Institute of Architecture of Application Systems

Automated Generation of Management Workflows for Applications Based on Deployment Models

Lukas Harzenetter, Uwe Breitenbücher, Frank Leymann, Karoline Saatkamp, Benjamin Weder, Michael Wurster

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{harzenetter, breitenbuecher, leymann, saatkamp, weder, wurster}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Harzenetter2019_AutomatedManagementWorkflows,
    author    = {Lukas Harzenetter and Uwe Breitenb{\"u}cher and Frank Leymann
                 and Karoline Saatkamp and Benjamin Weder and Michael Wurster},
    title     = {{Automated Generation of Management Workflows for Applications
                  Based on Deployment Models}},
    booktitle = {2019 IEEE 23\textsuperscript{rd} International Enterprise
                 Distributed Object Computing Conference (EDOC)},
    publisher = {IEEE},
    pages     = {216--225},
    month     = dec,
    year      = 2019
}
```


Universität Stuttgart
Germany

Automated Generation of Management Workflows for Applications Based on Deployment Models

Lukas Harzenetter, Uwe Breitenbücher, Frank Leymann, Karoline Saatkamp, Benjamin Weder, and Michael Wurster
Institute of Architecture of Application Systems, University of Stuttgart, Germany
{harzenetter, breitenbuecher, leymann, saatkamp, weder, wurster}@iaas.uni-stuttgart.de

***Abstract*—To automate the deployment of applications several deployment technologies have been developed. However, the management of deployed applications is only partially covered by existing approaches: While management functionalities such as scaling components or changing their configurations are covered directly by cloud providers or configuration management technologies such as Chef, holistic management processes that affect multiple components probably deployed in different environments cannot be automated using these approaches. For example, testing all deployed components and their communication or backing up the entire application state that is scattered across different components requires custom management logic that needs to be implemented manually, e. g., using scripts. However, a manual implementation of such management processes is error-prone, time-consuming, and requires immense technical expertise. Therefore, we propose an approach that enables automatically generating executable management workflows based on the declarative deployment model of an application. This significantly reduces the effort for automating holistic management processes as no manual implementation is required. We validate the practical feasibility of the approach by a prototypical implementation based on the TOSCA standard and the OpenTOSCA ecosystem.**

***Index Terms*—Management Automation; Application Management; Deployment Models; TOSCA; Workflows**

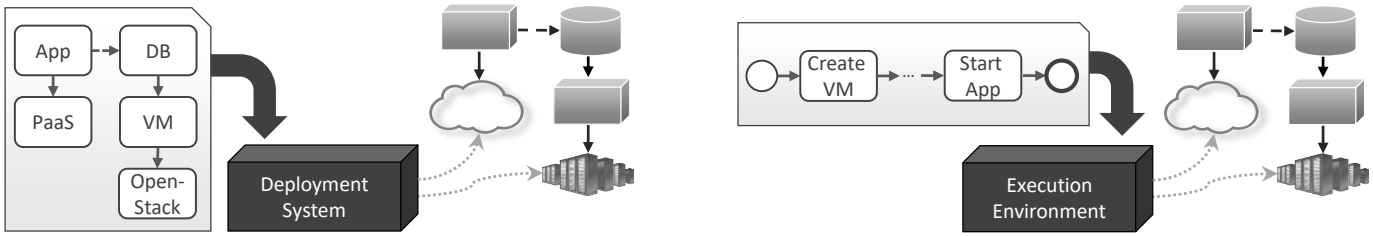
I. Introduction

The complexity of today's applications is constantly increasing as they are composed of more and more interacting components. A manual deployment and configuration of such complex composed applications is error-prone and time-consuming [1]. Therefore, a plethora of technologies for automating the deployment of applications have been developed in recent years. This includes general-purpose technologies, such as Chef [2] or Terraform [3], as well as provider-specific technologies, such as CloudFormation for Amazon Web Services (AWS) [4]. While provider-specific technologies often only support single cloud deployments, general-purpose technologies mostly enable multi-cloud deployments for distributed applications. To automatically deploy an application, many of these technologies support *declarative deployment models*, which describe the structure of the application to be deployed including all components and their relations [5], [6]. Based on these models, the deployment system automatically derives and runs the tasks that need to be executed [7]. In contrast, *imperative deployment models* explicitly specify the tasks and the order in which they must be executed in the form of a process model that can be implemented, for example, using scripting or workflow languages such as BPMN [8] or BPEL [9].

Although the deployment of applications can be automated completely using such technologies, automating the management of deployed applications is only supported in a very limited way: While management functionalities such as scaling or reconfiguring application components are typically natively supported by cloud providers or declarative configuration management technologies, holistic management functionalities that affect multiple components which are possibly distributed across environments are only supported to a limited extent. For example, to perform backups of all stateful components or to test the availability of all deployed components, a manual implementation of such functionalities is required as multi-environment deployments can neither be managed by a single provider, nor do declarative configuration management technologies support tasks such as backing up a database or testing components. Unfortunately, manually implementing such functionalities, for example, in the form of an imperative script or workflow, requires immense technical knowledge and is time-consuming, error-prone, and, if the deployment model changes over time, it quickly becomes outdated [10], [11].

To tackle these issues, we address the following research question in this paper: *How can executable management workflows for different kinds of functionalities be automatically derived from a declarative deployment model of an application?* Our approach consumes declarative deployment models and enriches the modeled components with component-specific management operations that are provided by reusable component types. Based on this enrichment, an executable management workflow gets generated that orchestrates these operations to execute the desired management functionality, e.g., to test all involved components. As a result, the effort required to enable the automated execution of management functionalities is significantly reduced as no manual implementation is required. We validate the practical feasibility of our approach by a prototypical implementation based on the TOSCA standard [12], [13] and the OpenTOSCA ecosystem [14] and present two case studies showing practical management use cases.

The rest of this paper is structured as follows: Section II provides fundamentals and a detailed motivating scenario. In Section III the management feature enrichment and workflow generation approach is introduced. Section IV presents the prototypical implementation and Section V illustrates two case studies. In Section VI the related work is discussed and Section VII concludes the paper and outlines future work.



(a) Declarative Deployment Model

(b) Imperative Deployment Model

Figure 1. Deployment Model Approaches

II. Fundamentals and Motivating Scenario

In this section, we first introduce fundamentals about deployment and management automation in Sections II-A and II-B. Afterwards, we describe the motivating scenario in Section II-C.

