set of built-in *resource types*, referred as *component types*, that specify the semantics of components, e.g., defining *properties* supported by a resource. CloudFormation enables to create reusable *component types* by defining *stacks* that can be in turn used in other templates. Each stack of the reference scenario in Fig. 1 can be modeled by one or more resources. For example, the AWS SQS and Lambda service an be used to implement the depicted *JMS 1.1 Queue* and *Order Worker* components. To deploy the *Admin App* an AWS EC2 instance can be defined where one can specify the required installation and configuration steps as an *operations*, provided in separate files. The semantic of relations is restricted as only one type of inter-component dependency can be specified, by using the attribute `dependsOn` on resources.

5.2.2 Mapping to Azure Resource Manager

In Azure Resource Manager (ARM), JSON templates describe the configuration of Azure resources and services. Azure services (e.g., compute instances, databases, or middleware) are modeled as *resources*. The structure and semantics of a resource (e.g., its supported *properties* and *artifacts*) are defined by built-in *resource types*. Hence, *resource* and *resource type* map to *component* and *component type* in the EDMM. For example, the MongoDB configuration hosted on Azure Cosmos DB can be expressed using a resource of type `Microsoft.DocumentDB/databaseAccounts`.
Relations between resources can be specified using the `dependsOn` element defining a dependency to one or more resources. The resources of a deployment can either be defined in a single template or divided into multiple ones in order to create purpose-specific, reusable templates. As ARM templates are logical and reusable units, they are mapped to *components* and *component types* in the EDMM. Post-deployment configurations, software installations, or other actions to configure a VM, can be achieved through *virtual*

machine extensions, which are semantically mapped to *operations* in the EDMM.

5.3 EDMM to PlatS-technology mapping

We hereby show how the EDMM can be semantically mapped to Kubernetes, CFEngine, and Docker Compose. In contrast to GP, these technologies are restricted to specific services (XaaS support) and to the use of specific *platform bundles* for realizing components.

5.3.1 Mapping to Kubernetes

With Kubernetes, developers can specify the *deployment model* of a multi-service application by indicating the "pods" to run, one for each *service* forming the application. Their desired configuration can then be specified by defining "deployments", each targeting a different subset of pods. For instance, each component in our reference application (cf. Fig. 1) should be placed in a different pod, and its desired configuration could be specified by defining a different deployment targeting its corresponding pod. Kubernetes hence provides a predefined set of *component types* allowing to define reusable units of pods, deployments, and services (*components* of an application). Kubernetes also supports a predefined set of *relation types* in the form of specifying which pods are targeted by which deployment or service. Attributes specifying the desired configuration for pods, deployments, and services are mapped to *properties* in the EDMM. In Kubernetes, *artifacts* for implementing *components* are reflected by container images, which contain the complete stack starting from the operating system to the application-specific component, depending on application requirements [35]. Moreover, container images also encapsulate *operations* representing the logic to install and configure the components. Therefore, container images are *platform bundles* as they are the unit of deployment.

5.3.2 Mapping to CFEngine

CFEngine assumes an already running computing infrastructure and, therefore, is assigned to the PlatS deployment technologies. In CFEngine, everything is a *promise*. Promises are used to define the desired state that should be reached, e.g., a package to be installed or a process to be started. Further, *bundles* can be used to logically group promises and are, therefore, mapped to *operations* in EDMM. *Bodies* are used to create reusable parts of promises and are mapped to *component types*. Bodies can also define *properties* and once they are used in a *deployment model* a concrete *component* is created. There can be explicit *relations* between components that specify the required execution order using the `depends_on` property on promises.

5.3.3 Mapping to Docker Compose

Docker Compose permits specifying the *deployment model* of a multi-container Docker application in a single file. The file is organized in "services", which are used to specify the *components* (i.e., its containers) of an application. *Relations* between components are expressed using the `depends_on` keyword. Mapping this to our metamodel, the only *component type* and *relation type* are expressed by "services" and by the `depends_on` attribute, respectively. Docker Compose predefines the set of *properties* that can be associated with the services forming an application. *Properties* can be associated to each service in a Docker Compose file and specify the desired configuration. *Artifacts* and *operations* are a special case since they have to be packaged as a so-called *platform bundle*. The artifact implementing a component as well as the logic to install and configure it must be supplied through a Docker image, either retrieved from a repository or built based on a `Dockerfile`. For example, to deploy the *Order App* the complete stack starting from the operating system to the application-specific component must be linked into a Docker image.

