SN Computer Science (2023) 4:370
https://doi.org/10.1007/s42979-023-01810-4

ORIGINAL RESEARCH

An Integrated Management System for Composed Applications Deployed by Different Deployment Automation Technologies

Lukas Harzenetter[1] · Uwe Breitenbücher[1] · Tobias Binz[2] · Frank Leymann[1]

Received: 10 January 2023 / Accepted: 20 March 2023 / Published online: 29 April 2023
© The Author(s) 2023

Abstract
Automation is the key to enable an efficient, fast, and reliable deployment of applications. Therefore, several deployment automation technologies emerged in recent years whereby each technology has its specific field of application: While some are bound to cloud providers and offer provider-specific functionalities, others enable multi-cloud deployments but mostly do not support provider-specific features. As a consequence, often companies have to use multiple deployment technologies in combination to deploy large applications. However, the management capabilities of most deployment technologies are limited or even non-existent. This issue becomes even more severe if different parts of a single application are deployed by different technologies. To tackle this issue, we present an approach that enables generating automatically executable management workflows for applications that consist of multiple components deployed by different deployment technologies. Our approach builds on top of instance models that are automatically generated based on information retrieved from the different deployment technologies involved. Based on the derived instance model, we generate workflows that manipulate the running application. We prove the technical feasibility by an open-source prototype and discuss a detailed case study.

Keywords Application management · Management automation · Workflows · TOSCA · OpenTOSCA

Introduction

Automatically deploying applications is crucial to fully benefit from the dynamic nature of cloud architectures as manual deployment is error-prone and time-consuming [38, 46]. Hence, *deployment automation technologies*[1], such as Terraform [32], Kubernetes [51], Chef [48], Puppet [49], or Ansible [50], arose that are capable of automatically installing, configuring, and starting software components. However, modern applications often consist of multiple different software components and running services that are composed to provide the application's functionality—often, these components even run on different infrastructures operated by different providers, e.g., in Multi-Cloud Applications [47]. Hence, because of the resulting immense complexity of such applications, companies often need to combine multiple deployment technologies to instantiate different parts of a single application [20, 28]. For example, if a hybrid cloud application consists of a public part and a private part, the public part may be deployed using AWS Cloud Formation [1], while the private part might be deployed using Terraform, Chef, or a combination of them: For example, as Terraform is dedicated to provision infrastructures, it may be used to provision virtual machines in a private cloud, while Chef may be used to install middleware and business components onto these infrastructure components. As a result, each part of an application could be deployed by different technologies and a single big picture of the application is missing.

This article is part of the topical collection "Advances on Cloud Computing and Services Science" guest edited by Donald F. Ferguson, Claus Pahl and Maarten van Steen.

✉ Lukas Harzenetter
harzenetter@iaas.uni-stuttgart.de

[1] University of Stuttgart, Institute of Architecture of Application Systems (IAAS), Universitätsstraße 38, 70569 Stuttgart, Germany

[2] Robert Bosch GmbH, IoT & Digitalization Architecture, 70442 Stuttgart, Germany

[1] In this paper, we refer to all kinds of technologies that can be used to deploy applications as "deployment automation technologies"—we use "deployment technologies" as a short form. Thus, this term also includes configuration management technologies such as Chef or Puppet, infrastructure management technologies such as Terraform, and also container orchestration technologies such as Kubernetes.

While there are approaches to combine different deployment technologies for such scenarios, e.g., as presented by Wurster et al. [55], the subsequent management of running applications is still a major problem: While some deployment technologies offer management capabilities for single components, for example, scaling the number of virtual machines (VMs), performing management tasks that are affecting multiple components at once are mostly not supported—especially not if the different affected parts of a management task are deployed by different technologies. We refer to such management tasks that affect several components at once as *holistic management processes*. For example, a holistic management process is to install security updates for all VMs running in multiple clouds or to perform a migration of stateful components to another cloud. Thus, to enable such holistic management processes, often different deployment and management technologies for the different environments of the application must be combined, e.g., by implementing a script, which is a manual task for operations personnel.

To tackle this issue, we previously introduced an approach which enables the generation of imperative management workflows for holistic management processes before the application is deployed, i.e., during *design time* [31]. The approach enables the enrichment of deployment models with management processes that have not been modeled manually. In the conference paper [29] of this journal extension, we reused this approach [31] to enable the enrichment of holistic management functionalities to already *running* applications by generating management workflows based on an automatically generated instance model of an application. However, this approach is only applicable if merely one deployment technology is used to deploy the entire application. Hence, it is still an open challenge to execute holistic management processes on *running applications that are composed of multiple components managed by different deployment technologies*. Therefore, in this paper, we extend our *Managing Running Applications by Generating Workflows* approach [29] by a concept that also supports running applications that are deployed by multiple deployment technologies in combination, which poses additional challenges: (i) An instance model of the entire application must first be derived describing all components of the application. Therefore, all employed deployment technologies must be queried for runtime information about the application to create models for each part of the application which then must be merged into one holistic instance model. To solve this, we derive a normalized and standardized instance model of the running application based on the *Topology Orchestration Specification for Cloud Applications (TOSCA)* [7, 43, 44]. (ii) To perform state-changing[2] management processes, the deployment technologies must be notified about the changes, since many technologies monitor the applications and might revert performed state changes. Therefore, we enrich the components of the running composed application based on the derived instance model with additional management features while considering the underlying deployment technologies managing the corresponding components. (iii) Moreover, the dependencies between the applications' components and their corresponding deployment technologies must be maintained to enable their management. Hence, the derived instance model must also contain the information about which component is managed by which deployment technology as well as how the deployment technologies can be accessed. Therefore, we generate automatically executable management workflows to execute the enriched management functionalities that are aware of the underlying deployment technologies. Thus, in this paper, we are answering the following research question (RQ):

> "*How can running applications that are composed of multiple components deployed by different deployment technologies be enriched with additional, holistic management functionalities which are not supported by the employed deployment technologies?*"

To prove the practical feasibility of our approach, we introduce a prototypical implementation based on TOSCA and the OpenTOSCA ecosystem [12]. This prototype extends our previous works [29, 31] to support managing applications that are deployed by multiple deployment technologies. Moreover, we present a detailed case study based on the *Sock Shop*[3]: A microservice demo application that demonstrates how different services can work together as one large composed application. We first deployed the services using three different deployment technologies to demonstrate and explain how our presented approach can be applied after deployment to manage the running instance by including the different deployment technologies in the execution of the generated management workflow.

Related Work and Fundamentals

In this section, we introduce fundamentals and outline research challenges based on related work.

[2] State-changing processes or operations are not only interacting with the component, but are also altering it, e.g., changing its configuration or installing a new version [9].

[3] https://microservices-demo.github.io/.

Deployment Models and Deployment Automation

The manual deployment of applications is cumbersome, error-prone, and time-consuming [46]. In addition, since Cloud Computing offers IT resources in the form of on-demand services, it enables dynamic provisioning and decommissioning of applications [37]. Therefore, automating the deployment of applications became very important and many different deployment automation technologies arose [59]. These technologies mostly use *deployment models* to automatically provision and configure the modeled application. Hereby, two types of deployment models can be differentiated: *imperative deployment models* and *declarative deployment models* [23]. While imperative models exactly define *how* a deployment is performed in terms of the activities that need to be executed, i.e., as an executable process, declarative models only describe *what* has to be deployed, i.e., the structure of the application in the form of its components, their relations, and configurations—usually in the form of a graph [59]. Thus, to describe an imperative deployment model, the modeler must exactly define the tasks that need to be performed, e.g., service invocations and script executions, as well as the control and data flows by creating an executable script or workflow. Although imperative models enable modelers to realize flexible and arbitrary custom processes, immense technical expertise is required to create such imperative deployment models. Moreover, creating them manually is typically a very time-consuming and—due to the technical complexity—an error-prone task [13].

On the other hand, declarative deployment models are generally easier to create [13] and can even be automatically transformed to imperative models, which has been shown by several works, e.g., [9, 10, 13, 31, 35, 53]. As a result, declarative deployment technologies have prevailed in research and industry: The 13 most used deployment automation technologies support them [59]. Thus, we are following this trend and also focus on declarative deployment modeling in this paper.

In general, regardless of the modeling style, i.e., imperative or declarative, two kinds of directed deployment relations between components can be differentiated, namely *horizontal relations* and *vertical relations* [58]. Hereby, horizontal relations define that a component *connects to* another component. For example, a software component needs to retrieve data from a database. Hence, it creates a connection to the database which is also referred to as a horizontal relation. On the other hand, the software component, as well as the database, are *hosted on*, e.g., a VM component. Thus, the relationship between the software component and the VM is vertical, since it is running on top of the VM.

