Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Multi-Deployment-Technology Instance Model Retrieval and Instance Management

Alexandros Fouskas

Course of Study: Informatik

Examiner: Prof. Dr. Dr. h.c. Frank Leymann

Supervisor: Lukas Harzenetter, M.Sc.

Commenced: May 10, 2021

Completed: November 10, 2021

Abstract

Many enterprise applications are built up from multiple components. Deployment and management of these applications is complex and error-prone, especially if performed manually. Thus, automation is a key factor especially with the advent of cloud computing. To cope with this, a variety of deployment technologies has been introduced in recent years. These technologies automate the deployment and management of applications and have been widely adopted in industry and research. Many organizations even use multiple deployment technologies in parallel. However, the management capabilities provided by these technologies are often limited. Thus, complex management operations, e. g., backups of all components, must still be executed manually. Moreover, deployment technologies may interfere with management operations, so the deployment technologies must be considered when executing the operations. This becomes even harder, if different deployment technologies are used to deploy different parts of the application that should be managed. Thus, this work extends the existing management workflow generation approach to support applications that have been deployed by multiple deployment technologies. To achieve this, this work connects to the APIs of the deployment technologies, in order to retrieve instance information. The retrieved information is used to derive an instance model that represents the current state of the application. The instance model is enriched with management functionality and is used to generate management workflows that can be executed on-demand.

Kurzfassung

Viele Enterprise-Anwendungen bestehen aus einer Vielzahl von einzelnen Komponenten. Sowohl Deployment als auch Management dieser Anwendungen sind komplexe und fehleranfällige Aufgaben, besonders wenn diese manuell ausgeführt werden. Daher stellt Automatisierung einen Schlüsselfaktor dar, besonders mit dem Einzug von Cloud Computing. Um dem zu begegnen, wurde in den letzten Jahren eine Vielzahl sogenannter Deployment Technologien eingeführt. Diese Technologien automatisieren Deployment und Management von Anwendungen und haben sowohl in Industrie als auch in der Forschung weite Verbreitung gefunden. Viele Organisationen setzen sogar mehrere verschiedene Deployment Technologien gleichzeitig ein. Die Mangementfunktionalitäten dieser Deployment Technologien sind jedoch meist begrenzt. Als Folge daraus, müssen komplexe Managementoperationen, zum Beispiel Backups aller Komponenten, weiterhing manuell ausgeführt werden. Zudem, können Deployment Technologien auch die Ausführung von Managementoperationen behindern. Daher müssen die Deployment Technologien beim erstellen der Managementoperationen berücksichtigt werden. Durch den parallelen Einsatz mehrerer Technologien wird dies noch erschwert, da unterschiedliche Komponenten einer Anwendung von unterschiedlichen Technologien verwaltet werden. Aus diesem Grund erweitert diese Arbeit den bestehenden Ansatz der Management-Workflow-Generierung, damit dieser auch Anwendungen unterstützt, welche durche mehrere Deployment Technologien bereitgestellt werden. Um dies zu erreichen, werden Instanzinformationen von den Schnittstellen der Deployment Technologien bezogen und benutzt, um ein Instanzmodell zu erzeugen. Dieses Instanzmodell spiegelt den aktuellen Zustand der Anwendung wieder. Darüber hinaus, wird das Instanzmodell mit zusätzlichen Managementoperationen angereichert. Das so angereicherte Instanzmodell wird benutzt um automatisiert Management-Workflows zu generieren, welche bei Bedarf ausgeführt werden können.

Contents

List of Figures

List of Listings

Acronyms

API	Application Programming Interface. 3, 14
AWS	Amazon Web Services. 7, 9, 17
BPEL	Business Process Execution Language. 25
BPMN	Business Process Modeling and Notation. 25
CLI	Command Line Interface. 23
CSAR	Cloud Service Archive. 7, 26
DBMS	Database Management System. 18
DrACO	Discovering Available Cloud Offerings. 31
DSL	Domain-Specific Language. 21
EAM	Enterprise Architecture Management. 31
EC2	Elastic Compute Cloud 2. 9, 22
EDMM	Essential Deployment Meta Model. 32
EDMMi	Essential Deployment Meta Model instance. 32
EFS	Elastic File System. 22
ETG	Enterprise Topology Graph. 31
HTTP	Hypertext Transfer Protocol. 33
IA	Implementation Artifact. 26
IaaS	Infrastructure as a Service. 31
IP	Internet Protocol. 25
JAR	Java Archive. 18
JSON	Java Script Object Notation. 22, 45, 46, 51, 57, 58, 61
JVM	Java Virtual Machine. 18
MARIO	Managing Applications Running In Opportunistic Fog. 32
MFEW	Management Feature Enrichment and Workflow Generation. 32
OASIS	Organization for the Advancement of Structured Information Standards. 24
OS	operating system. 14

PaaS	Platform as a Service. 31
REST	Representational State Transfer. 23
SaaS	Software as a Service. 19
SSH	Secure Shell. 33
TCP	Transmission Control Protocol. 31
TOSCA	Technology and Orchestration Specification for Cloud Applications. 5, 7, 15
UDP	User Datagram Protocol. 31
UI	User Interface. 23
VM	Virtual Machine. 13
YAML	YAML Ain't Markup Language. 20, 22, 55

1 Introduction

Many modern applications are built up from several components and often use complex middleware [Dea07]. The deployment and configuration of such applications are inherently complex tasks, that require deep technical knowledge [BKH05]. Thus, the manual execution of these tasks is not only time-consuming, but also error-prone [Opp03]. To tackle this issue, Oppenheimer [Opp03] proposes to automate the deployment and management processes. With the advent of cloud computing, IT resources can be provisioned by external providers in on-demand self-service model and are payed-per-use [LF09]. Thus, it can also save costs when the deployment process is automated [LF09]. As a response to this, many so called deployment technologies have been developed that automate the deployment process. Examples are AWS CloudFormation [Ama21], Puppet [Pup21a], Chef [Pro21] or Terraform [Has21b]. These technologies differ in the modeling languages used and the use cases they best fit. As an example, Terraform focuses on providing infrastructure in cloud environments, e. g., Virtual Machines (VMs), using its own configuration language. Yet, in order to deploy software components onto the provided infrastructure, Terraform proposes to use other tools, e. g., Puppet [Has21a]. However, most deployment technologies take a model of the application, the *deployment model*, and provide mechanisms to put the application into production in the target environment [BBF+18]. In most cases, a *declarative deployment model* is used [WBF+19]. A declarative model describes the desired application state i. e., the components and their relations. A simple example consists of two components, a web application and a database, and one relation, the web app is connecting to the database. The deployment technology transforms this model into concrete execution steps for deployment and executes them. In contrast, *imperative deployment models* define concrete workflows that contain ordered steps necessary to deploy the application.

In addition to deployment automation, most deployment technologies provide simple management functionalities, like scaling of single components or observing component health [HBB+21]. However, modern enterprise applications require more complex management operations, e. g., (cross-) component backups or installation of security patches. Such operations are not supported by most modern deployment technologies [HBB+21]. As a consequence, these operations must again be performed manually, which makes them cumbersome and error-prone [Opp03].

As a solution Harzenetter et al. [HBL+19] introduced an approach to automate the execution of management operations based on the declarative deployment model of an application. They propose to enrich the components in an existing deployment model with predefined, reusable management features. For example, a database component may be enriched with a backup management feature that defines how the backup operation can be executed. The enriched deployment models are used to generate *management workflows* for each defined management feature, that can be executed on-demand. The approach allows for extensive automation but requires an existing deployment model as input. As a consequence, it does not support the management of already running applications or applications for which no single deployment model exists.

Moreover, deployment technologies may interfere with the execution of management workflows, as the deployment technologies are monitoring component state for deviations from the desired sate and possibly revert applied changes [HBB+21]. For example, consider the installation of security updates for the operating system (OS) of a VM. The deployment technology used to create the VM might detect a deviation of the VM from its desired state. Thus, it could stop the VM and restart it without the applied security updates. Another example are component backups. For a successful backup, it might be necessary to stop or pause specific components for the duration of the backup operation. For example, a database may need to stop accepting new connections to allow a clean snapshot for backup purposes. However, a deployment technology may interpret this behavior as failure and restart the database service, thus interrupting the backup operation.

To solve this, Harzenetter et al. [HBB+21] proposed the retrieval of *instance models* for running applications. *Instance models* are declarative models that represent the current state of the application. The approach retrieves information about the running application using the Application Programming Interface (API) of a single deployment technology. This *instance model* can the again be enriched with management features by adapting the previous approach of Harzenetter et al. [HBL+19]. In addition, the instance model contains information about the used deployment technology. This allows the management workflow generation to consider the deployment technology, to prevent interference in workflow execution. However, the approach only considers a single deployment technology.

Complex applications might not only use a single but multiple deployment technologies to deploy different parts of the application. As an example, the VMs for an application might be deployed using Terraform, while the software components are deployed upon these machines using Puppet. The usage of different deployment technologies often implies that there is no single *deployment model* that describes the complete application. As a consequence, previously presented approaches cannot be used to automate the management of these applications. Thus, this work introduces an approach, to automatically generate executable management workflows for running applications that have been deployed using multiple deployment technologies.

With the usage of multiple deployment technologies, several additional issues arise for the management of running applications. First, all deployment technologies, used for the deployment of an application, must be identified. This can be tedious, as the information may have to be retrieved across development team boundaries. Second, instance information must be retrieved from all involved deployment technologies and must be mapped to a normalized instance model. This requires deep technical knowledge, in order to connect to the deployment technologies, retrieve the provided instance information, and map the deployment technology specific information to a normalized instance model. Moreover, retrieving instance information from multiple deployment technologies leads to multiple instance models, i. e., one for each deployment technology. To provide a single holistic instance model for the complete application, these instance models must be merged. This process is hard, since the retrieved instance models may overlap in some parts, e. g., specifying the same components. In addition, the instance model must contain information about the involved deployment technologies and must specify, which deployment technology manages which component. In addition, the usage of multiple deployment technologies impacts the process of management feature enrichment as well as management workflow generation. Both processes must be adapted to consider the deployment technologies represented in the instance model.

To tackle the aforementioned issues, this work retrieves instance models for running applications, that are deployed by multiple deployment technologies. Moreover, it describes, how the retrieved instance model can be used to generate management workflows that target the different deployment technologies in use. The approach extends the existing work of Harzenetter et al. [HBB+21] by describing, how application components, that are managed by different deployment technologies can be modeled in an instance model based on the Technology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13a], a standard for modeling the provisioning and management of cloud applications. In addition, this work describes how the APIs of multiple deployment technologies can be queried, in order to retrieve such an instance model. As the APIs of deployment technologies often provide only limited instance information [HBB+21], this work also includes how the retrieved instance models can be refined with additional information. The completed instance model is enriched with management operations, based on the work of Harzenetter et al. [HBL+19] and management workflow generation is extended to work with multiple deployment technologies. To validate the proposed approach, a prototypical implementation based on the OpenTOSCA ecosystem [BEK+16] is also part of this work.