A. Deployment Models

To automate the deployment of applications, several technologies, such as Puppet, Chef, and Terraform, have been developed. Many of these deployment technologies are based on *deployment models*. These deployment models can be processed by the respective deployment system which is capable of deploying the modeled application. Using the definition of Endres et al. [5], deployment models can be classified into two general approaches as shown in Figure 1: Declarative models (see Figure 1a) and imperative models (see Figure 1b). Declarative deployment models specify the desired state of an application, while imperative deployment models define the tasks and their execution order to reach a specific state.

In previous work [6], we investigated 13 deployment technologies to derive the *Essential Deployment Meta Model (EDMM)* that describes common features supported by the analyzed technologies. Thus, EDMM enables creating declarative deployment models that (i) are vendor- and technology-agnostic and (ii) that can be mapped to arbitrary technologies. We use EDMM as basis in this paper to provide a technology-agnostic approach. A declarative deployment model based on EDMM describes the application components and their dependencies. *Components* are, for example, a Java application or a Platform as a Service (PaaS) [15] offering. Hereby, components are typed by *component types* which define *properties* and *operations* for the component. For instance, to use AWS Beanstalk, user credentials must be provided and for the Java application at least an operation to install it is required. Operations and components can be implemented by so-called *artifacts* which can be, in the case of a web application, a WAR file, or any other executable software artifact. To describe the dependencies of components, *relations* specified by a *relation type*, e. g., hostedOn or connectsTo, are used. Hence, a web application can be hosted on AWS Beanstalk and may connect to a database (DB) that is hosted on a virtual machine (VM) in an on-premise infrastructure such as OpenStack (see Figure 1a). By interpreting such a declarative deployment model, a deployment system can derive the required steps to deploy the application.

Even though a declarative modeling approach is intuitive, it has some limitations: For example, the order in which tasks are executed to reach a desired state cannot be customized and application-specific tasks cannot be implemented in an arbitrary manner [5], [16]. Thus, *imperative deployment models* are required for modeling complex application deployments as they enable the specification of arbitrary tasks that are performed during the deployment process. Examples for imperative deployment technologies are workflow languages such as BPMN4TOSCA [17]. Also, general-purpose technologies, such as scripting languages, are often used to implement deployment and management processes. In contrast to a declarative deployment model, an imperative deployment model, as depicted in Figure 1b, explicitly describes what activities must be executed in which order. For example, one activity invokes the API of a cloud provider to create a new VM, while the next one establishes an SSH connection to the VM and executes shell commands in order to install a web server. Thus, implementing these activities quickly becomes a significant challenge as (i) immense technical expertise is required, (ii) this approach is error-prone, and (iii) the implementations become outdated quickly if the application changes.

B. Automating Management Functionalities

Holistic management functionalities that involve multiple, distributed components are hard to automate [16]. First, management functionalities offered by providers are limited to the hosted components. Thus, if multiple providers are involved, the individual management functionalities must be orchestrated. For example, if an application consists of multiple storage components hosted by different cloud providers, e.g., database or cache services, several provider APIs need to be orchestrated to backup the entire system. In contrast, available declarative configuration management technologies are able to support multi-environment deployments, but are limited in the management tasks they support. For example, state-preserving management tasks such as testing the HTTP connection between two components cannot be modeled declaratively [16]. Thus, to automate holistic management functionalities, typically several individual technologies must be orchestrated by a program, script, or workflow [10]. Typical management functionalities, besides testing and backup, are, e. g., extending the license of proprietary components, installing security updates on all VMs, or adding a new user to a VM.

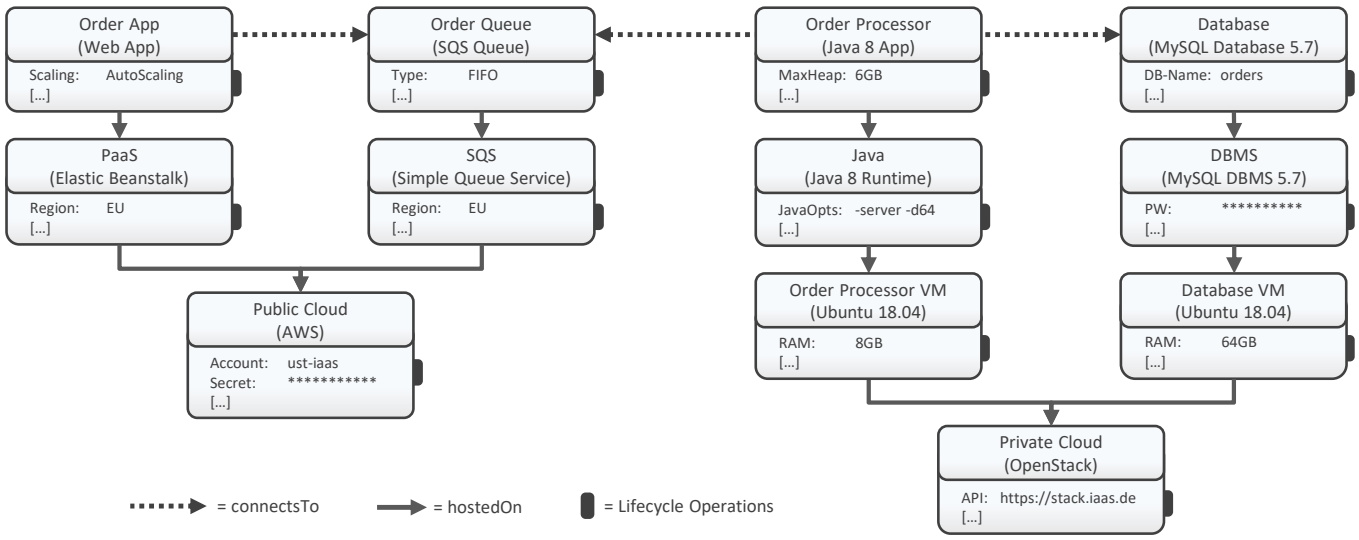


Figure 2. Declarative deployment model of a common web application. Rectangles represent components of the application while arrows illustrate the dependencies between them. The types of the components are shown in braces while types of the relations are encoded by their line type.