Application Management

During the whole lifecycle of an application, it passes multiple management stages in which different operations and processes must be performed [18]: In the first stage, the application must be *provisioned*, whereby all its components are deployed, i.e., installed and configured. As a result, the application is running and can be accessed by its users. However, during the runtime of the application, it must be managed and maintained, as components of the application may crash because of invalid inputs or environmental failures. Thus, to ensure that the application operates correctly and to avoid data loss, backups and other management tasks must be performed. Hence, the application is in the *management stage*. Finally, if the application is not needed anymore, it is in the so-called *decommission stage* in which all its components are stopped and uninstalled whereby all resources are ultimately freed.

Moreover, two kinds of operations can be differentiated during the management of an application's lifecycle: *state-changing* operations and *state-preserving* operations [9]. While sate-changing operations are changing the state or configuration of an application component, e.g., opening a certain port of a VM, state-preserving operations are only interacting with a component and do not change its state or configuration. For example, an operation that creates a backup of a database in an application is referred to as a state-preserving operation as it only accesses the database and does not change it or its configuration.

The TOSCA Standard

TOSCA [43, 44] is a language standardized by OASIS to describe the deployment and management of cloud applications. It supports declarative and imperative deployment models as it enables the declarative description of an application's components and relations, as well as the specification of imperative workflows to deploy and manage them. The workflows in TOSCA are called *Management Plans* and can be realized using workflow languages, such as BPEL [42] and BPMN [45], or using their own workflow definition language [44].

To describe an application in TOSCA, so-called *Service Templates* are used. Within a Service Template, the *Topology Template* defines the structure of the modeled application in the from of a directed and weighted graph: The nodes of this graph are called *Node Templates*, representing the application's components, while the edges, and thus their relations, are called *Relationship Templates*. In TOSCA, Node Templates and Relationship Templates are semantically defined by their type, which makes it ontologically extensible [3]. Thus, *Node Types* and *Relationship Types* are used to define *Properties*, *Interfaces*, and the corresponding

Operations of their instances, i.e., Node Templates and Relationship Templates, respectively. To avoid repeatedly defining, e.g., common Interfaces or Properties, TOSCA introduces inheritance between types. Thus, a Node Type defining, e.g., a Ubuntu VM, may inherit from an abstract VM Node Type which may already define a "Public-IP" address property. Additionally, TOSCA uses the concept of *namespaces* to uniquely identify and group definitions. For example, the set of infrastructure Node Types, such as VM types like Ubuntu or Windows, can be defined in one namespace, while another defines a set of webservers like Nginx or Tomcat. Finally, TOSCA defines the *Cloud Service Archive* (CSAR): A package format that contains all required TOSCA definitions, i.e., the Service Template and all referenced definitions such as Node Types or Relationship Types, as well as all necessary scripts and executables required to deploy the modeled application.

Related Work and Research Challenges

Our main goal of this work is to enable executing management processes for running applications composed of multiple components which are deployed by different deployment technologies. In general, there are several works dealing with the management of (cloud) applications ranging from basic application provisioning [10, 22, 33, 41, 56, 57, 60] to state-changing management functionalities such as updating the configuration of application components [16, 19, 29–31]. Moreover, several works exist that are generating workflows in languages such as BPEL or BPMN to deploy and manage applications [9–11, 17, 21, 22]. However, while workflows ensure reproducibility of the management processes, none of the approaches explicitly consider the underlying deployment technologies that typically monitor the application. Hence, if a management workflow changes the application's state without notifying the used deployment technology, the state change may be reverted by its deployment technologies, as they usually try to keep the application in a predefined state. Thus, these approaches only work for applications that are deployed manually or by technologies that do not monitor the application. However, many deployment technologies also monitor the application and continuously enforce a certain state of the application. As a result, when performing state-changing management processes, the underlying deployment technologies must be invoked to perform the state change; otherwise, it might be reverted by them. Therefore, in our work [29], which is extended by this journal, we tackled the research question *how state-changing management operations can be performed on running applications without interfering with the underlying deployment technology*. As we motivated above, modern applications are often composed of multiple components that need to be deployed by different deployment technologies. However, our previous approach is not sufficient to tackle this issue as it only supports one single deployment technology that is used to deploy and manage the entire application. Thus, we are tackling the following research challenge (RC 1):

> "*How can state-changing management operations be performed on running applications that are composed of multiple components which are deployed by different deployment technologies while avoiding that they revert performed changes?*"

The approach we present in this paper is based on instance models which we use to enable the holistic management of an application. To retrieve the current state of an application, i.e., an *instance model* of the application, several approaches exist that have been developed in different research areas. Brogi et al. [14] discover different cloud services that can be used afterwards to model applications. Holm et al. [34] are scanning the network traffic to identify active components of an application. Other approaches, such as Binz et al. [6], Farwick et al. [25], Fittkau et al. [26], Machiraju et al. [39], and Menzel et al. [40] aim for more details and try to identify every component of an application and their corresponding configuration. Hereby, these approaches focus on retrieving and deriving application components and their configurations using dedicated software, such as network scanners, crawlers, or dynamic analysis. Moreover, there are also approaches that enable generating topology models from infrastructure as code (IaC) artifacts such as Chef cookbooks [24, 52]. However, these approaches require the original IaC artifacts to generate the topology models and do not consider instance information, which is the basis in our approach to manage running applications. Other approaches that aim at retrieving the current state of applications and represent them in instance models are available in the models@run.time community [2, 8]. However, many of these approaches from the models@run.time community require an a-priori-model of the application that is enriched with additional details. Thus, runtime inference is still an open research area [2]. Especially, if an application is composed of components managed by different deployment technologies, instance information is spread across several technologies and must be integrated. Hence, the second challenge (RC 2) we are tackling is:

> "*How can a single instance model of a running application that is composed of multiple components deployed by different deployment technologies be retrieved and represented in a normalized and standardized fashion?*"

To enable the execution of management functionalities during runtime, many approaches require a deployment model of the application which also specifies management operations and processes. For example, the TOSCA

standard [43, 44] supports attaching management operations to Node Types that can be used during runtime to execute management functionalities on components. Even executable management workflows (Management Plans) can be specified and executed during runtime to manage large parts of the application consisting of multiple components. Such Management Plans typically invoke the management operations provided by the Node Types to implement a higher level management functionality. Similarly, event-driven approaches as, for example, presented by Brogi et al. [15] have been introduced: In the *Manage Applications Running in Opportunistic fog scenarios (MARIO)* approach, the authors focus on manually modeling management operations declaratively as *policies* annotated to nodes within a network [15]. Such policies are formulated in PROLOG and are periodically evaluated by the proposed framework whether a particular management operation, such as undeploy, migrate, or replicate, must be executed. However, while Brogi et al. also manage running applications, they do not derive instance models of applications and rely on "*minimal monitoring runtime information on the infrastructure*" [15] from monitoring tools to trigger the management operations described in the annotated policies of applications. In contrast, in our work, we want to enable the management of running composed applications without the need for dedicated monitoring tools. Moreover, since management operations in TOSCA and in the approach by Brogi et al. [15] must be modeled manually, we are automatically searching for management operations and make them executable by automatically generated management workflows for each management feature.

We did a first step towards the goal of managing applications in a previous work [31] and introduced an approach to enable the management of applications that have not been deployed yet, i.e., enrich management functionalities at *design time*. Hereby, the components of the modeled application are investigated whether there are additional management operations available that can be enriched to them. For example, if there are test operations available for a Tomcat webserver, e.g., an operation that checks if the Tomcat replies to HTTP requests on port 80, and a MySQL database, e.g., an operation that checks if it can be accessed only from within the local network, they can be used to enrich such components with the corresponding test functionality. However, this approach only works for applications that have not been deployed yet and, thus, is only applicable during design time. Therefore, we extended the approach in the conference paper [29], which is the basis for this journal extension, to support *running applications*. However, even the extended *runtime* approach does not support composed applications that are deployed using *multiple deployment technologies*. Hence, the third research challenge (RC 3) we tackle is:

"*How can management functionalities be executed for a running composed application that are not supported by the employed deployment technologies without the need to implement these operations for each application separately?*"

An Integrated Management System for Composed Applications Deployed by Different Deployment Automation Technologies

In the following, we present our new concept to enable holistic management processes for *running* and *composed* applications that have been deployed by multiple deployment technologies. Next, we describe an overview of the approach, which is illustrated in Fig. 1, before presenting the details in the following subsections.