5.4 EDMM to TOSCA

As various technologies support the TOSCA standard, i.e., OpenTOSCA [7], ALIEN 4 Cloud [1], Cloudify, and TosKer [10], this section presents a mapping of EDMM to TOSCA—although it was out of scope of this paper due to its rank. EDMM only uses a subset of entities specified by the standard: *Service templates* are used to express *deployment models*, while *components* and *component types* are referred as *node templates* and *node types*, respectively. TOSCA allows the definition of arbitrary *relations* and *relation types*, called *relationship template* and *relationship type*, but defines a certain set of normative types every compliant orchestrator needs to support, i.e., `hosted_on` and `connected_to`. Node types and relationship types support the definition of *properties* as they are used to define semantics. Further, *operations* in EDMM are mapped to *management operations*, which are realized by *implementation artifacts*. In contrast, there are *deployment artifacts* that are mapped to *artifacts* required for the execution of a component.

6 Related work

In this section we discuss closely related work on reviews and comparisons regarding cloud computing deployment technologies and their respective meta modeling results. Weerasiri et al. [41] introduced a taxonomy for cloud resource orchestration based on a survey examining eleven cloud orchestration approaches. They describe the notion of

a *Resource Entity Model* consisting of entities, relationships, and constraints. The Resource Entity Model shows on a high level the structure of a cloud application. However, to automate a deployment more information are required, e.g., what artifacts have to be installed and how. Therefore, we have examined the semantics of used deployment technologies and formulated a metamodel containing the common elements.

A detailed comparison of six different *Infrastructure-as-Code* platforms has been conducted by Masek et al. [26]. They distinguish between "configuration management" tools, designed to install and manage software on existing nodes, and "orchestration tools", designed to provision the servers themselves and leaving the job of configuring nodes to other tools. Also, Wettinger et al. [42] introduce a similar classification by differentiating between node-centric and environment-centric artifacts. Node-centric artifacts are deployment models that are executed on single nodes, such as Chef or Ansible, whereas environment-centric artifacts are deployment models that are executed on a higher level including more that one node, such as Terraform. In both works, the essence is that these two categories are not mutually exclusive. Most configuration management tools can do some degree of provisioning and most orchestration tools can do some degree of configuration management or can even integrate other tools. We derived a different grouping criteria resulting from our review in order to make a clear assignment for each technology.

Besides the variety of tools, there are also standards in the field of application deployment. Markoska et al. [25] give an brief overview about different cloud deployment technologies and standards, such as AWS CloudFormation, TOSCA, CAMP, and others. Di Martino et al. [17] focuses only on the TOSCA standard and OpenStack Heat templates for a qualitative comparison. Further, Bergmayr et al. [6] provide a very detailed overview and comparison of different cloud modeling languages (CML). They conducted a systematic review of CMLs, their features, and discuss core domain concepts of such. However, these studies do not review a variety of used deployment automation technologies and, further, do not abstract to a commonly denominated metamodel.

Vergara-Vargas and Umaña-Acosta [40] developed a new Architecture Description Language (ADL) that comprises deployment aspects. One part of their ADL is to support software deployments based on a model-driven deployment (MDDep) approach expressing components and relations among them. Alipour and Liu [2] focuses on a model-driven approach presenting a *Cloud Platform Independent Model* in order to deploy auto-scaling services in a cloud-agnostic way. In general literature about software architectures (SAs), SAs are described as structures that comprise software elements, relations among them, and properties of both [38]. Also Chen [13] describes in his paper a unified view to model data, consisting of entities, and relationships. Thus, these elements are similar to the parts identified by the EDMM. Further, representing an application structure as a graph is a common approach in research. For example, GENTL [5] is a CML to express topologies whereas Breitenbücher [8] proposes a graph-based description language (DMMN) to enable the declarative modeling of management activities [8]. However, as they are proposing similar elements, these findings are not based on the analysis of widely used industrial tools.

7 Conclusions and outlook

We conclude that there are three EDMM groups a technology can be assigned to: General-purpose (GP) technologies support multi-cloud and XaaS deployments, provider-specific (ProvS) technologies are restricted to the respective cloud provider services, and platform-specific (PlatS) technologies support only a subset of XaaS and require specific platform bundles for deployment. A understanding of essential deployment model elements helps to compare technologies regarding deployment features and mechanism and supports decision making processes when selecting an appropriate technology for an use case. The introduced classification and the presented EDMM technology mapping support the migration from one deployment technology into another one. Further, this does not only support industry to compare and select technologies, but also helps researcher to evaluate concepts in the area of deployment automation research: If new research can be realized using EDMM, our mappings prove that this research can be also applied to the technologies analyzed in this paper. This significantly eases practically validating new concepts.