C. Motivating Scenario and Problem Statement

Figure 2 illustrates a declarative deployment model describing a multi-cloud scenario based on EDMM [6]: A highly scalable user front-end, i. e., the *Order App*, is hosted on an AWS cloud service and communicates with a back-end processing service, i. e., the *Order Processor*, that is hosted on a private cloud of type OpenStack. Of course, the Order Processor could also be hosted on a public cloud for improved scalability. However, this scenario is selected to demonstrate the challenges of multi-cloud deployments using different cloud offerings as a service (XaaS). To ensure that the application conforms to the cloud-native properties [18], all data in our scenario is stored in a *Database*, which is depicted at the top right in Figure 2. Additionally, an *Order Queue* of type SQS Queue is utilized as messaging system to enable loose coupling between the Order App and the Order Processor component. While the Order App and the Order Queue are hosted on PaaS offerings, i. e., *Elastic Beanstalk* and *Simple Queue Service*, the Order Processor and the Database are hosted on *Ubuntu 18.04* VMs and require additional middleware to run, i. e., the *Java 8 Runtime* and the *MySQL DMBS 5.7* components. All components define properties expressing the target state or configuration that are set during deployment, e. g., the Order Queue defines its implementation type to be "`FIFO`". Each component also defines at least its *lifecycle operations* which encompass operations to create, configure, start, stop, and delete the component.

For such an application scenario, it is crucial to create copies of the current data in regular intervals. In case of the motivating scenario, respective AWS APIs must be utilized in order to backup and empty the Order Queue correctly. Further, to backup the Database in our scenario, either the backup functionality of the underlying DBMS must be utilized or a direct SQL connection to the database must be established, which requires correct credentials and a correct query to retrieve all data. Thus, to reproducibly backup stateful components, additional technology- and domain-specific logic is required that must be orchestrated and executed in the correct order.

In the domain of testing, it is not only important to ensure the technical success of an application deployment, but also to check the deployment success from a business perspective. In cases where applications get deployed across multiple clouds, components may need to communicate with each other. However, cloud providers typically apply different security settings by default, i. e., some open common ports for created VMs while others keep all ports closed. Furthermore, depending on the test, an SSH connection, HTTP connection, or even an SQL connection, which requires additional credentials, must be established. This requires immense expertise in each technology used to detect possible issues and to realize the tests. Therefore, the generation of automated tests in a reproducible manner is crucial every time an application is changed to verify that all components work as intended, configuration properties are set correctly, defined computing instances are running, and communication among components is established correctly.

Thus, even automating the management of such simple applications is a major challenge if the corresponding functionality must be implemented manually as such implementations are error-prone, time consuming, and require expert knowledge in the respective domains and used technologies. Moreover, even if this is possible for a company to create such management scripts, programs, or workflows manually, one change in the deployment model can obsolete them as the implementations do not match the deployment model anymore. This leads to major effort in refactoring or rewriting such management programs to reflect the changes from the deployment model. Therefore, we present an approach to automatically generate holistic management workflows from declarative deployment models.

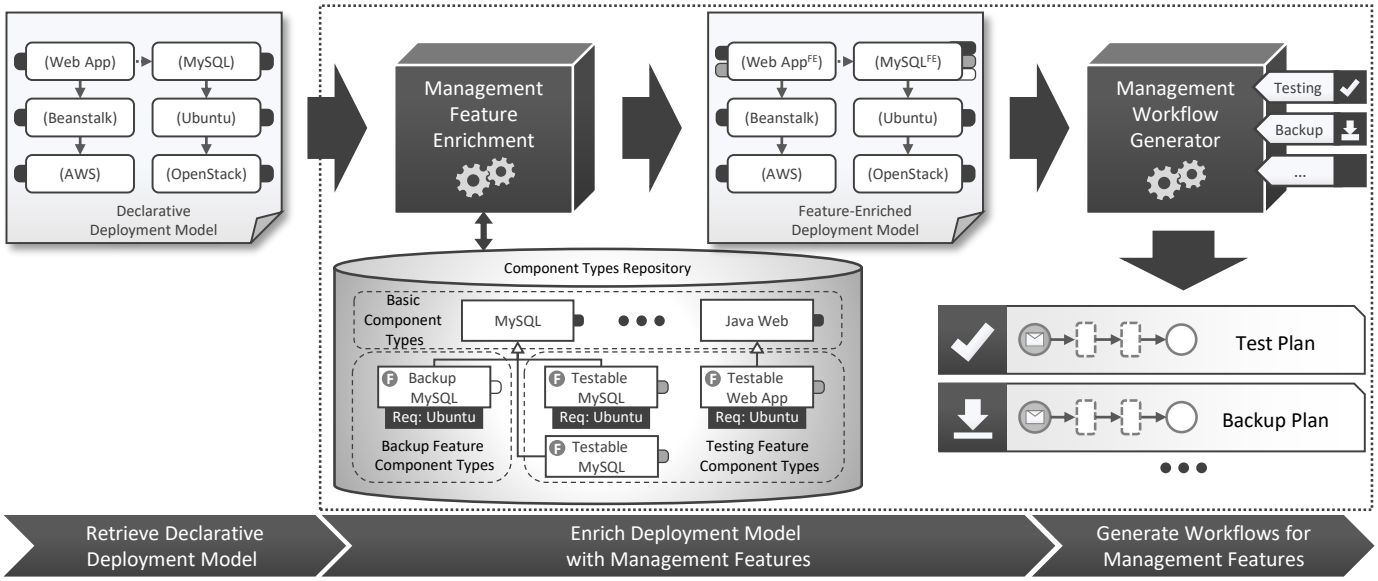


Figure 3. Overview of the approach: Automated Generation of Management Workflows for Applications Based on Deployment Models.

III. Approach

In this section, we introduce our approach to automatically generate management workflows for applications based on deployment models. The overall concept is described in Section III-A while the following sections outline the details.

A. Overview

To automatically derive executable management workflows for different kinds of functionalities from declarative deployment models of existing applications, we propose the management feature enrichment and workflow generation approach. An overview of the approach is shown in Figure 3:

(1) Retrieve Declarative Deployment Model: The input for our approach is a declarative deployment model as described in Section II-A. The declarative deployment model is modeled based on *Basic Component Types*. These Basic Component Types only define the lifecycle operations, i. e., the operations to create, configure, start, stop, and delete an application component of this type. The exemplary deployment model depicted in Figure 3 on the left expresses only the desired state of the application and does not define any management related operations. A deployment model that already contains management operations can, of course, also serve as input.