In the first step, see ❶ in Fig. 1, the so-called *Instance Information Retriever* component is used to retrieve the runtime information about a running application from its underlying deployment technologies (see "Instance Information Retriever"). The retrieved information are passed to the *Instance Model Normalizer* component which is interpreting the deployment technology-specific data and generates a standardized and normalized instance model of the application based on the TOSCA standard in the step ❷ (see "Instance Model Normalizer"). Afterwards, the model is passed to ❸, the *Instance Model Completer*: A plugin-based component which iteratively executes several component-specific plugins that are able to identify more details about the running application, e.g., a *MySQL plugin* may be able to identify a running MySQL database. This is required, since the deployment technologies may not hold information about all components [29] (see "Instance Model Completer"). In step ❹, the model is enriched with management functionalities using the *Instance Model Enricher* component. Hereby, our previous *Management Feature Enrichment and Workflow Generation* approach [31] is extended and adapted to support instance models as well as applications that have been deployed using multiple deployment technologies (see "Instance Model Enricher"). Hence, in step ❺, the enriched model is passed to the *Management Workflow Generator* component which generates management workflows for each enriched management functionality (see "Management Workflow Generator"). Finally, to manage the application, the generated workflows can be executed on a corresponding *Workflow Engine* in step ❻ (see "Workflow Engine").

Instance Information Retriever

To derive instance models of running applications, we designed a plugin-based *Instance Information Retriever*

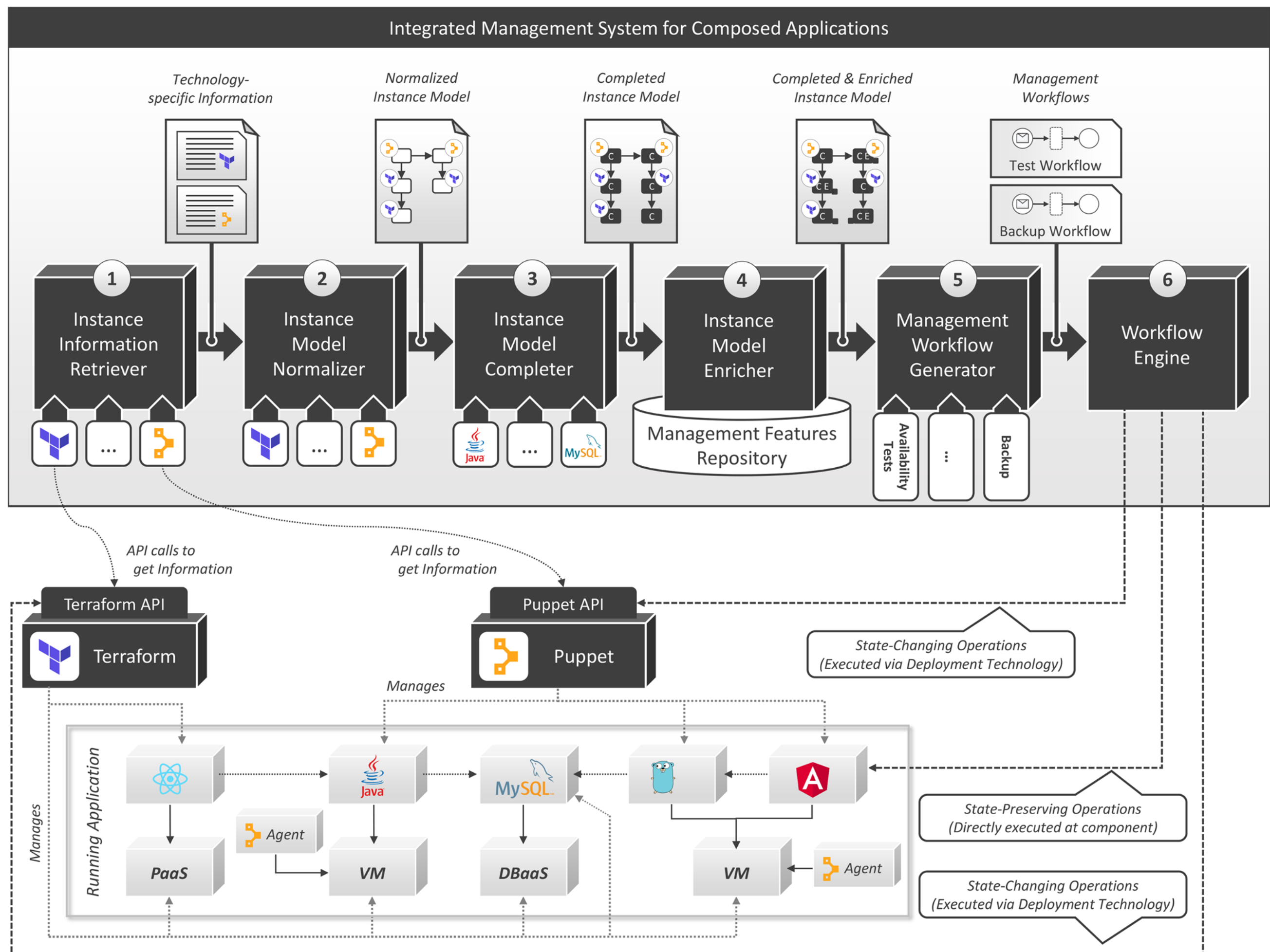

Fig. 1 Overview of our new integrated management system for composed applications method

component. It retrieves the deployment technology-specific information about the running application from the underlying deployment technologies using their application programming interfaces (APIs). In contrast to existing work, such as network scanning [34], we explicitly use the deployment technologies to derive instance models of running applications as they are required to perform state-changing management functionalities. Therefore, to avoid their interference after performing changes to the application, we annotate the information how each used deployment technology can be accessed to the derived instance model.

However, depending on the used deployment technology, the granularity level of the information that can be retrieved varies: While Terraform and Kubernetes, for example, only maintain information about the currently running infrastructure, such as VMs or containers, Chef and Puppet provide more detailed information about the concrete software and middleware components as they are specialized in managing the lifecycle and configuration of these components [59]. Although Terraform can be used to also execute arbitrary scripts to install, e.g., a webserver on a VM, it does not hold any runtime information about such components. Thus, we generate a normalized and standardized instance model of the application based on TOSCA, as it is also suitable to represent instance models [4]. Nevertheless, the technology-specific information must first be interpreted and represented in a normalized and standardized format for further processing. Hence, we extended the Instance Information Retriever in this paper by additional plugins to support the retrieval of instance information from Terraform and Kubernetes.

Instance Model Normalizer

After the deployment technology-specific instance information about the running application has been retrieved by the Instance Information Retriever, the data are passed to the *Instance Model Normalizer*. Because the data retrieved from the deployment technologies are technology-specific, custom logic is required to interpret it and derive a normalized and standardized instance model. Hence, we designed

a plugin architecture as it i) enables technology-specific logic to be encapsulated and separated from others, and ii) facilitates the extension of new technologies. Additionally, the instance models must contain information about the underlying deployment technology, such as how to access the technology and technology-specific IDs to uniquely identify the components inside the deployment technology. Therefore, we designed the Instance Model Normalizer as a plugin-based component whereby each plugin represents a deployment technology and, thus, must be able to process the corresponding instance information from the represented deployment technology.

Since the TOSCA standard is a vendor and technology independent modeling language, and it is ontologically extensible (see "The TOSCA Standard"), we use it to also describe instance models. Thus, identified components are mapped to Node Templates in a Topology Template, while their dependencies are represented as Relationship Templates. For example, if an Nginx webserver is found to be running on a Ubuntu operating system, the components would be mapped to a Node Template that instantiates the standardized Nginx Node Type, and it would be connected with another Node Template of type Ubuntu using a Relationship Template that is an instance of the *hostedOn* Relationship Type. The normalized types are hereby defined in a *TOSCA Repository* and are used to specify the semantics, i.e., the properties as well as available interfaces of the component, in a standardized way. Hence, each plugin in the Instance Model Normalizer must be able to process the retrieved instance information, i.e., (i) identify deployed components, (ii) detect their types and map them to corresponding Node Types, as well as (iii) fill the properties defined by the Node Type and assign the current values in the generated Node Template [29].

To support applications that have been deployed using multiple deployment technologies, the Instance Model Normalizer must also be able to merge multiple instance models that are generated by the plugins. To achieve this, components that occur in two instance models must be identified and merged. Thereby, a similarity check is required based on the properties of the corresponding Node Template, which may differ for various Node Types. For example, to detect that two Node Templates in two different models represent the same VM instance, their public IP address can be used as a unique property. In contrast, to detect, e.g., that two deployment technologies establish a connection to the same database instance, its type and the location it is running in can be used. Hereby, we assume that a component is managed by one deployment technology and others only depend on the running instance: If a deployment technology manages a component, it is annotated with an annotation identifying it to be *managed by* this technology. Otherwise, it is annotated to be *identified by* the corresponding deployment technology. As a result, the Instance Model Normalizer outputs a *Normalized Instance Model* that conforms to the TOSCA standard. Thus, we resolved RC 2 as we now are able to generate normalized instance models of the application that are conforming to the TOSCA standard. Additionally, we created the foundation to solve RC 1, since we added the information about how to access the deployment technologies managing the components. Hence, we extended the Instance Model Normalizer in this paper to be able to combine multiple technology-specific instance models into a single TOSCA-based instance model of the entire application.