The remainder of this work is structured as follows: Chapter 2 describes and explains the fundamental concepts of cloud application deployment, instance models and deployment technologies. Chapter 3 lists related work and highlights the differences to the concepts and approaches presented in this work. The main approach proposed by this work is described in Chapter 4. Chapter 5 describes the details of the prototypical implementation and Chapter 6 presents a case study that applies the prototypical implementation to an example application. Chapter 7 summarizes all presented aspects and describes remaining issues for future work.

2 Foundations

This chapter explains the fundamentals, necessary for understanding the remainder of this work. First, the concepts of a deployment model and deployment technologies are presented. This includes information on the concrete deployment technologies that are later used to validate the approach. In addition, the TOSCA [OAS13a] standard is presented, including a description of the OpenTOSCA [BEK+16] ecosystem, an open source implementation to model and manage TOSCA deployment models. Lastly, the concept of an instance model is described.

2.1 Automated Application Deployment

Modern enterprise applications are often complex distributed systems which consist of many components and make use of complex middleware [Dea07]. As a consequence, the complexity of deploying and managing these applications has increased as well [BBKL14b]. Oppenheimer [Opp03] investigated the causes of failures of such complex systems and came to the following conclusions: (i) Operating such complex applications manually is cumbersome and operator errors are a primary cause for failure and unavailability. (ii) Tools that visualize the components and their relationships may prevent such errors, as visualizations ease the understanding of complex applications. (iii) Automated processes also reduce errors, as automation prevents misconfigurations or issuing wrong commands.

Further, automation can not only reduce errors, but also cost by speeding up the execution of recurring management operations. This is especially important, with the advent of cloud computing, where computing resources can be acquired on demand with a pay-per-use model [LF09]. Highly automated deployment and management allows for fast creation and release of these cloud resources, minimizing their cost. Thus, providers of cloud resources, e. g., Amazon Web Services (AWS), provide APIs that can be integrated in automated deployment and management processes. However, manually defining dedicated deployment processes for every application, suffers the same issues as performing the deployment manually. As a result, many deployment technologies emerged, that assist in automating the deployment process for complex applications. Most of these deployment technologies require a *deployment model*, which has to be provided by the software engineers, and use that model to automatically deploy the application into the target environment.

2.1.1 Deployment Model

A deployment model of an application contains all information, necessary for deploying said application. There exists a plethora of different approaches on how this information is modeled. However, there are two main categories: *imperative* deployment models and *declartive* deployment models [EBF+17; WBF+19]. An imperative deployment model explicitly defines the complete

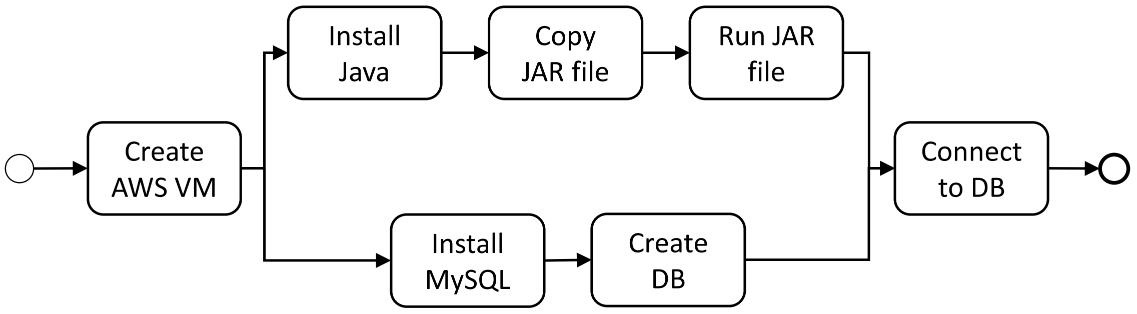


Figure 2.1: Example of an imperative deployment model

workflow to deploy the application, with all necessary steps in the correct order. Each step of the workflow describes a specific operation, e. g., a call to an API or the execution of a shell command. In Figure 2.1 an example for an imperative deployment model is shown. The depicted workflow first creates a VM on the AWS cloud. Once the VM is running, the workflow installs a Java Virtual Machine (JVM) and the MySQL Database Management System (DBMS) [Ora21] on the VM. Following the installation, the workflow copies a Java Archive (JAR) file to the VM and executes it. Moreover, the workflow creates a new MySQL database in the MySQL DBMS. The workflow specifies that, the setup of the Java Application and the MySQL database can run in parallel. However, the setup of the Java application and the database must be completed successfully, before the last step, that connects the Java application to the database, can be executed.

In contrast to an imperative deployment model, a declarative deployment model does not define the necessary deployment steps, but describes the components and their relations that make up an application. For example, a component can be a virtual machine, a database, or a web server. To model the dependencies between the different components, the declarative deployment model can include different types of relations: a service *connects to* a database, a webserver is *hosted on* a virtual machine or a service *depends on* another service. To specify additional information, components and relations can have properties defined, e. g., the port on which a web server should listen for incoming requests or the protocol which is used for a *connects to* relationship. An example declarative deployment model is shown in Figure 2.2. It specifies a VM that runs Ubuntu [Can21] as its OS which is indicated by the type *Ubuntu VM*. The VM is deployed onto the AWS cloud as indicated by the *HostedOn* relation between the *VM* and the *Public Cloud* component. Moreover, the VM is specified to have *8 GB* main memory, as indicated by the *ram* property. In addition, the VM hosts a Tomcat application server [The21a] that is listening on port 80 and in turn hosts a Java application. The Java application accesses a database which is indicated by the *ConnectsTo* relation between the *Webshop* and the *Database* component. The database is installed on the same VM as indicated by the respective *HostedOn* relations.

Most deployment technologies use a declarative deployment model, as it is deemed superior to imperative models [HAW11; WBF+19]. While imperative models offer more control over the deployment process, declarative models are more intuitive. In declarative models the operator simply defines the desired state, without considering the current state, which simplifies the task, reducing errors and cost. Thus, this work only focuses on declarative modeling.

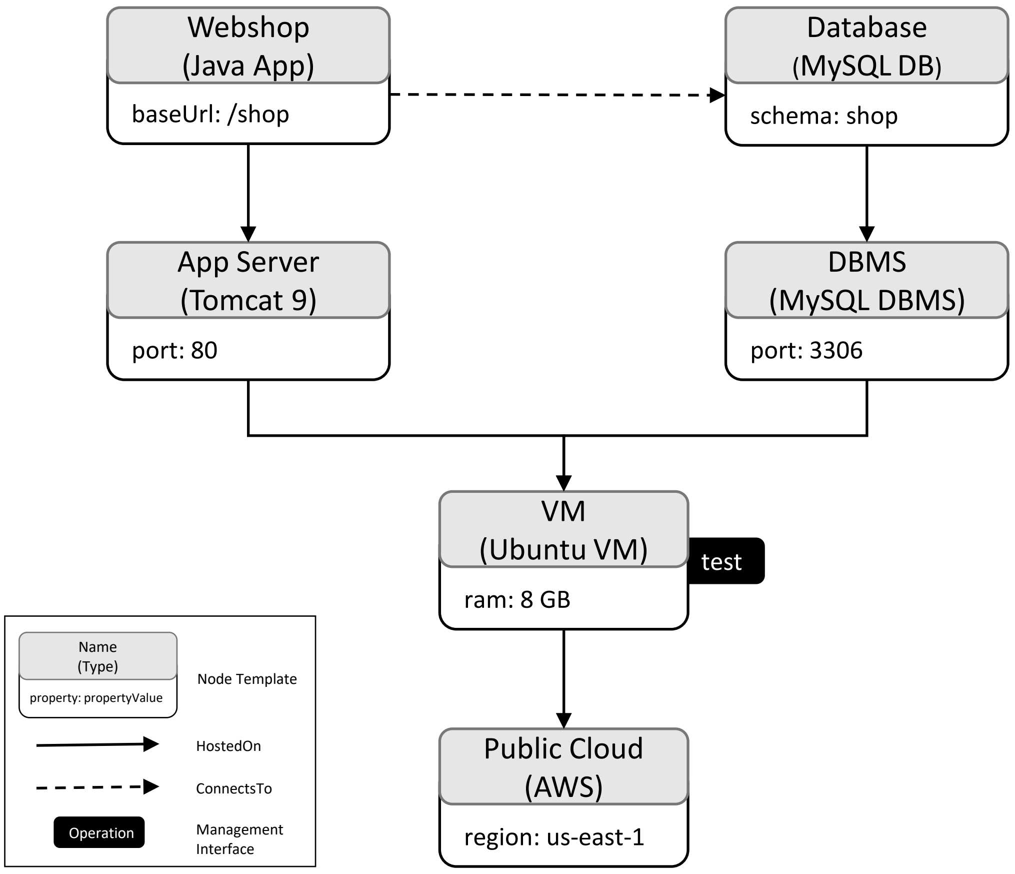


Figure 2.2: An example topology template

2.1.2 Deployment Technologies

In industry and research a plethora of deployment technologies has been developed. Most of them are based on the same principles, but differ in focus, as identified by Wurster et al. [WBF+19]. They conducted a survey and discovered three categories for deployment technologies: (i) provider-specific deployment technologies, (ii) platform-specific deployment technologies, and (iii) general-purpose deployment technologies. Provider-specific technologies are often developed by cloud providers and are mostly part of their Software as a Service (SaaS) portfolio. These technologies can be used to provision various resources the provider offers, e. g., creating VMs or allocate data storage. However, they are not able to provision resources at cloud providers. An example for such a technology is AWS CloudFormation [Ama21] which supports only AWS resources. Platform-specific technologies on the other hand, focus on specific technologies or platforms. While they support deployment to different cloud providers, they may limit the types of artifacts that can be deployed. Kubernetes [Clo21] is an example for such a technology, as it only provides the orchestration of containers. General-purpose technologies do not have any of the limitations provided above. They support all cloud resources as well as any cloud providers and thus, are very flexible. However, these technologies may still have a specific focus. For example, Terraform and Puppet are both general-purpose deployment technologies. Still, Terraform specializes on

Listing 2.1 Example definition of a Kubernetes *Pod*

```
kind: Pod
spec:
    containers:
        - name: Tomcat
          image: Tomcat:9.0.45
          ports:
              - containerPort: 80
```

orchestrating infrastructure resources, e. g., VMs or data storage. Configuring the provisioned resources, e. g., installing software on a VM, is technically possible but conceptually out of scope. Puppet, on the other hand, focuses on managing existing infrastructure, for example, by providing the possibility to edit configurations files or install software on an already running VM. These differences in functionality and scope can be a reason to use multiple deployment technologies side-by-side for a single application. For example, the infrastructure, e. g., VMs, is provided by Terraform, while Puppet is used to manage the software running on the infrastructure. In addition, different teams may prefer different deployment technologies. For example, two teams are responsible for different components of an application. While one team prefers to use AWS CloudFormation for provisioning VMs, the other team uses Terraform for the same task.