(2) Enrich Deployment Model with Management Features: The Component Types Repository contains all Basic Component Types and available *Feature Component Types*. Feature Component Types provide component-specific management operations for certain features, e. g., testing or backup, which are implemented and provided by domain experts. For instance, for the *Web App*, a *Testable Web App* Feature Component Type is available, while there is an additional *Backup* Feature Component Type available for *MySQL*. The Basic and Feature Component Types form Component Type Hierarchies which are explained in detail in Section III-B. The Component Type Repository is searched for available *Feature Component Types* that are applicable to the Basic Component Types used in the given model. For one feature of a certain Basic Component Type, several implementations of the management operation can be available: For instance, if a MySQL database exists in the given deployment model, an operation for testing can be implemented as a shell-script that is executed on a Linux environment or as an external web service. In the Component Types Repository in Figure 3, two *Testable MySQL* features are depicted. One of these defines an `Ubuntu` requirement. Hence, it requires the Database component to run on an Ubuntu VM, for example, to execute script-based operations. Thus, the *applicability* of a certain Feature Component Type is determined by the hosting components in the deployment model. In the example in Figure 3, the MySQL component is hosted on an Ubuntu VM and, thus, the requirement of the Feature Component Type is satisfied.

Before the features can be applied, a new component type providing all selected management operations is generated. A Basic Component Type is merged with the respective Feature Component Types to a *Feature-Enriched (FE) Component Type*. The merged component type for MySQL is called *MySQL^{FE}* providing lifecylce, testing, and backup operations. The newly generated FE Component Types, i. e., MySQL^{FE} and Web App^{FE}, are then applied to the deployment model by replacing the component types of the respective components, i. e., the component of type MySQL is updated to the MySQL^{FE} type. The result is a *Feature-Enriched (FE) Deployment Model*.

(3) Generate Workflows for Management Features: For executing the added management operations, imperative workflows are generated based on the operations in the FE Deployment Model using feature-specific plan generation plugins. These

plugins encompass the logic defining the order in which the operations must be executed. For example, a *Testing* plugin generates a management workflow that is able to execute all operations related to the Testing feature, i. e., all light grey operations in Figure 3. Hence, an executable *Test Plan* that runs all tests is created fully automatically. Similarly, the *Backup Plan*, as a result of the *Backup* plugin, contains the corresponding backup operations only, i. e., all white operations in Figure 3. Details of the plan generation, including the order of execution, of these two use cases are introduced in Section III-D. The resulting plans, one for each selected feature, can then be executed by a workflow engine depending on the underlying language. For example, BPEL workflows can be executed using the Apache ODE [19] workflow engine.

B. Component Type Hierarchies

For feature definition we distinguish two kinds of component types, namely *Basic Component Types* and *Feature Component Types*. Hereby, Basic Component Types specify the basic functionality to deploy the respective component, i. e., the lifecycle operations. Besides, Feature Component Types define component-specific management operations. Thus, a Feature Component Type extends the Basic Component Type, whereby a *Type Hierarchy* is formed. For example, to create a feature for Web Apps, the Feature Component Type must inherit from the Web App Basic Component Type and define new operations for implementing this new feature. In general, a feature declares an interface defining operations that have to be implemented by specific Feature Component Types respectively. For example, the *Backup* feature declares a single `backup()` operation, whereas the *Testing* feature defines that each operation prefixed with `test` is considered to be a testing related operation having a defined return value that indicates the success or failure. Although EDMM does not include inheritance, this concept does not affect the deployment model itself, as type hierarchies are only needed for feature selection.

C. Deployment Model Enrichment

To enrich a declarative deployment model with additional management functionalities, the set of available features must be determined first. Therefore, as described in Algorithm 1, the set of Feature Component Types FT are retrieved from a Component Types Repository and passed, alongside the deployment model $model$, to the deployment model enrichment algorithm. The enrichment is achieved in four major steps: First, for each component in the given model, all applicable features are determined (lines 2–10). Then, the applicable Feature Component Types must be selected by a user or custom logic since multiple Feature Component Types of the same kind, e. g., two Testing Feature Component Types as illustrated in Figure 3, could overwrite themselves (line 11). Third, the selected Feature Component Types $F^*_{c_i}$ that are applicable to a specific component c_i are combined with its Basic Component Type into an FE Component Type ct^{FE} (lines 12–15). Finally, the Component Type of each component in the deployment model is updated (line 16).

Algorithm 1 enrichDeploymentModel($model, FT$)

1: **for all** $\big(c_i \in \texttt{components}(model)\big)$ **do**
2: **let** F_{c_i} be the set of applicable $ft_j \in FT$ of c_i
3: **for all** $\big(ft_j \in FT\big)$ **do**
4: **if** $\big(\texttt{basicType}(c_i) == \texttt{supertype}(ft_j)\big)$ **then**
5: **let** $C^{successors}_{model}(c_i)$ set of hosted on successors of c_i
6: **if** $\big(\forall req_k \in \texttt{requirements}(ft_j) : \big(\exists c_m \in C^{successors}_{model}(c_i) : req_k = \texttt{type}(c_m)\big)\big)$ **then**
7: $F_{c_i} := F_{c_i} \cup \{ft_j\}$
8: **end if**
9: **end if**
10: **end for**
11: $F^*_{c_i} := \texttt{selectFeatures}(F_{c_i})$
12: **let** $ct^{FE} = (props_{ct^{FE}}, ops_{ct^{FE}}) := \texttt{type}(c_i)$
13: **for all** $\big(ft_j \in F^*_{c_i}\big)$ **do**
14: $ct^{FE} := (\{props_{ct^{FE}} \cup props_{ft_j}\}, \{ops_{ct^{FE}} \cup ops_{ft_j}\})$
15: **end for**
16: $\texttt{type}(c_i) := ct^{FE}$
17: **end for**

To determine the available features for a component in a given deployment model, the set of Feature Component Types is searched for applicable features: The applicability of a Feature Component Type is determined based on (i) its Component Type Hierarchy, as a Feature Component Type must inherit from a Basic Component Type, and (ii) its defined requirements, which must be satisfied by the deployment model. For example, as shown in Figure 3, the Testable MySQL Feature Component Types are both applicable to the MySQL Basic Component Type in the given deployment model as all requirements for both features can be fulfilled: While one does not define any requirements, the other one requires an `Ubuntu` VM, which can be satisfied in the given deployment model as the MySQL database is hosted on an Ubuntu VM. For this, it is first checked if the considered Feature Component Type ft_j inherits from the Basic Component Type of the component c_i (line 4). Second, it must be checked whether the requirements of ft_j are satisfied by the model. All components that are directly or transitively connected with c_i using a hostedOn relation are the set of its *successors* (line 5). If all required component types, stated by the requirements of ft_j, are contained in this set, the requirements are satisfied (line 6) and ft_j is an applicable Feature Component Type (line 7).