Instance Model Completer

In the third step, the *Instance Model Completer* component interprets the Normalized Instance Model and iteratively completes the model with additional information as the retrieved models from the deployment technologies may not contain all information. For example, the Instance Model Completer is able to identify hidden components, refine the types of already identified components, and fill missing property values, and is able to detect horizontal relations between components. This step is required, because the deployment technologies may not hold all information that are required to manage the application [29]. To achieve this, we are reusing the iterative and also plugin-based approach presented by Binz et al. [6] to detect, e.g., specific component versions, property values, and horizontal relations, such as connections to databases or queues [29]. Thereby, the plugins are component-specific. For example, while a Tomcat plugin may be able to detect the concrete version that is installed and on which Port a particular application is listening to, a second plugin may be able to detect that a Java application is connecting to a database and where this database is located. Hence, the second plugin is able to derive a horizontal relation of type *connectsTo* between a Java application and an identified database component. However, since the previous approach of the instance retriever plugins by Binz et al. [6] did not consider the underlying deployment technologies, we extended the approach in the conference paper [29] accordingly and execute the plugins until no more plugins can be found that are capable of identifying more details of the application.

To identify plugins that are able to detect more details about the application, we re-implemented the concepts presented by Binz et al. [6] and extended them with a sub-graph matching mechanism. Thus, each plugin may define multiple *Detectors*, i.e., graphs defining the component constellations they can refine, to add more details to the application, such as missing properties or refining types. As a result, plugins that define matching detectors are run in a loop until no more plugins can be applied to the instance model to gain more details about the running application. For example, a Tomcat

plugin may be able to (i) refine an abstract *Webserver* Node Type [44] to a concrete *Tomcat* Node Type, (ii) identify the concrete version of a Tomcat webserver, and (iii) detect the context path and port of a web application that is running on it. Therefore, the plugin would specify three detectors whereby (i) one simply contains a Node Template of type *Webserver*, (ii) a second defines a Node Template of type *Tomcat* without any version identifier, while (iii) the last one would define an abstract Web Application Node Template that is hosted on a Tomcat webserver. As a result, after no more plugin can be found that defines a detector matching any sub-graphs in the current instance model, the *Completed Instance Model* contains all necessary information to enrich and perform management tasks. In this paper, we added additional plugins for the *Instance Model Completer* to identify, e.g., properties of MongoDBs and Docker Containers.

Instance Model Enricher

To enrich the Completed Instance Model with additional management features, the *Instance Model Enricher* is used in the fourth step (see Fig. 1). We hereby extended our previously introduced approach [31] in the conference paper [29] to (i) consider the annotated deployment technologies and (ii) differentiate between *state-changing* and *state-preserving* management operations. While-state-preserving operations can be enriched to all supported components, state-preserving operations must consider the underlying deployment technology to avoid its interference. For example, a state-preserving functionality is the execution of tests whether the components run as expected. Thus, operations implementing such functionality can always be enriched to a component, no matter how it was deployed, since the operation only interacts with the component and does not change its state. On the other hand, performing, e.g., an update of a component, changes its state. The underlying deployment technology may detect this change and revert it during its next actions. Therefore, the corresponding implementations must communicate with the deployment technology to notify it about the performed changes. This is achieved by performing calls to the APIs of the deployment technologies when executing a state-changing operation.

To realize the feature enrichment of instance models that have been deployed using multiple deployment models, the selection of so-called *Feature Node Types* requires a refinement. In general, the enrichment of additional management features to a model of an application exploits the inheritance concept of Node Types [31]. This is illustrated in Fig. 2 which illustrates the contents of a *Management Features Repository*. Hereby, Feature Node Types are inheriting from *Normalized Node Types*, such as a MySQL Node Type or a Ubuntu Node Type (see Fig. 2), which are annotated to represent a particular management feature for their

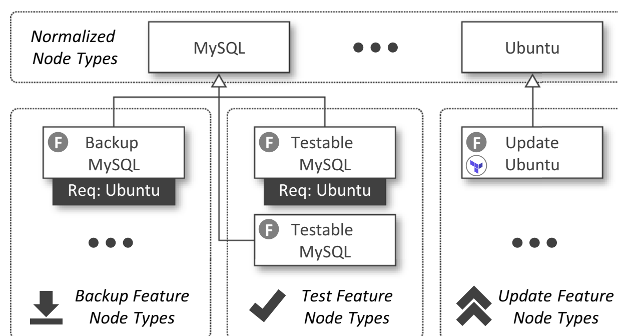

Fig. 2 The TOSCA-based management feature types

parent type. For example, in this case, there are two available features for the MySQL Node Type implemented by three Feature Node Types. Additionally, while there is only one implementation to back up a corresponding MySQL database component, there are two different implementations to test the communication with such a database: one has an additional requirement as it requires that the database is running on a Ubuntu VM, while the second represents a more generic implementation. However, since the backup and test functionalities are state-preserving operations, the Feature Node Types do not have any further annotations as shown in Fig. 2. In contrast, in this repository, there are two Feature Node Types available for the Ubuntu Node Type, offering the capability to also test its availability and to perform an update of the operating system. Hence, by executing the update feature, the state of the application is changed, i.e., it is a state-changing operation, and requires to notify the underlying deployment technology. As depicted on the left side of the *Update Ubuntu* Feature Node Type in Fig. 2, the update feature can only be selected if the Ubuntu VM is managed by an underlying Terraform instance as its implementation only uses the Terraform API to notify it about the state change. Therefore, we extended the Instance Model Enricher to support selecting only features that are able to notify the deployment technology managing a corresponding component. The Feature Node Types, however, may also support multiple deployment technologies and, hence, can be selected if any of these technologies matches the component's deployment technology. As a result, we got one step closer to solve RC 1 and RC 3: We are able to enrich any kind of state-preserving and state-changing functionalities to the running application while ensuring that the implementations are aware of the underlying deployment technologies.

Management Workflow Generator

To execute the enriched management features on running applications, we previously introduced the *Management Workflow Generator* component which generates workflows

for each feature [31]. Thus, they can be executed independently and repeatedly. In general, workflows can be generated from declarative, i.e., graph-based [59], instance models [10]. Hereby, the models are interpreted and, based on the relations between the application's components, the required order in which the operations must be executed can be derived [10]. For example, while backups of independently running databases can be executed in parallel, testing components in one stack, i.e., components on the top depend on components underneath them, should start from the bottom up: If a VM is not reachable, a component running on it will also not be accessible. Therefore, each plugin in the Management Workflow Generator is able to derive the necessary steps and the correct order in which the corresponding feature operations must be performed. Hence, the order in which the tests are run is important to help developers identify the issues.

Similar to the derived instance models, we rely on standardized workflow languages to ensure a complete, standards-based approach. Thus, the workflows can be realized in languages, such as *BPEL* [42] or *BPMN* [45]. As a result of this step in our method, we are able to resolve RC 1 and RC 3, since it is now possible to generate executable workflows that perform any kind of management functionalities.

Workflow Engine

Finally, the generated management workflows must be deployed and executed. This is realized by a standards-based *Workflow Engine* that is capable of running the workflows. Depending on the selected language, an appropriate Workflow Engine has to be selected on which the workflows can be deployed. As a result, a user is able to invoke any of the management features by simply triggering the corresponding workflow. Additionally, the workflows can be integrated into other automated management tasks. For example, by scheduling a regular task triggering a backup workflow.

Architecture and Prototypical Realization of the Integrated Management System

In the following, our prototype realizing the *Integrated management System for Composed Applications* is described. The system is implemented as an extension to the *OpenTOSCA Ecosystem*[4] [12] and a new component which we introduce in this paper called the *TOSCA Instance Model Retriever (TOSCin)*.[5] Both are available open-source on GitHub. The overall system architecture of our prototype is depicted in Fig. 3.