To cover all of the categories presented by Wurster et al. [WBF+19], this work investigates the following deployment technologies: (i) AWS CloudFormation as a provider-specific technology, (ii) Kubernetes as a platform-specific technolgy, (iii) Puppet and (iv) Terraform as general-purpose technologies. Puppet and Terraform are both selected as they have a different focus and complement each other, as described above. In the following, these deployment technologies are briefly explained, including their offered functionality, their focus and a brief description of how they operate.

Kubernetes

Kubernetes [Clo21] is a platform-specific, declarative deployment technology. It aims to orchestrate and manage containerized applications over a cluster of computing nodes, including autmoated scaling and load balancing, automated rollouts and rollbacks, self-healing as well as secret and configuration management [The21b]. Kubernetes employs a manager-worker architecture [The21d]. Multiple computing nodes, i. e., physical servers or VMs, may be joined to form a Kubernetes cluster. Every node in a cluster must run the *kubelet* service. The kubelet service connects the node to the cluster and exchanges information with the *control plane*, which acts as the manager. The control plane may be spanned over multiple nodes to ensure fault-tolerance and availability.

To deploy an application with Kubernetes, its services must be provided as containers. Containers can be seen as light-weight Virtual Machines that provide their own operating system but are executed as isolated processes on the host. To run containers, every computing node in the Kubernetes cluster must execute a container runtime, e. g., Docker [Doc21].

To describe the desired state of an application, Kubernetes uses YAML formatted text files. These files, are used to specify *Kubernetes objects* that should be deployed. The smallest deployable unit in Kubernetes is a *pod*, which defines a list of containers that should be run together. Listing 2.1

Listing 2.2 Example Puppet manifest to copy a file and execute it

```
node 'agent01' {
  file { 'app-file':
    ensure => 'file',
    source => 'puppet:///modules/app/app',
    path   => '/usr/local/bin/app',
  }

  exec { 'exec-app':
    command => '/usr/local/bin/app',
    require => [File["app-file"]]
  }
}
```

shows an excerpt of a pod definition as an example. The property *kind* specifies which type of Kubernetes object should be created, in this case a pod. The *spec* property contains the specification, i. e., all properties, of the pod. Here, a single container running a Tomcat application server is specified to be deployed inside the pod. A *Deployment* is a Kubernetes object that allows to scale one or multiple pods. A deployment definition references pod definitions and defines additional properties. For example, the deployment defines the replication strategy for pods. The replication strategy determines if and how a pod should be replicated to handle increased workload. There are more kinds of Kubernetes objects, however these are out of scope for this work.

The control plane of a Kubernetes cluster offers an API which can be used to upload definition files or to update existing ones. The control plane then analyzes the files and derives and executes any necessary action, to apply the specified changes. This API can also be used to retrieve information about currently deployed Kubernetes objects, like pods or whole deployments.

Puppet

Puppet [Pup21a] is an agent-based, general-purpose, declarative configuration management tool. Its focus is to provide automated configuration management of running infrastructure, e. g., installing software on a server or manipulate configuration files [Pup21b]. Puppet uses a primary-secondary architecture, where every server that should be managed by Puppet must have the Puppet agent installed on it and the agent must be connected to the primary server. The primary server is responsible for managing the configuration of the agents it controls. Puppet uses a declarative approach for the deployment model, which can be defined in *manifest* files. These manifests are text files that contain declarations of required resources using a custom Domain-Specific Language (DSL) [Pup21c]. Each declared resource, has a type, a name and several properties. For example, the manifest shown in Listing 2.2 specifies two resources. First, a file to be copied from the primary server to the node with the logical name “agent01”. Here, the resource is of type *file* and has the name “app-file”. Moreover, the *source* property specifies that the file contents can be found on the primary server, while the *path* property specifies where the file should be copied to on the agent node. The second resource of type *exec* specifies that the previously copied file should be executed. The file can only be executed, if the copy operation was successful, thus, the exec resource specifies

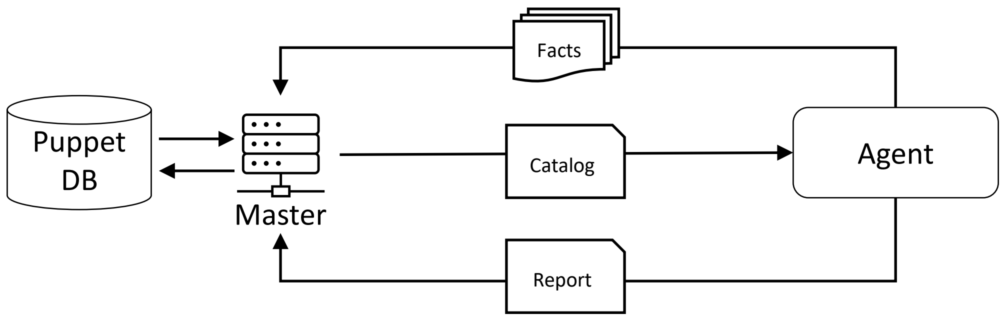


Figure 2.3: Puppet workflow, based on [Pup21b]

Listing 2.3 Example template to create an EC2 Instance with AWS CloudFormation

```
Resources:
    AppInstance:
        Type: 'AWS::EC2::Instance'
        Properties:
            ImageId: ami-071a13877ce8467d4
            InstanceType: t2.micro
```

this relationship in the *required* property. There exists a plethora of other resources and users are also able to define their own resource types. A manifest can apply to only one node, like in the example, or to multiple nodes. Additionally, multiple manifests may target the same node.

The process of deploying changes using Puppet is depicted in Figure 2.3. First, whenever an agent connects to the primary server, it sends *facts* about its current state to the primary server. The facts are any information that might be relevant for the primary server, like the hostname or the list of installed packages. The primary server uses these facts together with all applicable manifests, to generate the *catalog* which expresses the desired state of the agent node. The catalog is retrieved by the agent, that computes and applies all necessary changes on the managed node. After applying the catalog, the agent sends a *report* to the primary server. The report states whether each specified resource was successfully created or if an error has been encountered. The primary server stores the facts, catalogs and reports for each managed node in a database, the *Puppet DB*. The information stored in the database can be queried over an API.

Amazon Web Services (AWS) CloudFormation

AWS CloudFormation [Ama21] is a provider-specific, declarative deployment technology. It only supports provisioning of AWS cloud resources, e. g., Elastic Compute Cloud 2 (EC2) instances or an Elastic File System (EFS). To model the desired state, AWS CloudFormation uses a declarative model, called *template*, that can be defined using either YAML or JSON. Similar to Puppet, the model specifies the resources which shall be provisioned. Every resource has a name, a type, and a set of properties describing the desired state for each resource. An example template, creating an EC2 instance, is shown in Listing 2.3. The list of *Resources* contains a single entry with the name

Listing 2.4 Excerpt of a Terraform file to create a AWS EC2 instance

```
provider "aws" {
  region     = "eu-central-1"
}

resource "aws_instance" "ec2-instance" {
  ami           = "ami-071a13877ce8467d4"
  instance_type = "t2.micro"
}
```

AppInstance. The type *AWS::EC2::Instance* indicates that AWS CloudFormation should start a VM on the EC2. The properties define more details for the VM. The *ImageId* defines the image containing the OS that should be used and *InstanceType* defines the computational resources, i. e., processor and memory, that should be allocated for the VM. To ease the template creation, AWS CloudFormation offers a visual designer.

AWS CloudFormation is provided as SaaS by AWS. Thus, a template must be uploaded either via the User Interface (UI) of the AWS Console, the exposed Representational State Transfer (REST) API, or the AWS Command Line Interface (CLI). To instantiate an uploaded template, AWS CloudFormation creates a *stack* that bundles all provisioned resources. Thus, a single template may be used to create multiple stacks. For every stack, AWS CloudFormation analyzes the respective template and derives the necessary operations, e. g., creating or modifying a resource. As it is operated by AWS and only supports AWS resources, AWS CloudFormation can utilize internal APIs to execute these operations.

Terraform

Terraform [Has21b] is a general-purpose, declarative deployment technology. Its focus is to provide the automated provisioning of infrastructure resources, e. g., networking or computing resources [Has21b]. However, the configuration management of the provisioned resources, for example installation of software on a VM, is out of scope for Terraform and should be handled by other tools, e. g., Puppet. In comparison to the previously presented technologies, Terraform does not require a running service. It provides a CLI that is used to analyze a declarative model and to generate a provisioning plan. To support resources of different providers, the Terraform CLI uses a plugin system. Each plugin specifies how resources can be defined in the Terraform model and provides implementations to provision the resources at the respective provider.

Similar to Puppet, Terraform defines its own DSL to specify resources that should be provisioned. Listing 2.4 shows an example Terraform file that can be used to provision an AWS EC2 instance. The file specifies to use the *provider* “aws”, which causes Terraform to load the AWS plugin and its respective resource types. The provider configuration also specifies that all AWS resources should be provisioned in the “eu-central-1” region. The actual EC2 instance is defined as a *resource* of type “aws_instance” and the name “ec2-instance”. The resource block also defines the properties of the instance that should be provisioned. The *ami* property defines the image that should be used, while the *instance_type* property defines the computational resources, i. e., processor and main memory that should be allocated for the instance.

Using the Terraform CLI, the Terraform file can be applied, which leads to all defined resources being created. After each run of Terraform, the current state is stored in a state file. The state file can be used to compare the desired state with the current state and to make only necessary changes. The state file can also be parsed by third parties, to retrieve instance information about the resources that were deployed using Terraform.