Afterwards, based on the determined applicable Feature Component Types for a certain component F_{c_i}, a user or a custom logic can select the desired Feature Component Types to be used for enrichment (line 11). This is required, since Feature Component Types of the same kind can overwrite themselves: For example, if both Testable MySQL Feature Component Types define the same test operations, invalid mappings between the operations and the implementing artifacts may arise during the merge of the different Feature Component Types.

In the third step, all selected Feature Component Types $F^*_{c_i}$ of a component c_i are combined into the FE Component Type. Each Component Type ct_k has a set of properties $props_{ct_k}$ and

a set of operations ops_{ct_k}. A new FE Component Type ct^{FE} is initially defined by the properties and operations of the type of the component c_i (line 12). For each Feature Component Type ft_j, the set of properties and operations are added to the corresponding sets of ct^{FE} (line 13-15). Lastly, the type of the component in the deployment model is exchanged with the FE Component Type to persistently attach all selected features to the respective component in the deployment model (line 16). Hence, the FE Deployment Model can be passed to the workflow generation to generate executable processes.

D. Workflow Generation

Our approach can be used to generate arbitrary management workflows fully automatically. Workflows are also referred to as *Plans* as they describe all activities that must be performed in a defined order. Hereby, we are following a similar approach as proposed by Breitenbücher et al. [7] who generate provisioning workflows based on declarative deployment models. However, instead of deriving the provisioning logic for the application itself, i. e., plans to deploy an application, we are creating management workflows for specific management functionalities. For example, to test the deployment, we are generating a *Test Plan* which executes only operations related to testing. Therefore, we combine and adapt existing approaches, such as application deployment testing [20] and backing up running cloud applications [21], to be used as independently executable features in the form of management workflows.

As depicted in Figure 3, our approach introduces a Management Workflow Generator component that requires feature-specific plugins to generate imperative workflows based on an FE Deployment Model. Such plugins encapsulate feature-specific logic on how the resulting imperative workflow is generated. For example, a plugin to generate a Test Plan encompasses the logic in which order test-related operations are executed, e. g., starting from the underlying infrastructure or platform component and following the *hostedOn* relationships in reverse direction until no other child component is found.

In general, each plugin first analyzes the FE Deployment Model and checks if it can work with the FE Component Types supplied within the model. If supported FE Component Types are found, the plugin starts to generate respective workflow fragments, e. g., BPEL [9] fragments, for each operation associated to the matching FE Component Types. According to the plugin's logic, the workflow fragments are ordered in the required sequence and the resulting workflow is stored and deployed to await its invocation. However, an operation can define multiple input parameters that are required in order to run it successfully. Each plugin must be able to satisfy such input parameters using matching property values of the respective component or from the underlying infrastructure or platform. On the one hand, a plugin must be able to write respective property values hard-coded to the workflow fragments, i. e., for configuration properties such as static port definitions. On the other hand, plugins must be able to generate according workflow fragments that are able to retrieve property values at runtime, e. g., an IP address of a deployed virtual machine.

IV. Validation

The presented concepts build upon the Essential Deployment Metamodel (EDMM) to provide a technology agnostic approach which is realizable in all technologies that can be mapped to EDMM, e. g., Terraform. To validate the practical feasibility of our concepts, we used the deployment modeling language TOSCA [12], [13] as it is vendor- and technology agnostic, ontologically extensible [22], and can be mapped to EDMM. Moreover, we implemented a prototype based on the OpenTOSCA ecosystem [14], [23].

A. Mapping EDMM to TOSCA

The *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [12], [13] is a modeling language to describe applications in a standardized, vendor- and technology-independent manner. Applications are described in declarative deployment models, so-called *Topology Templates* inside of *Service Templates*. Hereby, *Node Templates* and *Relationship Templates* represent Components and Relations following predefined semantics which are defined by *Node Types* and *Relationship Types*, i. e., Component Types and Relation Types, respectively. A Node Type specifies the available *Properties* of Node Templates that are instances of this type. For example, the Database component shown in Figure 2 is, in TOSCA jargon, a Node Template which is an instance of the Node Type *MySQL Database 5.7* that defines a property called *DB-Name*.

Similarly, a Node Type defines a set of *Operations* inside an *Interface* that can be executed on a Node Template instance. These operations can be mapped to Operations in EDMM. The TOSCA standard introduces a lifecycle interface that defines a *create*, *configure*, *start*, *stop*, and *terminate* operation [13]. These operations can be implemented using *Node Type Implementations* associating each operation with an *Implementation Artifact*, i. e., an Artifact in EDMM, that represents the actual executable, e. g., a shell script. In addition, *Requirements* can be defined for Node Types which express that, e. g., certain *Capabilities* or Node Templates of specific Node Types are expected in a Topology Template the Node Type is used in. A TOSCA type can inherit from another type to further improve the reusability. For example, the MySQL Database 5.7 Node Type can inherit from an abstract Database Node Type. In EDMM, inheritance and requirements of component types can be expressed by using properties.

All elements defined by the TOSCA standard, e. g., Service Templates, Node Types, or Relationship Types can be persisted in a repository enabling modelers to reuse them in multiple applications. In addition to the declarative deployment model, TOSCA also supports imperative models [12]. To describe these, workflow languages such as BPMN [8] or BPEL [9] can be used. Besides deploying an application, imperative models can also perform arbitrary management tasks.

B. System Architecture based on OpenTOSCA

To proof our concepts, we implemented a prototype based on the open-source TOSCA ecosystem OpenTOSCA [14], [23].

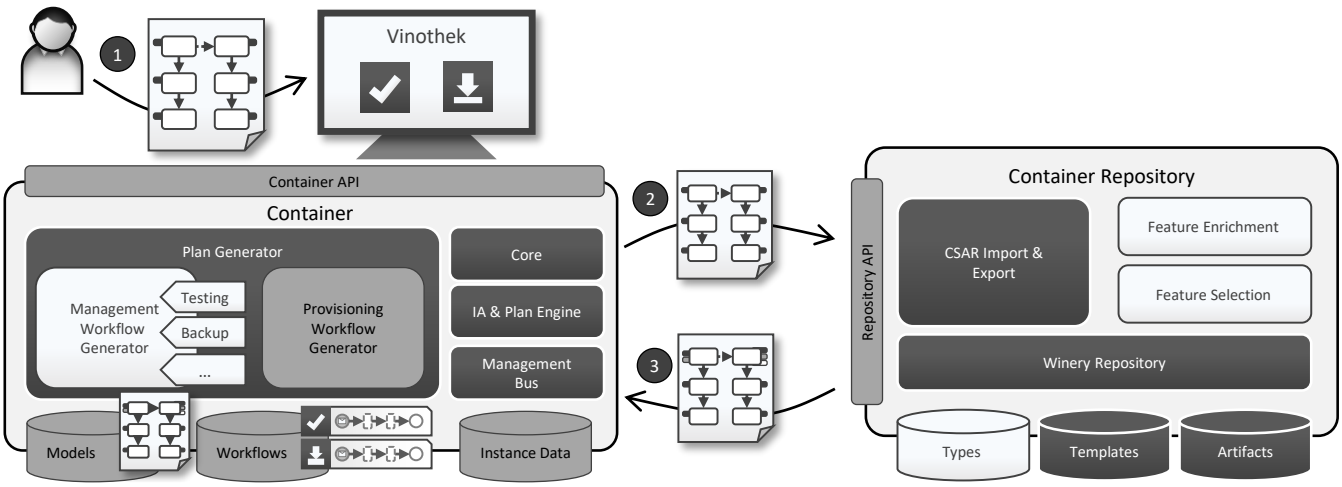


Figure 4. System architecture of the extended OpenTOSCA Ecosystem.