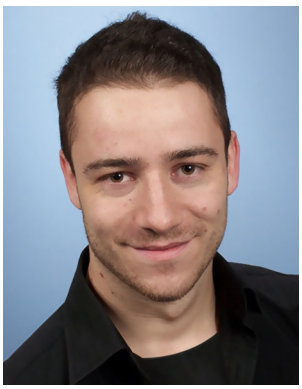
Acknowledgements This work is partially funded by the European Union's Horizon 2020 research and innovation project *RADON* (825040), the BMWi projects *IC4F* (01MA17008G) and *SePiA.Pro* (01MD16013F), the DFG project ADDCompliance (636503), the POR-FSE project *AMaCA*, and the project *DECLware* (PRA_2018_66, University of Pisa).

References

1. ALIEN 4 Cloud (2018) ALIEN 4 cloud official site. http://alien4cloud.github.io. Accessed 14 Aug 2019
2. Alipour H, Liu Y (2018) Model driven deployment of auto-scaling services on multiple clouds. In: 2018 IEEE international conference on software architecture companion (ICSA-C). IEEE, pp 93–96

3. Amazon Web Services, Inc (2018) AWS CloudFormation official site. https://aws.amazon.com/de/cloudformation. Accessed 14 Aug 2019
4. Amazon Web Services, Inc (2018) AWS OpsWorks official site. https://aws.amazon.com/de/opsworks
5. Andrikopoulos V, Reuter A, Gómez Sáez S, Leymann F (2014) A GENTL approach for cloud application topologies. In: Villari M, Zimmermann W, Lau KK (eds) Service-oriented and cloud computing. ESOCC 2014. Lecture Notes in Computer Science, vol 8745. Springer, Berlin, Heidelberg
6. Bergmayr A, Breitenbücher U, Ferry N, Rossini A, Solberg A, Wimmer M, Kappel G (2018) A systematic review of cloud modeling languages. ACM Comput Surv (CSUR) 51(1):1–38
7. Binz T, Breitenbücher U, Haupt F, Kopp O, Leymann F, Nowak A, Wagner S (2013) OpenTOSCA—A runtime for TOSCA-based cloud applications. In: Proceedings of the 11th international conference on service-oriented computing (ICSOC 2013). Springer, pp 692–695
8. Breitenbücher U (2016) Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Ph.D. thesis, Universität Stuttgart
9. Brogi A, Soldani J, Wang P (2014) TOSCA in a nutshell: promises and perspectives. In: Service-oriented and cloud computing: 3rd European conference, ESOCC 2014, Manchester, UK, September 2–4, 2014. Proceedings, Springer, vol 8745, pp 171–186
10. Brogi A, Rinaldi L, Soldani J (2018) TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications. Softw Pract Exp 48(11):2061–2079
11. Burgess M, College O (1995) Cfengine: a site configuration engine. In: USENIX computing systems
12. Canonical Ltd (2018) Juju Official Site. https://jujucharms.com. Accessed 14 Aug 2019
13. Chen PPS (1976) The entity-relationship model—toward a unified view of data. ACM Trans Database Syst 1(1):9–36
14. Cloudify Platform Ltd (2018) Cloudify Official Site. https://cloudify.co. Accessed 14 Aug 2019
15. CNCF (2018) Kubernetes Official Site. https://kubernetes.io. Accessed 14 Aug 2019
16. Delaet T, Joosen W, Vanbrabant B (2010) A survey of system configuration tools. In: Proceedings of the 24th international conference on large installation system administration (LISA 2010), USENIX
17. Di Martino B, Cretella G, Esposito A (2015) Defining cloud services workflow: a comparison between TOSCA and OpenStack Hot. In: 2015 Ninth international conference on complex, intelligent, and software intensive systems (CISIS). IEEE, pp 541–546
18. Docker, Inc (2018) Docker Compose Documentation. https://docs.docker.com/compose. Accessed 14 Aug 2019
19. Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J (2017) Declarative vs. imperative: two modeling patterns for the automated deployment of applications. In: Proceedings of the 9th international conference on pervasive patterns and applications. Xpert Publishing Services (XPS), pp 22–27
20. HashiCorp (2018a) Terraform Official Site. https://www.terraform.io. Accessed 14 Aug 2019
21. HashiCorp (2018b) Vagrant Official Site. https://www.vagrantup.com. Accessed 14 Aug 2019
22. Herry H, Anderson P, Wickler G (2011) Automated planning for configuration changes. In: Proceedings of the 25th international conference on large installation system administration (LISA 2011). USENIX, pp 57–68
23. Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional, Boston
24. Humble J, Molesky J (2011) Why enterprises must adopt devops to enable continuous delivery. Cut IT J 24(8):6
25. Markoska E, Chorbev I, Ristov S, Gusev M (2015) Cloud portability standardization overview. In: 2015 38th International convention on information and communication technology, electronics and microelectronics (MIPRO). IEEE, pp 286–291
26. Masek P, Stusek M, Krejci J, Zeman K, Pokorny J, Kudlacek M (2018) Unleashing full potential of ansible framework: University labs administration. In: 2018 22nd conference of open innovations association (FRUCT), pp 144–150. https://doi.org/10.23919/FRUCT.2018.8468270
27. Microsoft, Inc (2018) Microsoft Azure ARM Official Site. https://azure.microsoft.com/en-us/features/resource-manager. Accessed 14 Aug 2019
28. Morris K (2016) Infrastructure as code: managing servers in the cloud. O'Reilly Media, Sebastopol
29. OASIS (2013) Topology and orchestration specification for cloud applications (TOSCA) version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html
30. OASIS (2019) TOSCA simple profile in YAML version 1.2. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html
31. OMG (2015) Unified modeling language (UML) version 2.5. https://www.omg.org/spec/UML/2.5.1/PDF
32. OpenStack Foundation (2018) OpenStack HEAT documentation. https://wiki.openstack.org/wiki/Heat. Accessed 14 Aug 2019
33. Oppenheimer D, Ganapathi A, Patterson DA (2003) Why do internet services fail, and what can be done about it? In: Proceedings of the 4th conference on USENIX symposium on internet technologies and systems (USITS 2003). USENIX
34. Opscode, Inc (2018) Chef official site. http://www.opscode.com/chef. Accessed 14 Aug 2019
35. Pahl C, Brogi A, Soldani J, Jamshidi P (2017) Cloud container technologies: a state-of-the-art review. IEEE Trans Cloud Comput
36. Puppet Labs (2018) Puppet official site. http://puppetlabs.com/puppet/what-is-puppet. Accessed 14 Aug 2019
37. Red Hat, Inc (2018) Ansible official site. https://www.ansible.com. Accessed 14 Aug 2019
38. Rozanski N, Woods E (2012) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, Boston
39. SaltStack, Inc (2018) SaltStack official site. https://www.saltstack.com. Accessed 14 Aug 2019
40. Vergara-Vargas J, Umaña-Acosta H (2017) A model-driven deployment approach for scaling distributed software architectures on a cloud computing platform. In: 2017 8th IEEE international conference on software engineering and service science (ICSESS). IEEE, pp 99–103
41. Weerasiri D, Barukh MC, Benatallah B, Sheng QZ, Ranjan R (2017) A taxonomy and survey of cloud resource orchestration techniques. ACM Comput Surv 50(2):26:1–26:41
42. Wettinger J, Breitenbücher U, Kopp O, Leymann F (2016) Streamlining DevOps automation for cloud applications using TOSCA as standardized metamodel. Future Gener Comput Syst 56:317–332