The new Java-based TOSCin Framework realizes the first two components of our approach as a command line interface (CLI): The *Instance Model Retriever* as well as the *Instance Model Normalizer* (see Fig. 3). Therefore, TOSCin is capable of retrieving instance information from the employed deployment technologies and deriving a TOSCA-based instance model. As illustrated in Fig. 3, TOSCin implements plugins for Kubernetes, Terraform, and Puppet to derive a TOSCA-based instance model using a repository that contains normalized TOSCA types. Such normalized types are, for example, already defined in the TOSCA standard definition [44] and are available open-source in a repository we maintain on GitHub[4]. The TOSCA Repository is included in our prototype in the modeling tool *Eclipse Winery* [36]. Therefore, TOSCin uses Winery's API to retrieve TOSCA Types from the Repository, as shown in Fig. 3. Hereby, as we already implemented a Puppet plugin in the conference paper [29], we now extended TOSCin to also support Kubernetes and Terraform. The biggest extension, however, is the merging of all instance models retrieved from all employed deployment technologies into one single instance model. This is realized in the *Instance Model Normalizer* component as its output is a single TOSCA-based instance model. Therefore, it implements multiple similarity checks to detect whether two identified components returned by two different deployment technologies are referring to the same real component. For example, if multiple components are connecting to the same database instance, the corresponding database must occur only once in the instance model which is achieved, e.g., by checking whether the type and location match.

After the Instance Model Normalizer in TOSCin generated a *Normalized Instance Model* of the application, which is a model conforming to the TOSCA standard, it is passed to Eclipse Winery which implements the *Instance Model Completer* as well as the *Instance Model Enricher* as Java-based components. The Instance Model Completer can be hereby easily extended with technology-specific plugins that are capable of identifying additional details about the application components and relations—such as detecting additional components or configuration details. Winery is a graphical modeling tool for TOSCA applications and is part of the OpenTOSCA ecosystem [12]. Hence, in the graph-based user interface (UI), a web application that is implemented in Angular,[6] the user can always see the current changes to the instance model: During the completion of the model using the technology-specific plugins, a user

[4] https://github.com/OpenTOSCA.

[5] Currently part of https://github.com/UST-EDMM/edmm.

[6] https://angular.io/.

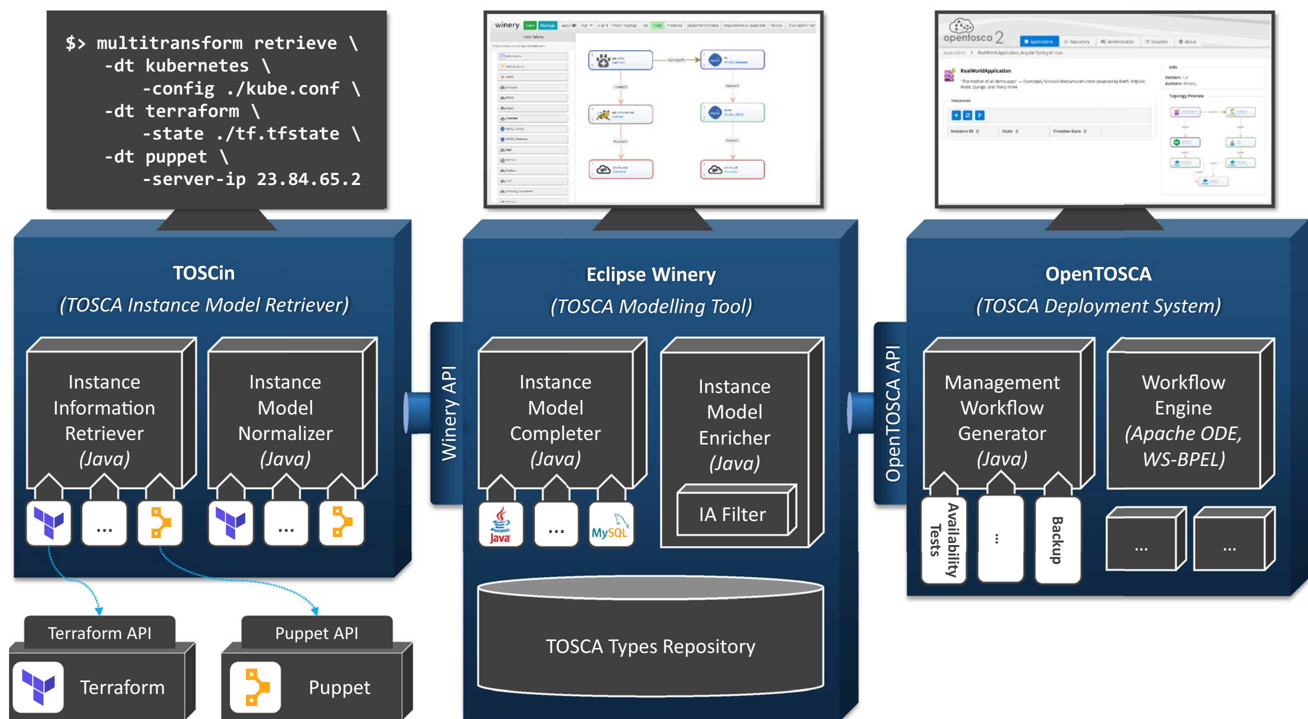

Fig. 3 Prototype architecture based on the TOSCin Framework and the OpenTOSCA ecosystem [12]

can select a plugin from all applicable plugins, i.e., plugins which define a detector that can be found as a sub-graph in the current model (see "Instance Model Completer") in an iterative fashion and directly see the changes if, for example, a new component was detected. Similarly, to enrich the model with additional management features, a user can select the desired management features for each component that are currently available in the TOSCA repository for the Node Types of the component currently used in the model. While the general concept of the *Instance Model Completer* was already presented in the conference paper of this journal extension [29], we extended it to also support different deployment technologies in one instance model. Additionally, we already introduced the *Instance Model Enricher* in a previous work [31], extended it to support deployment technology-specific management features in the conference paper [29], and added support for multiple technologies during the work for this paper.

As a result, Winery outputs a completed and enriched instance model of the running application which is imported to the Java-based OpenTOSCA Orchestrator [5] to generate the BPEL-based management workflows. In the *Management Workflow Generator*, which we presented in the previous work [31], we also use plugins to derive workflows for different kinds of management functionalities. The implementation hereby generates BPEL workflows that can be deployed and executed in an *Apache ODE*[7] instance, as shown in Fig. 3.

Case Study

To prove the practicable feasibility of the approach, we demonstrate how the approach works using a modified version of the Sock Shop[3] application. The Sock Shop is a demo application to demonstrate how applications can be designed and deployed as microservices. It consists of a front-end component in the form of a NodeJS[8] web application where users can buy socks. Therefore, the front end retrieves its catalog data from a Go[9] service which stores its data in a MySQL[10] database. Similarly, the users are managed by a Go application persisting the users' information in a MongoDB[11] database, while items that are currently in a user's

7 https://ode.apache.org/.

8 https://nodejs.org/en/.

9 https://golang.org/.

10 https://www.mysql.com/.

11 https://www.mongodb.com/.

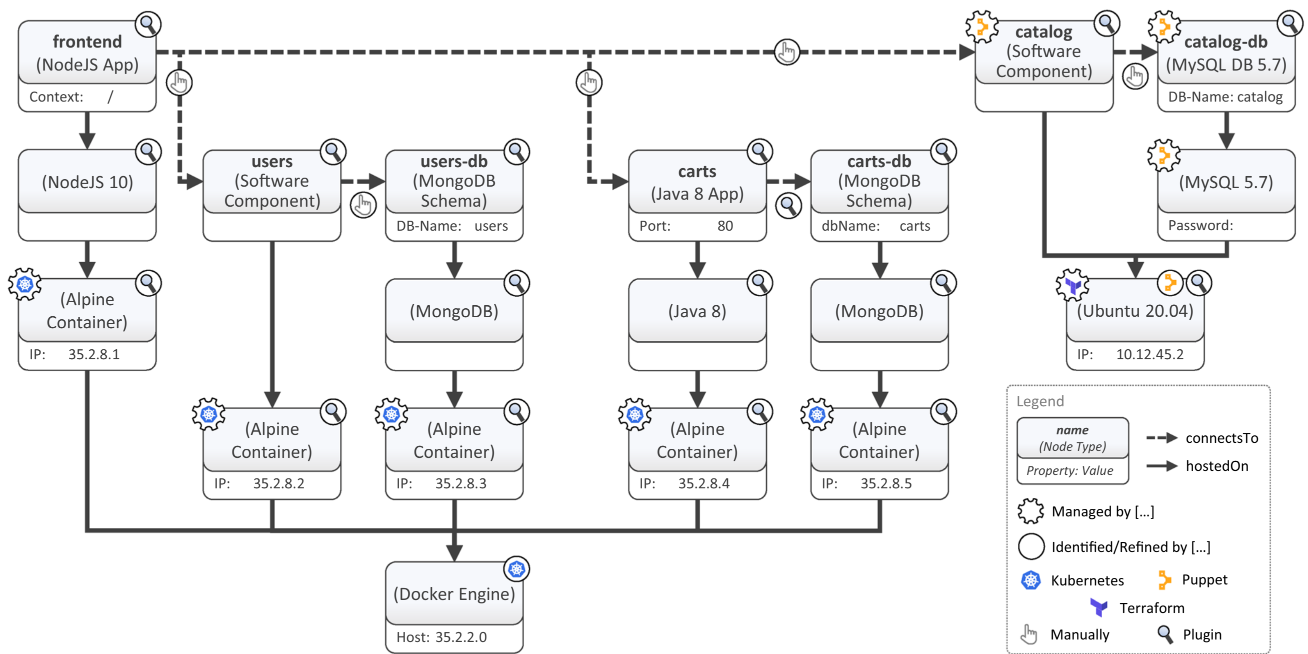

Fig. 4 The instance model derived from an adapted instance of the Sock Shop[3] using our framework: It was deployed without the order service using Kubernetes and using Terraform and Puppet to deploy and configure the catalog service and its database

cart are stored by a Java service which also uses a MongoDB database to save the data. Additionally, the Sock Shop consists of an ordering service, a payment service, as well as a shipping service that places the orders in a RabbitMQ[12] messaging service. However, for simplicity, the last four services and their associated databases are omitted in Fig. 4 and the following explanations.