The OpenTOSCA ecosystem consists mainly of three components: Winery [24] which provides modeling functionality for applications modeled in TOSCA, the deployment system OpenTOSCA Container [25] or just Container for short, and a management UI called Vinothek [26]. The extended system architecture is depicted in Figure 4. Since our approach is based on an existing deployment model the modeling tool Winery is not explicitly illustrated. However, since the enrichment affects the topology model, the existing Winery capabilities to deal with topology model elements are utilized. Therefore, the *Container Repository* is based on the Winery.

The presented approach for enriching declarative deployment models with management operations used for an automated workflow generation (see Section III) is implemented in the OpenTOSCA ecosystem as depicted in Figure 4: First, an existing Cloud Service Archive (CSAR), the standard packaging format for TOSCA models, is uploaded to the Container using Vinothek. In the second step, the Container calls the repository API of the Container Repository to request the feature enrichment for the given CSAR.

In the Container Repository all TOSCA elements are stored and managed by the *Winery repository*. In the *Types* repository the Node Types defining the lifecycle operations, i. e., representing the Basic Component Types, as well as the Node Types specifying management operations, i. e., representing the Feature Component Types, are contained. We utilize the inheritance capability of TOSCA: The Feature Node Types inherit from the Basic Node Types and define management operations. The Feature Node Types can also specify *Feature Requirements*, which must be satisfied by the Topology Template the Feature Node Type shall be used in. For example, the Feature Requirement `Ubuntu`, or `Ubuntu_16.04` if a certain version is required, can be added to a Feature Node Type. This means, a Node Template of type Ubuntu must be contained in the hosting stack of the Node Template, encompassing all Node Templates that are directly or transitively connected by a hostedOn relation. The *Feature Selection* component determines for each Node Template contained in the given Topology Template the applicable Feature Node Types based on Node Types and Requirements matching. The *Feature Enrichment* component creates FE Node Types and replaces the original Node Types in the Topology Template by the FE Node Types.

In the third step, the enriched Topology Template serves as input for the *Plan Generator*. For this, the Topology Template is exported as CSAR from the Container Repository and imported to the Plan Generator and stored in the *Models* repository. The initial functionality of the Plan Generator to derive and generate provisioning workflows from declarative deployment models in a generic manner has been extended with the generation of management workflows. The *Management Workflow Generator* component enables to define plugin based abstract workflows for each feature which are used to generate concrete management workflows for a certain Topology Template. This is achieved by interpreting and analyzing the Topology Template and the Node Types of the contained Node Templates. Thus, a Plan, e. g., for testing the entire modeled application, can be generated. In our prototypical implementation, the resulting plans are executable BPEL workflows. Of course, BPMN workflows could also be generated using our approach.

All plans returned by the Plan Generator are stored in the *Workflows* repository within the Container. Afterwards, they can be triggered using Vinothek and are executed on a workflow engine like Apache ODE. To execute an Implementation Artifact implementing a management operation, a workflow calls the API of the Container which forwards the call using the *Management Bus*. If needed, the *IA Engine* can be used to deploy Implementation Artifacts, e.g., to run Web Services. Therefore, the user can select available features and execute the corresponding workflow at any time, e.g., if a current backup is needed. Thus, features can be provided in a reusable manner by implementing a management workflow generation plugin and the corresponding Feature Node Types.

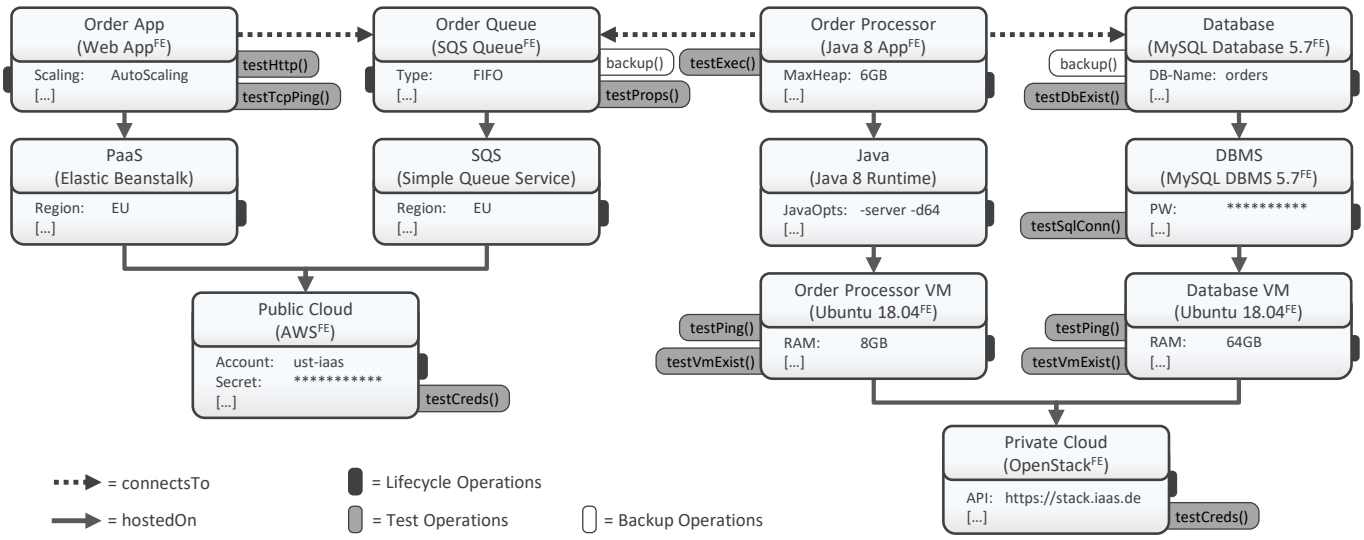


Figure 5. Feature Enriched Deployment Model.