2.2 Technology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA is a standard by OASIS to describe the deployment and management of cloud applications in a portable and interoperable way [OAS13a]. The standard aims to provide a model of the application that contains information on (i) how the application should be deployed and (ii) how specific management operations can be executed. The main central construct of the standard is the *service template*, that contains all necessary information for a single application. The service template has two main parts: a *topology template* and *plans*. In addition, a service template can have any number of *tags*, key-value pairs, that provide additional description of the template. The topology template is a declarative deployment model, as described in Section 2.1.1, and defines the structure of the application in form of a directed and weighted graph. The graph consists of *node templates* as its nodes and *relationship templates* as its edges. A *node template* represents a component of the application, e. g., a VM, the operating system, or a database. A *relationship template* represents a connection between two application components, e. g., the connection between a service and a database. To specify the semantics, every node template and relationship template has a reusable type, i. e., a *node type* or a *relationship type*, assigned to it. A node type defines the *properties* and *management operations* of an application component. An example for such a node type is a *MySQL DBMS* type. It may specify properties, like the port on which the server should listen or the password for the root user. Each node template that has the *MySQL DBMS* node type assigned can provide values for the properties specified by the node type. Moreover, the *MySQL DBMS* node type might define a management operation *test* that can be executed to check the availability of the DBMS. Relationship types define the semantics of a relationship between two components and can define properties for the relationship. There are three normative types defined for TOSCA [OAS13b]: *HostedOn*, *ConnectsTo* or *DependsOn*. A *DependsOn* relationship defines that the source component is dependent on the target component. This means, that the source component cannot run, if the target component is unavailable. A *HostedOn* relationship defines, that the source component is installed on the target component and runs in the context of the hosting component. An example is a Tomcat application server that must be installed on some sort of computing component, e. g., a VM. A *ConnectsTo* relationship defines, that the source component in some way connects to the target component. An example for this is a service which connects to a database.

Figure 2.2 shows an example topology template of a simple web shop application. Every node template has a name and a type, e. g., the node template, that represents the business logic component of the web shop, is called "Webshop" and is of type *Java App* as it is a Java application. The Java application should be *HostedOn* an application server of type *Tomcat 9*, which is defined to listen on port 80. The Tomcat server in turn should be *HostedOn* a VM of type *Ubuntu VM*, as it uses Ubuntu as its operating system. The VM is defined to run in the region *us-east-1* of the AWS

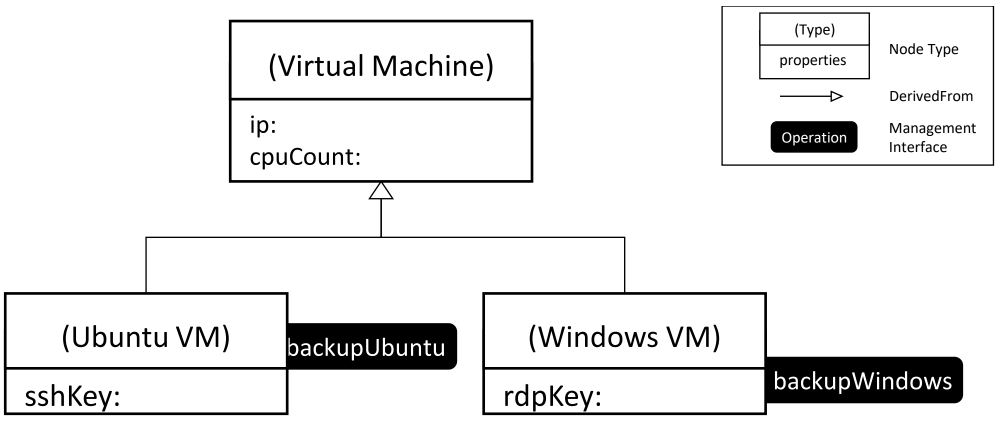


Figure 2.4: An example type hierarchy

public cloud, thus, its node template is connected with a *HostedOn* relationship to the public cloud node template of type *AWS*. To store and retrieve data the "Webshop" component *ConnectsTo* a database of type *MySQL DB* which has the schema name *shop*. This database should be *HostedOn* a *MySQL DBMS*, listening on port 3306. The database server should again be hosted on the same VM as the Tomcat server. In addition, the *Ubuntu VM* type defines a management interface with the management operation *test*. This operation can be called to check the availability of the VM.

To execute these management operations, the service template includes *plans*. Plans are process models which describe a workflow that contains all the steps necessary to execute the management operations in a service template. A plan can be parameterized and take input values from the properties specified in the topology template. Moreover, a plan can execute any number of management operations at the same time. For example, if multiple node templates in a service template specify a *restart* operation there might be a single plan that restarts all components at the same time. To ensure interoperability and portability, the TOSCA plans should be specified in standardized process languages, e. g., Business Process Modeling and Notation (BPMN) [Obj10] or Business Process Execution Language (BPEL) [OAS07]. However, plans may include the execution of arbitrary code supplied as *implementation artifacts*, e. g., a shell script.

As mentioned above, node types are reusable, i. e., multiple node templates may have the same type assigned. The same is true for relationship types and relationship templates. To improve reusability, node types and relationship types can be stored in a dedicated repository and be used in multiple service templates. For example, an organization might develop multiple Java applications. Thus, the type *Java App* will be used in almost every service template. Storing the types in a organization-wide repository prevents repeated tasks and supports knowledge transfer across the organization. Further, the TOSCA standard employs inheritance for types. A node type or relationship type inherits all elements of a super type, e. g., properties or interfaces, by defining a *DerivedFrom* clause. This is especially useful for semantically similar types that share many properties. An example type hierarchy for VMs is depicted in Figure 2.4. The hierarchy considers the types *Ubuntu VM* and *Windows VM*. Both types do have several common properties, e. g., their Internet Protocol (IP) address or the count of processors. However, the *Ubuntu VM* specifies the additional *sshKey* property, while the *Windows VM* specifies the *rdpKey* property. The values of the two additional properties are required to connect to the respective VM. Moreover, the backup

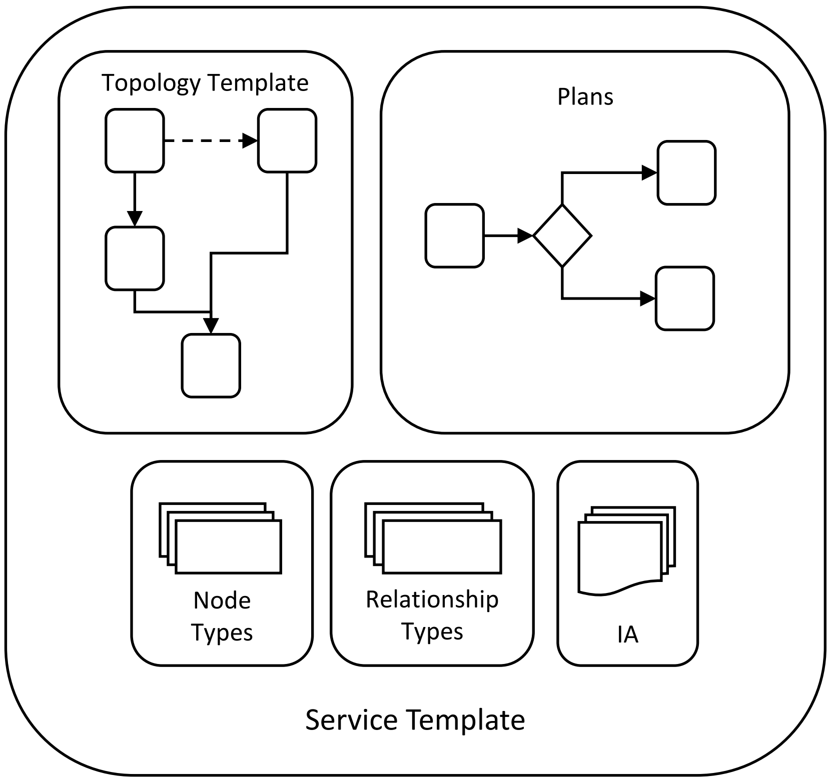


Figure 2.5: Structure of a CSAR

management operation must be executed in different ways for both node types. Thus, both types inherit from a common super type *VM* that defines the common properties but define their own management operations, *backupUbuntu* and *backupWindows* respectively.

To actually deploy an application modeled with TOSCA, the service templates are packed in a Cloud Service Archive (CSAR). The CSAR contains one or more service templates and all necessary information to deploy them, as depicted in Figure 2.5. Beside the topology template and the plans, this includes the definitions of all node types and relationship types used in the topology template. Further, all Implementation Artifacts (IAs) are part of the CSAR, so that the specified plans can be executed correctly. To deploy the application the TOSCA runtime may process the CSAR either in an imperative or in a declarative way. For imperative processing the CSAR must contain explicitly specified plans to provision the application. If they are not provided, the imperative processing fails. In case of declarative processing, however, the TOSCA runtime tries to derive the necessary steps for deployment from the contained topology template, analogously to the behavior defined in Section 2.1.2.

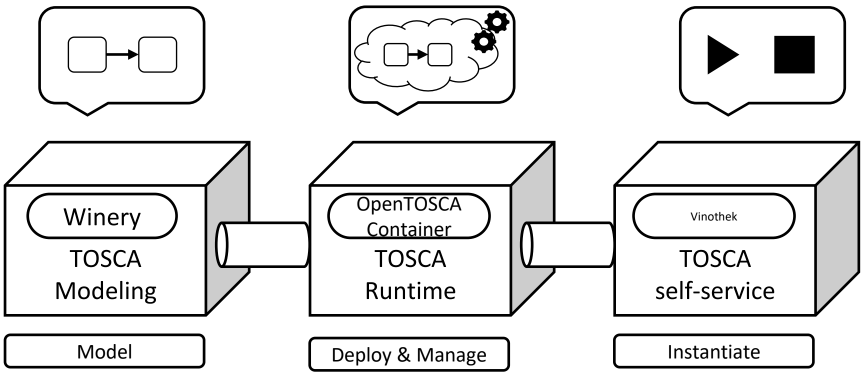


Figure 2.6: Overview of the OpenTOSCA ecosystem. Based on [BEK+16; Mat20]

2.3 OpenTosca

OpenTOSCA [BEK+16] is an ecosystem providing the possibility to model, deploy, manage, and instantiate cloud applications defined using the TOSCA standard as described in Section 2.2. It consists of three main components as depicted in Figure 2.6: (i) *Winery* [KBBL10], (ii) *OpenTOSCA Container* [BBH+10], and (iii) *Vinothek* [BBKL14c]. Winery is a modeling tool, that helps to create TOSCA models by providing a comprehensive UI. In addition, Winery incorporates a type repository that allows defining and reusing node types and relationship types in multiple service templates. The service templates created in Winery, can be exported in the CSAR format, either using the UI or the Winery API. The exported CSAR files can be loaded into the OpenTOSCA Container, which is a fully TOSCA compatible runtime. The OpenTOSCA Container is responsible for provisioning the modeled application. To achieve this, it incorporates a plan generator, a plan engine, and an IA engine. The plan generator analyzes a topology template and builds provisioning, termination and management plans for the specified application. The plan engine takes care of executing all plans contained in the CSAR or generated by the plan generator. This includes the provisioning plans, when instantiating the application, as well as any on-demand management plans that are requested by the user. The IA engine is used to run the implementation artifacts, that are part of any plans. The Vinothek is a self-service portal that lists all service templates, i. e., applications, that are available in an OpenTOSCA container. In this portal, the user can see all applications that have been installed into the OpenTOSCA Container. Moreover, the user can execute the provisioning plan to start an application. For already instantiated and running applications, the user can check the status and request the execution of available management operations, which triggers the execution of the corresponding plan by the plan engine inside the OpenTOSCA Container.