The application was deployed using Terraform, Puppet, and Kubernetes: While the front-end component as well as the users service and the carts service, alongside with their associated databases, are deployed in the form of Docker Containers[13] using Kubernetes, the catalog service and its database are deployed using Puppet on a VM provisioned by Terraform. Based on this setup, we used our prototype to enrich and execute additional management functionalities for the Sock Shop.

To enable holistic management processes for the running Sock Shop application composed of multiple parts that are deployed with three different deployment technologies, first, an instance model must be derived. Therefore, TOSCin is invoked with the endpoints and credentials for the deployment technologies used to deploy the application. In this case, the endpoints and credentials for the Kubernetes cluster, the Terraform state file, as well as the Puppet's primary server that we used to deploy the Sock Shop are passed as input to TOSCin. Using this information, the Instance Information Retriever component invokes the corresponding technology-specific plugins to retrieve runtime information about the running application. In the next step, the data are interpreted by the Instance Model Normalizer plugins which derive a TOSCA-based instance model of the application. Hereby, the technology-specific types are mapped to normalized TOSCA Node Types, while the detected components are represented as Node Templates that have the identified Node Types assigned. Moreover, as the technology-specific types may define different properties than the identified normalized Node Types, they must also be mapped. For example, while a technology-specific type may define a whole URL, a corresponding Node Type may define the hostname and port separately. Thus, the property must be split into the corresponding parts to be mapped correctly to the normalized type using, e.g., regular expressions, as we presented in detail in more detail in the previous work [29]. Additionally, to support multiple deployment automation technologies in one instance model, all components are annotated with an *identified by* annotation to indicate that the component was identified by a corresponding deployment technology. Because some deployment technologies are able to identify components that are not managed by them, an additional *managed by* annotation is used to map the components to their managing deployment technology. For example, as shown in Fig. 4, all containers and the underlying Docker

12 https://www.rabbitmq.com/.

13 https://www.docker.com/.

Engine that is used in this case to run the containers are annotated with an *identified by Kubernetes* annotation. In contrast, only the containers are tagged with the *managed by Kubernetes* annotation as the Docker Engine is only used by Kubernetes.

In the next step, all Topology Templates generated by the Instance Information Retriever plugins containing the identified and managed components of a single deployment technology are merged into a single Topology Template to represent all parts of the application in one model. Thereby, if a component has been identified by multiple deployment technologies, the component is merged and tagged with all *identified by* annotations. For example, the Ubuntu VM shown in Fig. 4 was detected by Terraform and Puppet as Terraform manages the VM and Puppet connects to it to install and configure the components it manages.

Up to this point, the derived instance model shown in Fig. 4 only contains those components that are annotated with an *identified by* annotation which contain deployment technology icons. All other components that are solely annotated with an *identified by a Plugin* annotation, i.e., a circle with a magnifier in its center, are not identified yet. These components are added to the model by iteratively running the component-specific plugins of the Instance Model Completer which is implemented in Winery. During each iteration, all registered plugins are first checked for their applicability, i.e., whether they could contribute additional information to the instance model. All applicable plugins are collected in a list, whereby each plugin specifies the list of components it can refine. Then, a user selects a particular plugin and a corresponding set of components the plugin should refine. During the plugin's execution, it may identify additional components, properties, and relations, or refine the types of components from abstract ones to more concrete ones. For example, a NodeJS plugin is able to identify that a NodeJS application is running on a NodeJS webserver on the front-end container. Another plugin identifies the containers' operating systems, while a third detects additional properties, such as the port a Java application is listening to for requests. Additionally, each plugin adds an *identified by* annotation with its ID to the component to document how the component was detected. This repeats until no more plugins can be found applicable, or the user stops the completion phase. Finally, a model, such as the one illustrated in Fig. 4, is derived. However, as shown in Fig. 4, the type of the users component as well as the catalog component are instances of the Node Type *Software Component* instead of *Go Application*. The reason for this is that Go applications are compiled to binaries. Therefore, we are able to identify that components have been deployed on the containers but cannot identify a concrete type. Similarly, the horizontal relations between most services are annotated with an *identified manually* annotation, as only the relation between the Java component, i.e., the carts service, and its database could be identified automatically. In future work, we plan to implement additional plugins that are using network scanning approaches, such as presented in [34], to detect more horizontal relations.

The completed instance model can then be interpreted by the Instance Model Enricher that is also implemented in Winery. It is based on the *Management Feature Enricher* from previous work [31] and was extended to support state-changing and state-preserving operations [29]. Additionally, different kinds of implementations for state-changing operations must be supported and chosen correctly. Therefore, the Instance Model Enricher uses the *managed by* annotation at the components to filter applicable feature implementations. For example, since the Ubuntu VM is managed by Terraform, state-changing management features must notify Terraform about the change and, thus, a management feature can be enriched to the application if a corresponding implementation is available. To achieve this, we extended our repository in this work to contain implementations to back up the MongoDB and MySQL databases, several test operations, as well as an update implementation of the Ubuntu VM that is able to communicate the state change to Terraform.

In the last step, the enriched instance model is interpreted by the OpenTOSCA Runtime which generates the different management workflows and deploys them on an Apache ODE workflow engine [31]. As a result, to perform a particular management feature, a user can invoke the corresponding management workflow. However, because properties, such as credentials and passwords, cannot be automatically retrieved, the user may have to pass them to the workflows as input.

Threats to Validity

In this section, the threats to validity regarding the presented case study are discussed, following the structure defined by Wohlin et al. [54]. As the goal of this paper is to enable the automated management of running applications that have been deployed using multiple deployment technologies, the threats to validity mainly affect the quality of the derived instance model.

In the *Internal Validity*, threats to the relationship between the treatment and the experiment's result are discussed [54]. However, since applications in our scenarios are deployed using different deployment technologies, a holistic instance model of the application that combines all components maintained by the different deployment technologies does not exist without explicit efforts. Therefore, we need an approach and prototype as ours to generate this holistic model. Thus, we have evidence that only our approach and

prototype caused the existence of the holistic model, which is then used for management. Moreover, there are no other causes that influenced the derived model or the subsequent management of the application. Thus, there are no threats to the internal validity.

The *External Validity* discusses limitations of the generalizations derived from the experiment [54]. The discussed case study was chosen because of its modern design and represents a state-of-the-art microservice application. In practice, however, legacy applications may not be deployed using state-of-the-art deployment technologies and, thus, the chosen case study may not represent them. However, it is also possible to run the approach without deployment technologies. In this case, a user can provide an entry point, such as a VM, together with credentials and a location in a manually created instance model. Thus, it is possible to start the approach with the Instance Model Completion phase whereby an instance model of the application is created only by executing different completion plugins. Moreover, since the management enrichment, management workflow generation, and execution of management features are not bound to any deployment technology, these steps can also be executed if not all components are deployed using deployment technologies.

The *Construct Validity* concerns the generalization of the experiment's results to the theory behind [54]. Since the main building block of our approach is the generation of a holistic instance model of a running application to enable its automated management, our case study represents exactly these aspects. However, as we only described a single case study, this may pose a threat to the construct validity. Thereby, the case study was designed as a modern architecture that contains the typical components of today's applications. Additionally, we used three of the most used deployment technologies that prevailed in practice and are commonly used in today's application deployments [59]. Therefore, we have shown that our approach supports a huge variety of components and deployment technologies.