V. Case Studies

Given the motivating scenario depicted in Figure 2, we discuss the application of two different management features, namely Testing, see Section V-A, and Backup, see Section V-B. By applying our approach, the initial types of the motivating scenario have been enriched with corresponding FE Component Types capable of testing or back up components. The added feature-specific operations are shown in Figure 5 by bars attached to the components, whereby grey bars identify test operations while white bars are used to show backup operations.

A. Testing Feature

The Testing feature defines an interface that follows the following contract: Each declared operation prefixed with `test` is considered to be testing related and must return predefined enumeration values indicating the success or failure. Based on this, arbitrary test operations can be provided and encapsulated into one or more Feature Component Types. For example, there is a `testTcpPing()` operation, as depicted in Figure 5, being able to test the availability of a *Web App*. Further, there is a `testHttp()` operation verifying that the application returns a suitable HTTP status code. In our case, these two operations are part of a *Web App Testing Feature Component Type*. Moreover, there are test operations to verify that the respective operating system process is running for *Java 8 App* components, to verify that related properties are set correctly for *SQS Queue* components, to verify that a SQL connection to *MySQL DBMS 5.7* components can be established, to verify that a configured database schema of *MySQL Database 5.7* components is present, to verify that *Ubuntu 18.04* components are reachable and computing instances are available, and to verify that respective cloud provider APIs can be utilized by checking supplied credentials. These operations are respectively provided by certain *Testing Feature Component Types* extending their Basic Component Types.

Our ecosystem (cf. Section IV) enables to enrich a given deployment model with FE Component Types and to generate executable test plans. The *Testing* plugin derives the execution order of the test operations based on the relations of type *hostedOn*: First, the tests of the target component of a hostedOn relation must complete successfully before the source component is tested. Thus, the Test Plan starts with executing all tests of components without outgoing hostedOn relations. For example, the tests annotated at the VMs run before the DBMS tests, but only after the OpenStack test completed successfully. The generated test plan can be executed at any point in time after the application is deployed.

B. Backup Feature

In addition to the Testing feature, the exemplary deployment model in Figure 2 has been enriched with back up functionality. The resulting FE Component Types with the respective backup operations are shown in Figure 5. Hereby, both stateful components, i. e., the Order Queue and the Database, have been enriched with *backup* operations enabling the retrieval of their internally stored states. To automatically perform a back up of all stateful components, a *Backup Plan* is generated. In contrast to the Test Plan generation, the *Backup* plugin creates a Backup Plan which executes all operations in parallel.

To backup a stateful component, its internal state must be retrieved. Hence, to create a backup of the Database component, an SQL connection to the MySQL Database must be established and an SQL query retrieving all tables and their contents is executed. Similarly, by establishing a connection to the Order Queue using the API of AWS, all stored requests can be retrieved. However, since a backup operation retrieves the state of a component, this state must also be saved persistently. Thus, the operations require a storage endpoint as an input. For example, in the OpenTOSCA ecosystem, Winery can be used to store the state artifacts in the context of the application [21].

C. Limitations and Benefits

The presented management feature enrichment and workflow generation approach significantly reduces the effort for creating management workflows for complex distributed applications as manual implementations are not needed. A deployment modeler can enrich an already existing deployment model with arbitrary management functionality by simply selecting features since they are implemented in a reusable manner. Additionally, the workflows are generated in a fully automated manner and, thus, manual implementations and adaptations for changing models are no longer required. However, to provide new management functionality, the workflow generation must be extended with a corresponding plugin which is capable of generating executable workflows based on specific operations. Additionally, domain experts have to implement the operations in new Feature Component Types. Although this requires immense expertise in the feature domain, after a feature plugin and the corresponding Feature Component Types are publicly available, all declarative deployment models that can be mapped to the EDMM can be automatically enriched with the new feature.

VI. Related Work

For managing deployed applications, cloud providers such as AWS provide operations, e. g., for scaling application components. However, these cloud provider specific functionalities are limited, i. e., they cannot be extended for custom management operations, and can only be used for single cloud deployments. Of course, configuration management technologies such as Chef and Infrastructure as Code technologies such as Terraform support the configuration of multi-cloud applications, but the management functionalities are limited. For example, Terraform enables to generate configuration and execution plans. However, these plans only support the general lifecycle interfaces to create, start, stop, and terminate application components. Complex custom management logic for specific management features cannot be covered.

Several research works also cover the generation of workflows based on declarative deployment models that transform the application into the desired state as described in the declarative model. This includes the generation of basic provisioning plans [27], [7], [28], [29], configuration changes [30], and also state-changing management functionalities such as migration as presented by Breitenbücher et al. [11], [16], [31], [32]. Eilam et al. [28] and Breitenbücher et al. both focus on generating workflows based on desired state models by using predefined elements, i. e., "automation signatures" [28] and "planlets" [11], [16] respectively, to create activities in a workflow. These predefined elements represent the operations and preconditions for executing these operations for application fragments. According to the desired state model, the operation execution order is generated. Breitenbücher et al. [11], [16], [31], [32] also use annotations attached to components in a declarative deployment model to specify the management functionality to be executed. Based on the predefined planlets and the annotations, workflows for holistic state-preserving management functionalities can be generated, but this approach requires a high amount of predefined planlets and annotations that are correctly attached to the declarative deployment model by the deployment modeler.

Mietzner and Breitenbücher et al. [7] first generate so-called Provisioning Order Graphs, which are transformed into executable workflows. Eiliam et al. and several other works [33], [34], [35], [36], [37] use planning techniques for deriving the required steps and execution order based on desired state models. However, they only focus on provisioning as the definition of a target state is not possible for state-preserving tasks. Furthermore, especially for management functionalities a feature-specific execution order of operations is important.

In previous works the generation of management workflows for specific management functionalities has already been presented [38], [20], [21]. Képes et al. [38] presented an approach to automate the generation of scaling plans for creating additional component instances on IoT devices. Wurster et al. [20] introduced a concept to define deployment tests directly along with the deployment model. The tests can then be executed by a test execution runtime. For saving and restoring the entire application state for applications that do not require to be "always on", Harzenetter et al. [21] introduce the Freeze and Defrost concept. For freezing and defrosting the application state, management plans are generated. The Backup feature case study presented in Section V-B is based on the freeze plan. We introduced a generic framework for providing management features for enriching declarative deployment models and to generate management workflows for arbitrary management functionalities.