The typical workflow to deploy an application in the OpenTOSCA ecosystem is as follows. First, a service template is created using the modeling tools in Winery. Second, the created model is exported as a CSAR and installed into the OpenTOSCA Container. Third, the user instantiates the created application in Vinothek, triggering the provisioning of the deployment by the OpentTOSCA container. Fourth, while the application runs, the user monitors and manages it, using the management operations provided in Vinothek.

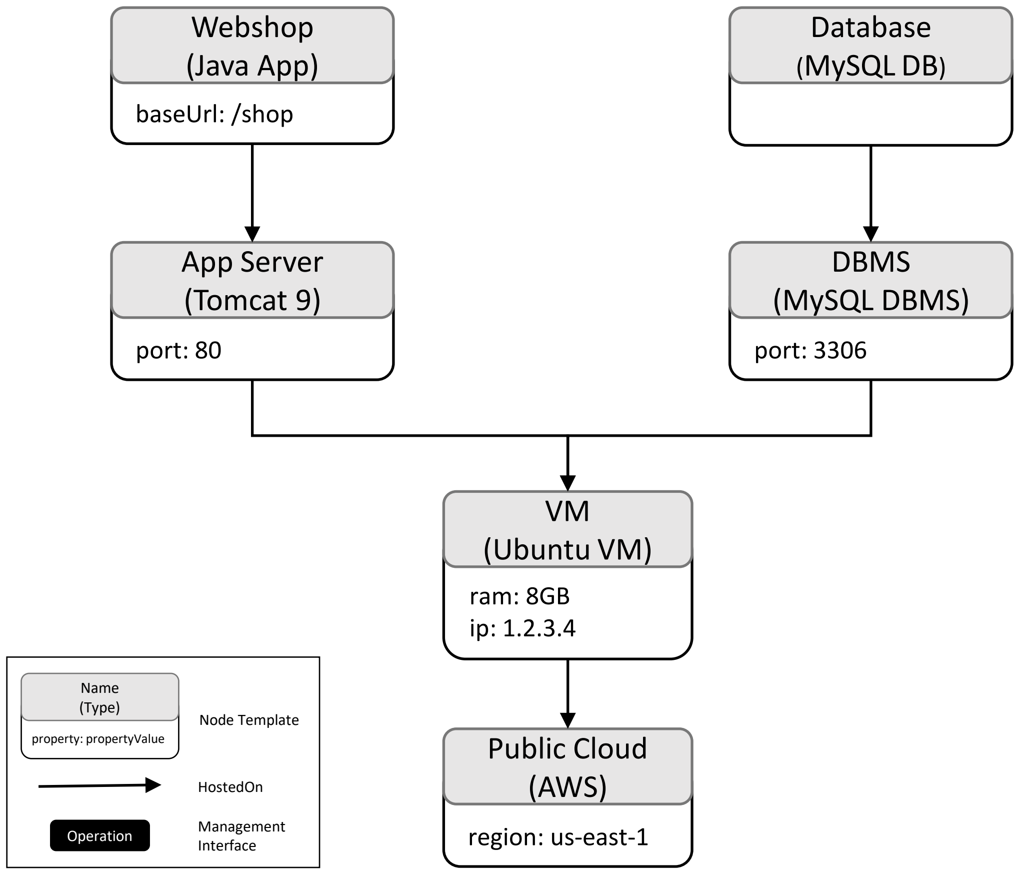


Figure 2.7: Example instance model

As the OpenTOSCA ecosystem can be used for every step of application deployment – from modeling to management – many research work has been done, to enhance the ecosystem with useful functionality [BBKL13b; HBB+21; WBK+20]. Thus, the prototype to retrieve instance models and to generate management workflows is implemented as part of the OpenTOSCA ecosystem.

2.4 Instance Model

Similar to a declarative deployment model, an *instance model* also describes the state of an application. However, instead of showing the desired state, it represents the current state of all components and their relations. The instance model can be used for documentation purposes (for an example see [BBKL13a]) or for management of the modeled application (for an example see [BBKL14a]). For example, the information contained in the instance model can be used to check the availability of the running components. Many parts of the instance model of an application are similar to the deployment model of the same application. For example, a complete instance model will describe the same components and the same relations as the deployment model. Moreover, many properties of the components and relations will be identical. An example is the *port* property, describing the port a web server listens on. The deployment model specifies the *port*, so that the deployment technology that deploys the application can configure the web server appropriately. The instance model also contains the *port* property, as its value is necessary to issue requests to the web server. However, instance models may also differ from the deployment model. For example, the IP address of a VM is often known only after its deployment. Nonetheless, the IP address is vital information for any management operation that needs to connect to the VM. In addition, instance models may not be complete. Depending on the method that is used to create the instance model, different detail of instance information is available. For example, an error in the manual creation of an instance model can lead to missing information. Also automatically generated instance models can miss information, for example, if the retrieving tool cannot connect to some components. As discussed by Binz et al. [BBKL13a], the TOSCA standard can be used to specify instance models. Thus, this work uses TOSCA instance models. Figure 2.7 shows an example instance model for the application from Figure 2.2. The instance model contains the same components as the deployment model. However, the *Ubuntu VM* component specifies the additional property *ip*. Moreover, the *schema* property of the *MySQL DB* component is missing, as is the *ConnectsTo* relation between the *Java App* and the *MySQL DB*.

3 Related Work

The retrieval of information about running applications has been the topic of many research. Especially in the field of Enterprise Architecture Management (EAM), many approaches have been proposed. For example, Farwick et al. [FAB+11] and Holm et al. [HBLE14] presented approaches for the automated generation of enterprise architecture documentation. Farwick et al. [FAB+11] aimed to improve the maintenance of existing enterprise architecture models. To achieve this, they propose an automated process that repeatedly queries defined data sources to adjust the existing model. These sources may be databases, APIs or even human input. From the received data, the maintenance process extracts the running application components, their relevant attributes, as well as, the relationships between them. Holm et al. [HBLE14] rely on network scanners to retrieve information about running applications inside the enterprise network. These scanners analyze the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic of the target network and extract information. Such information can be the hosts transmitting over the network, their OSs and installed services. Using source and target addresses the relationships between services can also be identified. From this information Holm et al. [HBLE14] generate an ArchiMate [The19] model that can be refined manually. The approaches of Farwick et al. [FAB+11] and Holm et al. [HBLE14] both aim at providing documentation of running applications as part of the EAM process, while this work focuses on providing automated management operations for single applications. Moreover, none of the approaches explicitly considers deployment technologies.

Further, Machiraju et al. [MDW+00] describe a generic approach for application discovery. Their "generic auto-discovery engine" uses application template models to search for instances of the defined application. The template describes what to discover and how to discover it. The engine then searches possible instances of applications, that fit the template. However, the template model requires the discoverable components to be defined in advance and thus, limits what application components can be found. This does not fit the approach in this work, as this work tries to discover arbitrary application components.

Brogi et al. [BCS17] introduced Discovering Available Cloud Offerings (DrACO), a tool to discover Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) offerings and model these as TOSCA node types. The tool retrieves all necessary information and adds a node type to the TOSCA repository. This node type can later be used in TOSCA deployment models to deploy application components onto said IaaS or PaaS offerings. This approach does only allow for the discovery of single node types for already known services. However, this work aims at providing a complete instance model for running applications that are composed of different components.

Binz et al. [BBKL13a] investigated the automated generation of Enterprise Topology Graphs (ETGs). An ETG is a model of all components, that are running on an IT infrastructure of an enterprise, and the relations between them. The presented approach starts on a manually provided entry part of the ETG. A plugin-based crawler framework extends the initial ETG in multiple iterations by performing arbitrary operations against the discovered software and hardware components.

The framework consists of dedicated plugins for specific component types, which may discover dependent components or refine the information on already discovered components. Although the concept of an ETG seems similar to an instance model as described in this work, it has a larger scope. The ETG aims to represent the entire landscape of enterprise IT, while the instance model is specific to an application.

In later work Binz et al. [BBKL14a] utilized the ETG to migrate applications to and between cloud environments. They proposed an approach to extract a sub-graph of the ETG, representing a single application and map its components to TOSCA types. The resulting deployment model is then modified to target a specific cloud environment, e.g. AWS. This approach defines an automated process to retrieve TOSCA models of running applications. However, it focuses on migrating an application between two environments, while this work aims at providing management operations for the application without changing its environment.

Brogi et al. [BFGL20] also proposed Managing Applications Running In Opportunistic Fog (MARIO), an automated approach for managing distributed applications. The approach uses declarative policies, which are defined manually for specific applications. MARIO monitors the distributed environment, the application is running in, and selects the best computing node for each service, according to the specified policies. As the environment may change, MARIO is able to select a new computing node and to move the service to said node. Similar to Binz et al. [BBKL14a], Brogi et al. [BFGL20] focus on moving parts of the application, while this work intents to enhance running components without moving them. Moreover, their approach is specific to fog environments.

Harzenetter et al. [HBL+19] introduced the Management Feature Enrichment and Workflow Generation (MFEW) method. This method enriches existing TOSCA deployment models with management features and creates executable workflows in an automated process. This is achieved by matching the node types of the modeled application with *Feature Node Types*. For example, a *Feature Node Type* for a MySQL database may specify a backup operation. If the deployment model contains a node with a matching MySQL node type, the management feature can be applied to the model by replacing the original node type. The replacement node type is automatically generated by merging the original node type and the *Feature Node Type*. As this process can be executed iteratively, multiple management features can be applied to the same node in the deployment model. The resulting deployment model can then be used to automatically derive executable workflows in form of TOSCA management plans. The drawback of this approach is, that it requires an existing deployment model as input. However, the approach can be modified to operate on instance models, and thus, can be reused for this work.

As described in Section 2.1.2, most deployment technologies define their own declarative modeling language. However, they all share some basic concepts e. g., components and relations. Wurster et al. [WBB+20] introduced the Essential Deployment Meta Model (EDMM) modeling and transformation system. It allows to create deployment models in a deployment-technology-agnostic metamodel, the EDMM, and offers functionality to transform these models to deployment-technology-specific models. This prevents vendor lock-in effects for applications modeled with EDMM. Based on that, Mathony [Mat20] proposed Essential Deployment Meta Model instance (EDMMi), a technology-agnostic metamodel, to define instance models. In EDMMi a *deployment instance* describes the current state of an application and consists of *component instances* and *relation instances*. All component instances and relation instances have a *component type* or a *relation type respectively*. Moreover, every instance can have multiple *instance properties*. In addition, Mathony [Mat20] provided the Instance Model Retrieval Framework to automatically retrieve EDMMi models from the APIs of different deployment technologies and showed, that EDMMi models can be trivially mapped to TOSCA instance models: a *deployment instance* becomes a *service template*, while *component instances* and *relation instances* can be mapped to *node template* and *relationship template* respectively. Also, the *component types* and *relation types* can be mapped to their respective TOSCA types. As a consequence, this work skips the creation of EDMMi models, but uses the mapping information, provided by the Instance Model Retrieval Framework, to directly retrieve TOSCA instance models from the information provided by the deployment technologies. Mathony [Mat20] also use the TOSCA instance models to automatically generate management workflows, but their approach is limited to a single deployment technology.