The *Conclusion Validity* refers to threats that influence the ability to draw conclusions from an experiment [54]. In the presented case study, the extracted instance model poses a threat to the conclusion validity regarding the completeness of the generated instance model. Although we were able to identify all components of the described case study, running the prototype against other applications may not yield a complete instance model of the application, since not all deployment technologies and components used in practice are integrated in our prototype yet. However, in our case study, we used different components that are typical for modern deployments. Moreover, the three selected deployment technologies all implement the declarative deployment modeling approach, which is also followed by the 13 most used deployment technologies in practice [59]. Therefore, the deployment technologies and components used in our case study are very similar to the ones we do not yet support in our prototype. As a result, there is only a low risk that our approach cannot incorporate other technologies in a way that the derived instance model is incomplete.

Discussion and Current Limitations

Our presented approach aims at enabling the automated management of composed applications deployed using multiple deployment automation technologies. This is achieved by retrieving instance models of running applications that have been deployed by multiple deployment technologies and enrich these models with additional management functionalities to enable an automated management of the application. Thereby, our assumption is that a component is managed by exactly one deployment technology. However, there might be situations in which this is not the case. For example, if one deployment technology is managing the physical infrastructure properties of a component, such as number of CPUs, RAM, and network capabilities, another might be managing the component's lifecycle and configuration. However, because this example might not be the only case in which multiple deployment technologies are working together at one component, we plan to evaluate how deployment technologies are working together in future work.

The instance information that can be retrieved from the deployment technologies differ significantly in their expressiveness and level of detail. For example, while Terraform and Kubernetes mostly provide information about infrastructural components, such as VMs and containers, Chef and Puppet provide more information about software components and their configurations. Hence, we added the *Instance Model Completion* step (see "Instance Model Completer") that reuses and extends an existing approach to identify as many components of the application as possible. Because the Instance Model Completer builds on a plugin system to identify or refine additional components, properties, or (horizontal) relations between components, a large amount of plugins is required to identify and refine various components and technologies. As an additional option, it is always possible to adapt the models manually of course to enhance their expressiveness.

Because the approach relies on many plugins and management operations that must be implemented for the approach to work properly, a large effort is required to support the wide variety of commonly available technologies. However, once the plugins and management operations are implemented, they can be reused to improve the retrieved instance model and the number of compatible management operations rises. Hence, the more plugins and management

operations are available, the more applications can be automatically enriched with management features.

Identifying horizontal relations, e.g., that an application is connecting to a database, is extremely difficult if no single deployment model of the entire application is available that describes all dependencies. Even if a large amount of plugins to complete the instance models are available, identifying horizontal relations remains difficult. The reason for this is that the configuration of each component can be realized in countless ways that are not always visible or accessible from the outside. For example, a common configuration option is to use environment variables as they can be set from the outside to configure an application's component. However, neither the component to which an environment variable belongs cannot be determined generically, nor are the names and values of the variables standardized. Thus, while scanning for common parts in the variable names or configuration flags may yield information that can be used to derive horizontal relations, there is no guarantee that they can be identified automatically by analyzing information retrieved from the deployment technologies and the running instance. However, there are other possibilities to implement plugins: a network scan and monitoring can find the packages that are sent from one component to another and, thus, detecting horizontal relations. Other possibilities are code scans, as shown by Genfer and Zdun [27]. Therefore, various possibilities exist that can be used to detect horizontal relations, although many of them are not trivial. On the other hand, once they are implemented, they can be reused directly in the future to identify the communication of arbitrary application components. Of course, adding horizontal relations manually is always possible.

To implement state-changing functionalities, a notification of the underlying deployment technology about the performed change must be realized. Otherwise, the changes may be reverted. Hence, for each state-changing functionality, different implementations for each supported deployment technology may be necessary. For example, to manage containers that are deployed using Kubernetes, so-called *Kubernetes Operations* can be used to implement custom management functionalities, while the management of components managed by Puppet may require the implementation to be in Puppets' domain specific language (DSL). Hence, the resulting implementations may become quite complex and operations engineers need to use the corresponding deployment technologies' API to realize the desired management feature as an operation that can be automatically enriched to a running application. However, similar to the plugins, once implemented, the operations can be reused in other applications that use a similar combination of deployment technology and component type.

Moreover, the management of container-based parts of an application poses additional challenges as a container is usually composed of multiple components that cannot be seen from the outside. However, similar to VMs, containers are also accessible, e.g., via interactive shells such as SSH. This can be exploited in the Instance Model Completer plugins to identify additional components that have not been detected yet. As a result, also hidden components, no matter if they are deployed in VMs or containers, can be identified and enriched with management functionalities.

Conclusion and Future Work

In this paper, we demonstrated a how running applications that are composed of multiple components deployed by different deployment automation technologies can be enriched with holistic management capabilities in retrospective. However, there are still manual steps required to enable the management of applications beforehand. For example, implementations for each management feature are required which, in the case of state-changing functionality, must be even realized in multiple versions, i.e., for each deployment technology. Nevertheless, if they are implemented once, they can be reused in any other application in contrast to custom implementations that are specialized for a specific application. Moreover, by generating management workflows for each functionality separately, we enable repeatable executions of the management operations.

In future work, we plan to implement more use cases and therefore more plugins and different management functionalities while also considering using code analysis tools as presented by Genfer and Zedun [27]. Additionally, we are investigating how it can be possible to retrieve also deployment artifacts to enable the deployment of the application based on the derived model.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was partially funded by the Federal Ministry for Economic Affairs and Climate Action of Germany (BMWK) project *PlanQK* (01MK20005N) and the German Research Foundation (DFG) IaC^2 (314720630).

Data availability The prototype's components and use case are all publicly available on GitHub. All corresponding links are provided in the footnotes.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes

were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Amazon: AWS CloudFormation 2023. https://aws.amazon.com/de/cloudformation/.
2. Bencomo N, Götz S, Song H. Models@run.time: a guided tour of the state of the art and research challenges. Softw Syst Model. 2019;18(5):3049–82.
3. Bergmayr A, Breitenbücher U, Ferry N, Rossini A, Solberg A, Wimmer M, Kappel G, Leymann F. A Systematic Review of Cloud Modeling Languages. ACM Comput Surv (CSUR). 2018;51(1):1–38.
4. Binz T. Crawling von Enterprise Topologien zur automatisierten Migration von Anwendungen: eine Cloud-Perspektive. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik 2015.
5. Binz T, Breitenbücher U, Haupt F, Kopp O, Leymann F, Nowak A, Wagner S. OpenTOSCA—a Runtime for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013), LNCS, 2013; vol. 8274, pp. 692–695. Springer.
6. Binz T, Breitenbücher U, Kopp O, Leymann F. Automated discovery and maintenance of enterprise topology graphs. In: Proceedings of the 6th IEEE international conference on service oriented computing and applications (SOCA 2013). 2013; pp. 126–134. IEEE.
7. Binz T, Breiter G, Leymann F, Spatzier T. Portable Cloud services using TOSCA. IEEE Internet Comput. 2012;16(03):80–5.
8. Blair G, Bencomo N, France RB. Models@run.time. Computer. 2009;42(10):22–7. https://doi.org/10.1109/MC.2009.326.
9. Breitenbücher U. Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology 2016.
10. Breitenbücher U, Binz T, Képes K, Kopp O, Leymann F, Wettinger J. Combining declarative and imperative cloud application provisioning based on TOSCA. In: International conference on cloud engineering (IC2E 2014). 2014; pp. 87–96. IEEE.
11. Breitenbücher U, Binz T, Kopp O, Leymann F. Pattern-based runtime management of composite cloud applications. In: Proceedings of the 3rd international conference on cloud computing and services science (CLOSER 2013). 2013; pp. 475–482. SciTePress.
12. Breitenbücher U, Endres C, Képes K, Kopp O, Leymann F, Wagner S, Wettinger J, Zimmermann M. The OpenTOSCA Ecosystem—Concepts & Tools. European space project on smart systems, big data, future internet–towards serving the grand societal challenges—Volume 1: EPS Rome 2016 2016; pp. 112–130.
13. Breitenbücher U, Képes K, Leymann F, Wurster M. Declarative vs. imperative: how to model the automated deployment of IoT Applications? In: Proceedings of the 11th advanced summer school on service oriented computing. 2017; pp. 18–27. IBM Research Division.
14. Brogi A, Cifariello P, Soldani J. DrACO: discovering available cloud offerings. Comput Sci Res Dev. 2017;32(3–4):269–79.
15. Brogi A, Forti S, Guerrero C, Lera I. Towards declarative decentralised application management in the fog. In: 2020 IEEE international symposium on software reliability engineering workshops (ISSREW). 2020; pp. 223–230.
16. Brown A, Keller A. A best practice approach for automating IT management processes. In: Proceedings of the 10th IEEE/IFIP network operations and management symposium (NOMS 2006). 2006; pp. 33–44. IEEE.
17. Calcaterra D, Cartelli V, Modica GD, Tomarchio O. Combining TOSCA and BPMN to enable automated cloud service provisioning. In: Proceedings of the 7th international conference on cloud computing and services science—Vol. 1: CLOSER. 2017; pp. 187–196. SciTePress.
18. Chappell D, et al. What is application lifecycle management. Westbury: Chappell & Associates; 2008.
19. Chardet M, Coullon H, Robillard S. Toward safe and efficient reconfiguration with Concerto. Sci Comput Program. 2021;203: 102582.
20. Di Nitto E, Matthews P, Petcu D, Solberg A. Model-driven development and operation of multi-cloud applications: the MODAClouds approach. New York: Springer Nature; 2017.
21. Eilam T, Elder M, Konstantinou AV, Snible E. Pattern-based Composite Application Deployment. In: Proceedings of the 12th IFIP/IEEE international symposium on integrated network management (IM 2011) 2011; pp. 217–224. IEEE.
22. El Maghraoui K, Meghranjani A, Eilam T, Kalantar M, Konstantinou A. Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools. In: Proceedings of the 7th international middleware conference (middleware 2006). 2006; pp. 404–423. Springer.
23. Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J. 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 2017). 2017; pp. 22–27. Xpert Publishing Services.
24. Endres C, Breitenbücher U, Leymann F, Wettinger J. Anything to Topology—a method and system architecture to topologize technology-specific application deployment artifacts. In: Proceedings of the 7th international conference on cloud computing and services science (CLOSER 2017), Porto, Portugal. 2017; pp. 180–190. SciTePress.
25. Farwick M, Agreiter B, Breu R, Ryll S, Voges K, Hanschke I. Automation processes for enterprise architecture management. In: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops. 2011; pp. 340–9.
26. Fittkau F, Roth S, Hasselbring W. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: ECIS 2015. AIS 2015.
27. Genfer P, Zdun U. Identifying domain-based cyclic dependencies in microservice apis using source code detectors. In: Biffl S, Navarro E, Löwe W, Sirjani M, Mirandola R, Weyns D, editors. Software architecture. New York: Springer International Publishing; 2021. p. 207–22.
28. Guerriero M, Garriga M, Tamburri DA, Palomba F. Adoption, support, and challenges of infrastructure-as-code: insights from industry. In: 2019 IEEE international conference on software maintenance and evolution (ICSME). 2019; pp. 580–589. IEEE.
29. Harzenetter L, Binz T, Breitenbücher U, Leymann F, Wurster M. Automated generation of management workflows for running applications by deriving and enriching instance models. In: Proceedings of the 11th international conference on cloud computing and services science (CLOSER 2021). 2021; pp. 99–110. SciTePress.
30. Harzenetter L, Breitenbücher U, Képes K, Leymann F. Freezing and defrosting cloud applications: automated saving and restoring