VII. Conclusion and Future Work

In this paper, we introduced an approach enabling the automated generation of executable management workflows for arbitrary management functionalities based on declarative deployment models. For this, component-specific management operations, e. g., for testing the availability of a Web App, backing up a MySQL Database, extending the license of a Windows machine, or adding security updates, are provided as reusable entities, so-called Feature Component Types. These types are used for enriching the application components in a declarative deployment model with additional management functionalities. Based on the enriched deployment model, executable management workflows can be generated. We based our concept on a generic metamodel, the Essential Deployment Metamodel (EDMM), to be technology- and vendor-agnostic. For validating the feasibility of our approach we selected the standard modeling language TOSCA, which can be directly mapped to the EDMM, and extended the open-source ecosystem OpenTOSCA.

For further validations additional management features will be realized as Feature Component Types and corresponding plugins defining the execution order of the workflow tasks. Additionally, we also want to consider data-driven workflows and to extend the approach to enable a management across hybrid environments. Such hybrid environments often require distributed control of involved application components which demand the control of their own components.

Acknowledgment

This work was partially funded by the BMWi project *IC4F* (01MA17008G), the European Union's Horizon 2020 research and innovation project *RADON* (825040), the DFG project *SustainLife* (379522012), and the DFG's Excellence Initiative project *SimTech* (EXC 2075).

References

[1] D. Oppenheimer, "The importance of understanding distributed system configuration," in *Proceedings of the 2003 Conference on Human Factors in Computer Systems Workshop (CHI 2003)*. ACM, Apr. 2003.

[2] Chef Software Inc., "Chef Official Site," May 2019, last accessed August 5th, 2019. [Online]. Available: https://www.chef.io

[3] HashiCorp, "Terraform Official Site," May 2019, last accessed August 5th, 2019. [Online]. Available: https://www.terraform.io/

[4] Amazon. (2016, Feb.) AWS CloudFormation. Last accessed August 5th, 2019. [Online]. Available: https://aws.amazon.com/de/cloudformation/

[5] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, Feb. 2017, pp. 22–27.

[6] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies," *Software-Intensive Cyber-Physical Systems (SICS)*, 2019.

[7] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, Mar. 2014, pp. 87–96.

[8] OMG, *Business Process Model and Notation (BPMN) Version 2.0*, Object Management Group (OMG), 2011.

[9] OASIS, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[10] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013)*. Springer, Sep. 2013, pp. 130–148.

[11] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Pattern-based Runtime Management of Composite Cloud Applications," in *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*. SciTePress, May 2013, pp. 475–482.

[12] OASIS, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[13] ——, *TOSCA Simple Profile in YAML Version 1.2*, Organization for the Advancement of Structured Information Standards (OASIS), 2019.

[14] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, and M. Zimmermann, "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.

[15] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011.

[16] U. Breitenbücher, "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements," Dissertation, University of Stuttgart, Faculty 5, 2016.

[17] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications," in *Proceedings of the 4th International Workshop on the Business Process Model and Notation (BPMN 2012)*. Springer, Sep. 2012, pp. 38–52.

[18] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Jan. 2014.

[19] A. S. Foundation. (2018). [Online]. Available: https://ode.apache.org/

[20] M. Wurster, U. Breitenbücher, O. Kopp, and F. Leymann, "Modeling and Automated Execution of Application Deployment Tests," in *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, 2018, pp. 171–180.

[21] L. Harzenetter, U. Breitenbücher, K. Képes, and F. Leymann, "Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications," *Software-Intensive Cyber-Physical Systems (SICS)*, 2019.

[22] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, and G. Kappel, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–38, February 2018.

[23] University of Stuttgart. (2019) OpenTOSCA – Open Source TOSCA Ecosystem. [Online]. Available: https://github.com/OpenTOSCA

[24] O. Kopp, T. Binz, U. Breitenbücher, and 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.

[25] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and 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.

[26] U. Breitenbücher, T. Binz, O. Kopp, and 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, Demonstration, pp. 69–72.

[27] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A Generic Configurable Customizable Composite Cloud Application Framework," in *On the Move to Meaningful Internet Systems: OTM 2009 (CoopIS 2009)*. Springer, Nov. 2009, pp. 357–364.

[28] T. Eilam, M. Elder, A. V. Konstantinou, and E. Snible, "Pattern-based Composite Application Deployment," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*. IEEE, May 2011, pp. 217–224.

[29] S. Herden, A. Zwanziger, and P. Robinson, "Declarative Application Deployment and Change Management," in *Proceedings of the 2010 International Conference on Network and Service Management (CNSM 2010)*. IEEE, Oct. 2010, pp. 126–133.

[30] A. Brown and A. Keller, "A Best Practice Approach for Automating IT Management Processes," in *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*. IEEE, Apr. 2006, pp. 33–44.

[31] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and 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.

[32] U. Breitenbücher, T. Binz, O. Kopp, and 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.

[33] T. Kuroda, M. Nakanoya, A. Kitano, and A. Gokhale, "The configuration-oriented planning for fully declarative it system provisioning automation," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 808–811.

[34] T. Kuroda and A. Gokhale, "Model-Based IT Change Management for Large System Definitions with State-Related Dependencies," in *Proceedings of the 18th International Conference on Enterprise Distributed Object Computing (EDOC 2014)*. IEEE, Sep. 2014, pp. 170–179.

[35] I. Georgievski, F. Nizamic, A. Lazovik, and M. Aiello, "Cloud ready applications composed via htn planning," in *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, Nov 2017, pp. 81–89.

[36] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. Konstantinou, "Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools," in *Proceedings of the 7th International Middleware Conference (Middleware 2006)*. Springer, Nov. 2006, pp. 404–423.

[37] H. Herry, P. Anderson, and G. Wickler, "Automated Planning for Configuration Changes," in *Proceedings of the 25th International Conference on Large Installation System Administration (LISA 2011)*. USENIX, Dec. 2011, pp. 57–68.

[38] K. Képes, U. Breitenbücher, and F. Leymann, "Integrating IoT Devices Based on Automatically Generated ScaleOut Plans," in *2017 IEEE 10th Conference on Service-Oriented Computing and Applications, SOCA 2017, 22-25 November 2017, Kanazawa, Japan*. IEEE Computer Society, Nov. 2017, pp. 155–163.