To overcome the drawback of their previous approach, Harzenetter et al. [HBB+21] provided a method to retrieve instance models for running applications. Similar to Mathony [Mat20], they utilize the APIs of the deployment technology, used to deploy the target application, to gather instance information. The *Instance Information Retriever* uses specialized plugins for every supported deployment technology, that collect instance information from the APIs of the deployment technologies. In contrast to Mathony [Mat20], Harzenetter et al. [HBB+21] directly map the retrieved instance information to a TOSCA instance model. In addition, the retrieved instance model contains information about the deployment technology itself. This way, the management workflow, generated in a later step, can access this information in order to prevent interference in workflow execution. Different deployment technologies, however, provide data of different granularity and expressiveness over their APIs. Thus, Harzenetter et al. [HBB+21] included an *Instance Model Completer* that retrieves additional information about the discovered components. Hereby, the completer consists of several plugins that may perform arbitrary operations, e. g., sending Hypertext Transfer Protocol (HTTP) requests or issuing shell commands over Secure Shell (SSH) connections. The resulting instance model can be enriched with management features, using the adapted approach from Harzenetter et al. [HBL+19]. This approach constitutes the basis of this work, however it only supports instance model retrieval from a single deployment technology, similar to Mathony [Mat20].

4 Concept

This chapter presents the approach, this work introduces. The goal is, to retrieve a single instance model for an application that has been deployed using multiple deployment technologies. The instance model is based on the TOSCA standard and is used to automatically generate management workflows that perform management operations on the discovered instances. The remainder of this chapter first provides an overview of the complete process from instance model retrieval to management workflow execution in Section 4.1. Following, Section 4.2 describes how the involved deployment technologies are discovered. Section 4.3 explains how instance information can be retrieved from the APIs of multiple deployment technologies, while Section 4.4 describes the approach to complete the instance model with information that is not available from the given deployment technologies. Lastly, Section 4.5 describes how the retrieved instance model can be enriched with management features and how the enriched model can be used to generate management workflows.

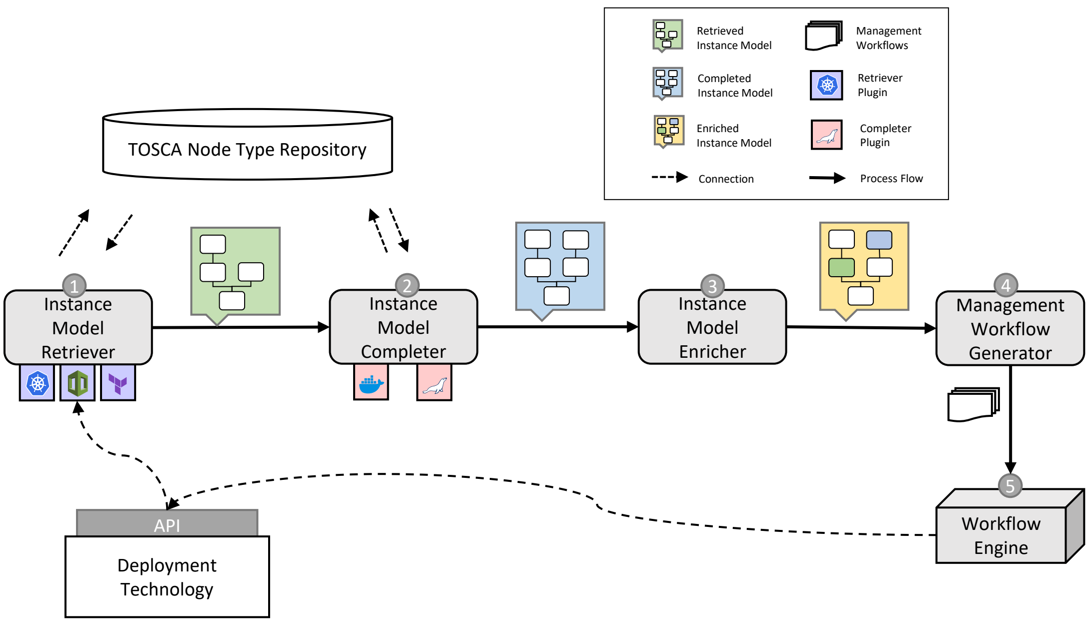


Figure 4.1: Mapping of Kubernetes entities to TOSCA entities

4.1 Overview

This work proposes a process for the retrieval of an instance model and the generation of management workflows depicted in Figure 4.1. The process is made up of five components: (i) the *Instance Model Retriever*, (ii) the *Instance Model Completer*, (iii) the *Instance Model Enricher*, (iv) the *Management Workflow Generator*, and (v) the *Workflow Engine*. The order in which the components are invoked are depicted by the numbers on each component. The first component, the Instance Model Retriever, is a plugin-based component that retrieves instance information from the deployment technologies and derives a TOSCA instance model. In order to map the retrieved instance information to TOSCA types, the Instance Model Retriever needs access to the TOSCA node type repository. For every supported deployment technology the Instance Model Retriever has a dedicated plugin, as depicted by the deployment technology icons in the figure. Each of these plugins is responsible for establishing a connection to the deployment technology and to map the provided instance information to the derived TOSCA instance model. A detailed description of the Instance Model Retriever is provided in Section 4.3.

The retrieved instance model is handed over to the Instance Model Completer. The Instance Model Completer is also a plugin-based component that tries to refine the instance model with additional information that cannot be queried from the deployment technologies. For this, the Instance Model Completer uses the information provided in the retrieved instance model as a starting point to gather additional information. For example, the Instance Model Completer may connect to the API of an already discovered component and retrieve additional information about the component. In order to refine the node types of the TOSCA model, the Instance Model Completer also needs access to the TOSCA node type repository. More details on the Instance Model Completer are provided in Section 4.4.

The completed instance model is handed over to the Instance Model Enricher which is responsible for enriching the contained components with management features. To achieve this, the Instance Model Enricher uses the concept of management feature node types. Each management feature node type extends an existing TOSCA node type and defines a single management feature for it. The original node type in the instance model can be replaced with a generated node type that represents a combination of the original node type and one or more management feature node types. The process of management feature enrichment is explained in more detail in Section 4.5.

The enriched instance model is given to the Management Workflow Generator, which is responsible for generating executable management workflows. The Management Workflow Generator generates a single plan for each management feature contained in the instance model. For example, if two components in the instance model are enriched with the *test* management feature, a single workflow will be generated which invokes the *test* operation on both components subsequently. As the invocation of management features may require inputs, e. g., network addresses or credentials, the Management Workflow Generator also extracts the necessary information from the instance model and provides the invocation with concrete values. The generated management workflows are deployed onto the Workflow Engine and can be executed on demand. More details on workflow generation are provided in Section 4.5.

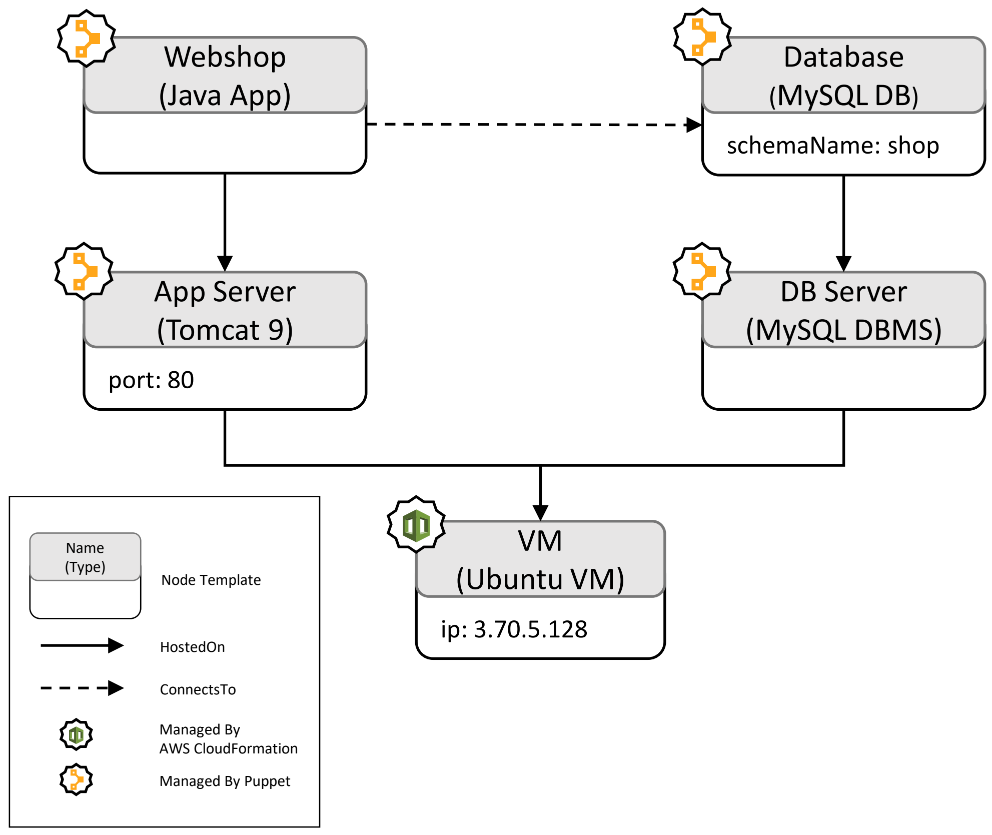


Figure 4.2: Running example

4.1.1 Running Example

For a better understanding the application depicted in the TOSCA topology in Figure 2.2 will be used as a running example throughout this chapter. It consists of a Java web application that is hosted on a Tomcat application server. The Java web application connects to a MySQL database which is hosted on a MySQL DBMS. The Tomcat application server and the MySQL DBMS are hosted on the same VM which is running Ubuntu as its OS. As indicated by the icon in the upper left corner of each node template, the VM is deployed by AWS CloudFormation, while all other components are deployed using Puppet.