of running applications. Softw Intensive Cyber-Phys Syst (SICS). 2019;35:101–14.

31. Harzenetter L, Breitenbücher U, Leymann F, Saatkamp K, Weder B, Wurster M. Automated generation of management workflows for applications based on deployment models. In: 2019 IEEE 23rd international enterprise distributed object computing conference (EDOC 2019). 2019; pp. 216–225. IEEE.
32. HashiCorp: Terraform.io 2023. https://www.terraform.io/
33. Herden S, Zwanziger A, Robinson P. Declarative application deployment and change management. In: Proceedings of the 2010 international conference on network and service management (CNSM 2010). 2010; pp. 126–133. IEEE.
34. Holm H, Buschle M, Lagerström R, Ekstedt M. Automatic data collection for enterprise architecture models. Softw Syst Model. 2014;13(2):825–41.
35. Képes K, Leymann F, Weder B, Wild K. SiDD: the situation-aware distributed deployment system. In: Service-oriented computing—ICSOC 2020 workshops, pp. 72–76. Springer International Publishing 2021.
36. Kopp O, Binz T, Breitenbücher U, Leymann F. Winery—a modeling tool for tosca-based cloud applications. In: Proceedings of the 11th international conference on service-oriented computing (ICSOC 2013). 2013; pp. 700–704. Springer.
37. Leymann F. Cloud computing: the next revolution in IT. In: Proceedings of the 52th photogrammetric week. 2009; pp. 3–12. Wichmann Verlag.
38. Leymann F, Fehling C, Wagner S, Wettinger J. Native cloud applications: why virtual machines, images and containers miss the point! In: Proceedings of the 6th international conference on cloud computing and service science (CLOSER 2016). 2016; pp. 7–15. SciTePress, Rome.
39. Machiraju V, Dekhil M, Wurster K, Garg PK, Griss ML, Holland J. Towards generic application auto-discovery. In: Proceedings of the 7th IEEE/IFIP network operations and management symposium (NOMS 2000). 2000; pp. 75–87. IEEE.
40. Menzel M, Klems M, Le HA, Tai S. A configuration crawler for virtual appliances in compute clouds. In: 2013 IEEE international conference on cloud engineering (IC2E). 2013; pp. 201–209. IEEE.
41. Mietzner R, Unger T, Leymann F. Cafe: A generic configurable customizable composite cloud application framework. In: On the move to meaningful internet systems: OTM 2009 (CoopIS 2009), 2009; pp. 357–364. Springer.
42. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS) 2007.
43. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) 2013.
44. OASIS: TOSCA Simple Profile in YAML Version 1.3. Organization for the Advancement of Structured Information Standards (OASIS) 2020.
45. OMG: Business Process Model and Notation (BPMN) Version 2.0. Object Management Group (OMG) 2011.
46. Oppenheimer D. The importance of understanding distributed system configuration. In: Proceedings of the 2003 conference on human factors in computer systems workshop (CHI 2003). ACM 2003.
47. Petcu D. Consuming resources and services from multiple clouds. J Grid Comput. 2014;12(2):321–45.
48. Progress Software Corporation: Chef Infrastructure Management 2023. https://www.chef.io/products/chef-infrastructure-management.
49. Puppet: Puppet 2023. https://puppet.com/.
50. Red Hat Inc. Ansible Official Site 2023. https://www.ansible.com/.
51. The Linux Foundation: kubernetes.io 2023. https://kubernetes.io/.
52. Weller M, Breitenbücher U, Speth S, Becker S. The deployment model abstraction framework. In: Enterprise design, operations, and computing. EDOC 2022 Workshops. 2023; pp. 319–325. Springer.
53. Wild K, Breitenbücher U, Képes K, Leymann F, Weder B. Decentralized cross-organizational application deployment automation: an approach for generating deployment choreographies based on declarative deployment models. In: Proceedings of the 32nd conference on advanced information systems engineering (CAiSE 2020), Lecture notes in computer science, vol. 12127, pp. 20–35. Springer International Publishing 2020.
54. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. Experimentation in software engineering. Berlin: Springer; 2012. https://doi.org/10.1007/978-3-642-29044-2.
55. Wurster M, Breitenbücher U, Brogi A, Diez F, Leymann F, Soldani J, Wild K. Automating the deployment of distributed applications by combining multiple deployment technologies. In: Proceedings of the 11th international conference on cloud computing and services science (CLOSER 2021), pp. 178–189. SciTePress 2021.
56. Wurster M, Breitenbücher U, Brogi A, Harzenetter L, Leymann F, Soldani J. Technology-agnostic declarative deployment automation of cloud applications. In: Service-oriented and cloud computing. ESOCC 2020. Lecture notes in computer science, Vol. 12054, pp. 97–112. Springer 2020.
57. Wurster M, Breitenbücher U, Brogi A, Harzenetter L, Leymann F, Soldani J. Technology-agnostic declarative deployment automation of cloud applications. In: Proceedings of the 8th European conference on service-oriented and cloud computing (ESOCC 2020), pp. 97–112. Springer International Publishing 2020.
58. Wurster M, Breitenbücher U, Brogi A, Leymann F, Soldani J. Cloud-native Deploy-ability: an analysis of required features of deployment technologies to deploy arbitrary cloud-native applications. In: Proceedings of the 10th international conference on cloud computing and services science (CLOSER 2020), pp. 171–180. SciTePress 2020.
59. Wurster M, Breitenbücher U, Falkenthal M, Krieger C, Leymann F, Saatkamp K, Soldani J. The essential deployment metamodel: a systematic review of deployment automation technologies. SICS Softw Intensive Cyber-Phys Syst. 2019;35:63–75.
60. Wurster M, Breitenbücher U, Harzenetter L, Leymann F, Soldani J, Yussupov V. TOSCA Light: bridging the gap between the TOSCA specification and production-ready deployment technologies. In: Proceedings of the 10th international conference on cloud computing and services science (CLOSER 2020), pp. 216–226. SciTePress 2020.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.