Michael Wurster is a research associate at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. He completed his master's degree at the Reutlingen University end of 2016. His research interests lie mainly in the DevOps-enabled provisioning, orchestration, and management of highly complex and distributed application systems. Furthermore, Michael has six years of practical experience in working as a Software Engineer focusing on developing microservice-based systems in a DevOps-enabled organization.



Uwe Breitenbücher is a research staff member and post-doc at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. His research vision is to improve cloud application provisioning and application management by automating the application of management patterns. Uwe was part of the CloudCycle project, in which the OpenTOSCA Ecosystem was developed. His current research interests include cyber-physical systems, blockchains, and microservices.

Michael Falkenthal is a research associate and Ph.D. student at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. He studied business information technology at the Universities of Applied Sciences in Esslingen and Reutlingen focusing on business process management, services computing and enterprise architecture management. Michael gained experience in several IT transformation and migration projects at small- to big-sized companies. His current research interests are fundamentals on pattern language theory, cloud computing and quantum computing.

Christoph Krieger received the Master of Science degree in Software Engineering from the University of Stuttgart in 2018. Currently, he works as a research associate at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. His research interests are in the area of architectural design decisions and how their compliant realization can be ensured during different stages of the development life cycle.

Frank Leymann is a full professor of computer science and director of the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. His research interests include service-oriented architectures and associated middleware, workflow- and business process management, cloud computing and associated systems management aspects, and patterns. Frank is co-author of more than 400 peer-reviewed papers, about 70 patents, and several industry standards. He is elected member of the Academy of Europe.

Karoline Saatkamp received the Master of Science degree in Information Systems from the University of Stuttgart and University of Hohenheim in 2016. Currently, she works as a research associate at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. Her research interests are in the area of multi-cloud deployment and management, focusing on the distribution of application fragments and their cross-cloud communication behavior.

Jacopo Soldani is a post-doc researcher at the University of Pisa (Italy). He holds a Ph.D. in Computer Science (2017, University of Pisa). His research interests include, but are not limited to, serviceoriented and cloud computing, adaptation, coordination, and integration of software elements, and formal methods. He is member of the IFIP Working Group on Service-Oriented Systems (IFIP WG 2.14/6.12/8.10) and of the OASIS TOSCA technical committee, and he has also been involved in several research projects on service, cloud and fog computing both at local and EU level.