4.2 Discovering involved Deployment Technologies

This work focuses on retrieving instance information from the APIs of deployment technologies, without any knowledge about the target application beforehand. Thus, the first step is to determine the concrete deployment technologies that are involved in deploying the target application. For the running example, the usage of AWS CloudFormation and the Puppet primary node must be discovered. One possible approach is, to automatically discover all deployment technologies that are used inside an organization. This could be achieved by utilizing network scanners, described by [HBLE14], or to extract the information from existing enterprise architecture documentation, like ETGs [BBKL13a]. However, this approach is insufficient, as network scanners are not able to discover deployment technologies that are running outside the enterprise network, e. g., AWS CloudFormation, or that do not need a continuously running service, e. g., Terraform. Moreover, most deployment technologies require additional information or some sort of credentials, to access their APIs. For example, the AWS cloud is split into independent regions. For accessing the AWS CloudFormation API the correct region must be known. Credentials are even impossible to discover automatically, as the nature of credentials is to be kept secret. Examples for such credentials are either private keys, e. g., for Puppet or Kubernetes, or access tokens, e. g., for AWS CloudFormation.

In addition, discovering all deployment technologies used by an organization is only a first step. As the organization will, most likely, operate several independent applications, only these deployment technologies must be selected that are actually involved in deploying the target application. For example, consider an organization that deploys the running example application. Beside AWS CloudFormation and Puppet, the organization might also use Terraform to provision VMs. When retrieving instance information for the running application, it is impossible to automatically determine if Terraform or AWS CloudFormation are used for deploying the VM in the running example application, without prior knowledge about the application.

As a consequence, this work proposes to manually define the list of involved deployment technologies as a starting point to discover instance information. Each entry in the list specifies a unique identifier for the entry and a list of properties, necessary for connecting to the API, e. g., credentials. Each entry must have a unique identifier, as the list might contain multiple entries with the same deployment technology. This is the case, if an organization runs multiple instances of a deployment technology, e. g., two independent Puppet primary nodes. The manual configuration is reasonable, despite the drawbacks of manual processes described by Oppenheimer [Opp03]. First, the effort of compiling this list should be manageable, since the list of involved deployment technologies will be significantly shorter, than the list of application components. While an enterprise application may consist of many components, there are only a couple of deployment technologies involved. Moreover, the list will be stable over the lifecycle of an application, compared to the components the application consists of.

4.3 Instance Model Retrieval

Given the list of deployment technology instances as specified in Section 4.2, the Instance Model Retriever retrieves a TOSCA instance model. The Instance Model Retriever reuses concepts from the *Instance Information Retriever* and the *Instance Model Normalizer* of Harzenetter et al. [HBB+21], but combines them into a single component. For every supported deployment technology, the Instance Model Retriever has a dedicated plugin, which is responsible for connecting to the API and to retrieve a TOSCA instance model. For example, the Puppet plugin is responsible for connecting to a Puppet primary server and to retrieve information about the running resources, managed by that primary server. As this work retrieves instance models from multiple deployment technology instances, one run of the Instance Model Retriever will include multiple plugin runs. In every plugin run, the plugin connects to the respective deployment technology instance and retrieves instance information. The retrieved information is mapped to a TOSCA instance model. As this mapping is specific to each deployment technology, Section 4.3.1 describes the mapping in detail

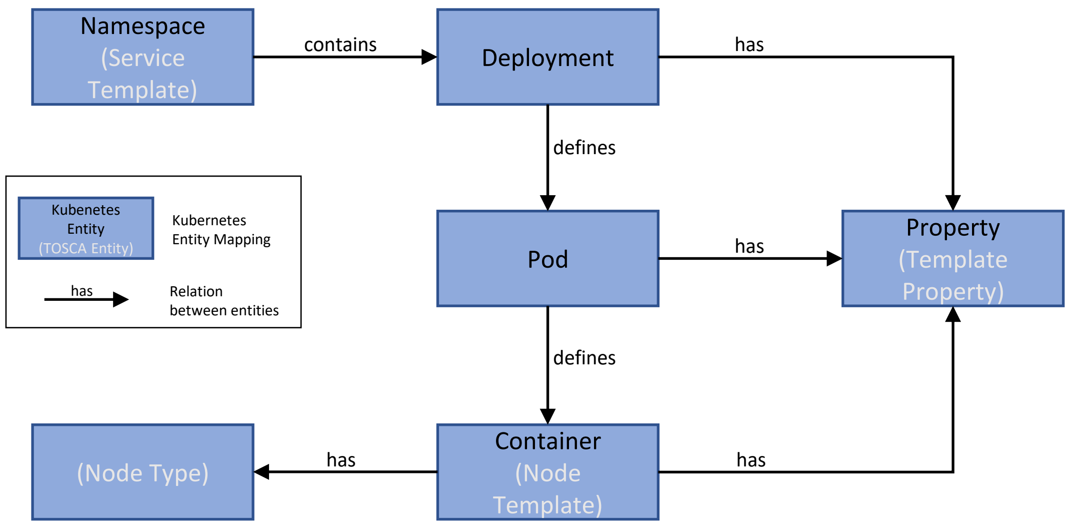

Namespace
(Service Template)
contains
Deployment
has
defines
Kubenetes Entity (TOSCA Entity)
Kubernetes Entity Mapping
has
Relation between entities
Pod
has
Property (Template Property)
defines
(Node Type)
has
Container (Node Template)
has

Figure 4.3: Mapping of Kubernetes entities to TOSCA entities

for every deployment technology investigated by this work. In addition, the information about the involved deployment technologies must be added to the instance model, to allow the generation of management workflows that need to connect to the deployment technologies. Section 4.3.2 describes how this can be achieved in TOSCA models. Joining the instance models produced by each plugin run into a single instance model is a complex task and is detailed in Section 4.3.3.

4.3.1 Mapping from Deployment Technologies to TOSCA Instance Models

Each plugin of the Instance Model Retriever is responsible for mapping the information provided by the deployment technology to the respective entity in a TOSCA instance model. This section provides conceptual mapping information for all deployment technologies mentioned in Section 2.1.2. Technical details on the respective plugin implementation are presented in Chapter 5.

As described in Section 4.2, a single instance of a deployment technology may be used to deploy multiple applications. Thus, it is necessary to filter the instance information retrieved from the deployment technology to only include information on the components of the target application. For example, AWS CloudFormation can be used to deploy VMs for other applications, in addition to the application from the running example. Thus, when querying the AWS CloudFormation API, it must be ensured, that only the information about the Ubuntu VM of the example application is added to the instance model. There is no standardized way of logically separating components of different applications in deployment models. However, most deployment technologies either provide technical concepts to achieve this separation or suggest workflows to achieve the separation on the process level. Thus, this section also describes for each deployment technology how the relevant components can be isolated.

Mapping Kubernetes to TOSCA

Kubernetes is used to orchestrate the deployment of containerized applications in a cluster of computing nodes. The cluster is managed by the control plane which provides an API to interact with the cluster. The control plane can be used to start containers, to stop containers, or to alter the configuration of running containers. To model application deployment, Kubernetes defines Kubernetes objects of different types, e. g., containers, pods, or deployments. The control plane provides instance information about all Kubernetes objects currently deployed in the cluster. To logically group Kubernetes objects into different environments, Kubernetes provides the concept of *namespaces*. Namespaces provide (i) a scope for object names, (ii) definition of access rights for users and (iii) ability to limit resource consumption. Thus, they provide an enclosed environment, that can be used to encapsulate different applications. Consequently, this work considers all objects inside one namespace as possible components for the target application. The name of the namespace that should be investigated has to be specified additionaly when defining the Kubernetes deployment instance as of Section 4.2.

Figure 4.3 shows how the different Kubernetes objects inside a namespace are mapped to TOSCA entities. The names of the Kubernetes objects are in black, while the mapped TOSCA entities are displayed in gray. As the namespace defines the scope of the instance retrieval, it is mapped to the service template. Each namespace may have several deployments which in turn may have several pods. However, these objects do not represent running components – strictly speaking. Thus, they are not directly mapped to any TOSCA entity. Each pod may consist of multiple containers and each container actually is running on some node in the cluster. Thus, each container is mapped to a node template. Deployments, pods and containers can all have properties, that might be relevant for the instance model. For example, the identifier of the container as well as the identifier of the pod are required to access the container from a management process. Thus, these properties are mapped to properties of the node template for each container. Any combination of these properties may be used to determine the TOSCA node type for a container. Most likely the image property will define the node type, however this is not guaranteed. Thus, no concrete Kubernetes ojbect can be directly mapped to the node type. This work uses a default fallback type *DockerContainer*, if no other node type could be found. Discovering horizontal relationships between two containers, e. g., a web application connecting to a database, is impossible, since no Kubernetes object allows to define such relationships.

Mapping Puppet to TOSCA

The Puppet primary server manages a set of agent nodes. Each agent supplies the primary server with facts. In combination with the user supplied configuration, the primary server derives a catalog for each agent that specifies the desired state for the agent. When an agent applies a new version of the catalog, it sends a report to the primary server indicating the new state of the agent. The primary server stores all this information – e. g., agents, reports, catalogs – in the Puppet DB, which provides an API to retrieve this information. For logically separating manifests and resource definitions, Puppet offers the concept of *environments*. The separation between Puppet environments is not as strict as Kubernetes namespaces, e. g., environments may share configuration and an agent may be referenced in multiple environments. However, an environment allows separating multiple

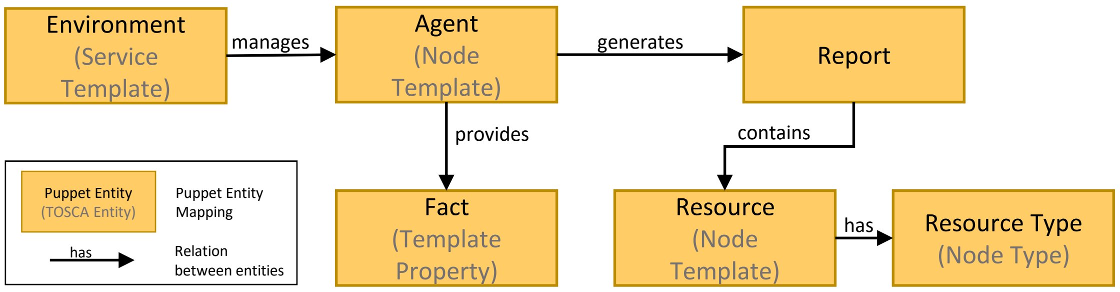


Figure 4.4: Mapping of Puppet entities to TOSCA entities

applications that are managed by a single Puppet primary server. Thus, the name of the environment that should be targeted must be specified as a property in the list of deployment technologies as of Section 4.2.

Figure 4.4 depicts how Puppet entities are mapped to TOSCA entities. Again, Puppet specific terms are displayed in black, while the respective TOSCA terms are displayed in gray. As the environment defines the scope of instance retrieval, it is mapped to a service template. The manifests defined inside an environment may reference multiple agents, each of which provides facts about itself. As every agent runs on a computing node where application resources are deployed to, it is mapped to a node instance template, which may be populated with properties that are retrieved from the facts of the agent, e. g., its IP address. Its corresponding node type must be retrieved from the supplied facts or the normative *Compute* type can be used as a fallback. For example, an agent running on a VM may supply a fact that the operating system of the node is Ubuntu. Thus, an appropriate node type would be *Ubuntu VM*. The reports for every agent, contain information about which resources were configured on the node by Puppet. Resources may be mapped to node templates. However, not all resources should actually be contained in the instance model. For example, a resource of type *file* might just alter a configuration file and thus, should not be mapped to a node template. The node type, that is assigned to a node template, is defined by the type of the corresponding resource. For example, a resource of type *package* with the name *mysql* defines an installed MySQL database server and should be mapped to a node template with the node type *MySQL DBMS*. Horizontal relations between node templates cannot be discovered, since the reports do not contain information about the dependents of a resource.

Mapping AWS CloudFormation to TOSCA

AWS CloudFormation is a SaaS offering and can be used to provision arbitrary resources in the AWS cloud. The resources that shall be provisioned are defined in a template that is uploaded to the API of AWS CloudFormation. Whenever a template is deployed by AWS CloudFormation, a stack is created that contains all provisioned resources. AWS CloudFormation provides an API which can be queried to retrieve information about deployed stacks. Creating a stack for each deployed template, AWS CloudFormation provides separation between applications by design. Thus, the name of the stack that should be used for instance retrieval must be specified as a property in the list of deployment technologies as of Section 4.2.

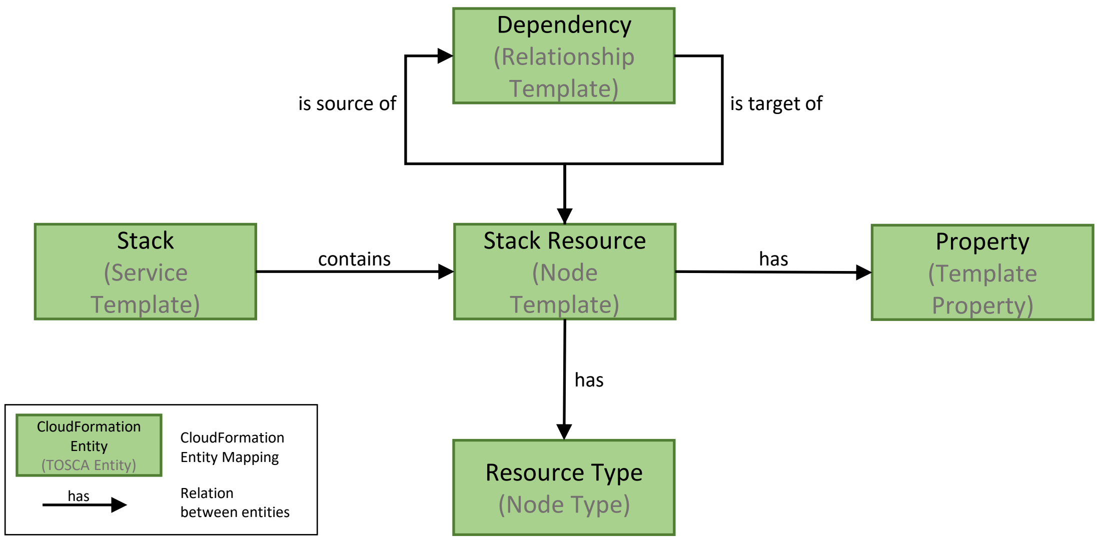


Figure 4.5: Mapping of AWS CloudFormation entities to TOSCA entities

Figure 4.5 shows how AWS CloudFormation entities are mapped to TOSCA entities. As before, AWS CloudFormation specific entities are in black, while the corresponding TOSCA entity is in gray. As a stack defines the scope of instance model retrieval, it is mapped to a service template. A stack contains multiple stack resources, each of which has a resource type and several properties. Every stack resource may be mapped to a node template, depending on its resource type. For example a resource of type *EC2::Instance* might be mapped to the TOSCA normative *Compute* type, while a resource of type *EC2::VPC* merely defines a network interface for the VM which does not need to be mapped to a node template. The properties of a stack resource can be mapped to properties of the node template, e. g., the IP address property of an *EC2::Instance* can be added as a property to the corresponding node template. In addition, stack resources may define dependencies between each other, which may be mapped to relationship template intances of the TOSCA normative type *DependsOn*.

Mapping Terraform to TOSCA

Terraform does not use an always-on service and, thus, provides no API to retrieve information from. However, the state files written by Terraform contain all information about provisioned resources and may be read and parsed for the purpose of instance model retrieval. Terraform does not provide any technical concept for separating components of different applications. However, it is encouraged to use separate Terraform workspaces for every application. As Terraform creates a single state file per workspace, parsing this file provides information on all resources that were provisioned for the target application, whilst resources of other applications should not be visible. Thus, the Terraform state file that should be parsed must be provided as a property in the list of deployment technologies as of Section 4.2.

Figure 4.6 shows how Terraform entities are mapped to TOSCA entities. As before, Terraform specific entities are in black, while the corresponding TOSCA entity is in gray. As the state file defines the scope of instance model discovery, it is mapped to a service template. A state file contains

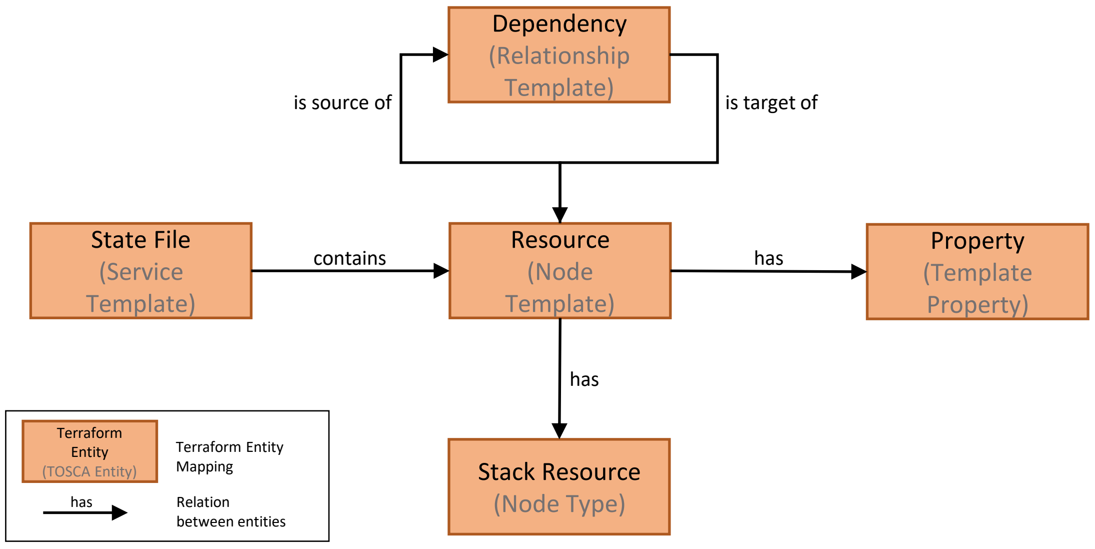


Figure 4.6: Mapping of Terraform entities to TOSCA entities

a list of provisioned resources, each of which may be mapped to a node template, depending on its resource type. The same example as in Section 4.3.1 applies: an EC2 instance should be mapped to a node template, while its network interface does not represent a dedicated application component. Every resource may have several properties which can mapped to properties of the corresponding node template. Similar to Section 4.3.1, Terraform resources may define dependencies amongst each other. These dependencies can be mapped to relationship templates of the TOSCA normative type *DependsOn*.

4.3.2 Representing Deployment Technology Information in the Instance Model

In addition to information about running component instances, the instance model must also contain information about the deployment technology used to deploy each component. This information can be used in later stages. For example, management workflows might require access to the API of a deployment technology. This section describes how this information can be included in a TOSCA instance model.

The goal is to create a TOSCA instance model, that holds all instance information about the components of an application and in addition provides all necessary information about the deployment technologies used to deploy it. The first step is to define, what information is necessary. First, the instance model should name the deployment technologies involved, in the running example these are AWS CloudFormation and Puppet. In addition, the instance model should provide information about which deployment technology was used to deploy a specific component. More importantly, it must provide information about which deployment technology still manages a specific component. For example, the Ubuntu VM runs the Puppet agent and, thus, is detected by Puppet. However, Puppet does not manage the VM as it is controlled by AWS CloudFormation. Finally, generated management workflows might need to connect to the API of a deployment technology. Thus, the instance model must contain the information, necessary for a successful connection.

There are several possibilities to represent the described information in an instance model, four of which are discussed in more detail:

1. Using nested service templates.
2. Add information about deployment technologies to every node template.
3. Add deployment technologies as additional node templates and introduce *Manages* relationship type.
4. Add information about involved deployment technologies as properties to the service template

The first option is based on the concept of nested service templates. The idea is to create a separate service template for each deployment technology, as described in Section 4.3.1. Each of these service templates would contain the information of the single deployment technology inside *tags*, as proposed by Harzenetter et al. [HBB+21]. For example, the Puppet retrieval plugin creates a service template and populates it with node templates for all discovered resources. It then, adds a tag with the key *SourceTechnology* and the value *Puppet* to indicate that the components described in the service template were deployed using Puppet. Another tag with the key *PuppetPrimaryIP* then contains the IP address of the Puppet primary server. Later in the process this information could be extracted to connect to the Puppet primary server under the specified IP address. The separate service templates, can be merged by creating a wrapping root service template. The root service template references the deployment technology specific service templates as part of its topology template. The option of nested service templates provides the highest degree of isolation between the deployment technologies and the respective plugin execution. Considering the running example, the Puppet plugin and the Terraform plugin could be executed completely independent of each other and create their isolated service template. Both service templates could easily be merged by simply wrapping them in the root service template. However, this approach has several drawbacks. First, the root template does not convey any instance information. It is a technical necessity to hold a list of other service templates and does not add any meaning to the instance model. Moreover, separate service templates do not provide the possibility to add connection between their respective node templates. For example, it is not possible to indicate, that the components managed by Puppet are *HostedOn* the VM deployed by Terraform.

The second option utilizes the properties of the node templates. Every node template could specify inside its properties, which deployment technology manages the node template and the necessary information to connect to its API. This is similar to the tag approach of the option with nested service templates. Instead of specifying the information once in the tags of the service template, the same information is replicated to the properties of each node template. The node templates inside this service template can be managed by different deployment technologies and management workflows may extract necessary information from the properties of the node template. However, these properties “pollute” the instance model, as they technically do not represent instance information about the represented application component itself. This issue gets even worse, considering that the information is replicated over each node template which clutters the instance model with redundant information.

The third option models each deployment technology as an additional node template. This requires a corresponding node type, e. g., a *Puppet* node type. The properties of this node template can hold all information about the deployment technology in a central place. To indicate which deployment technology manages an application component, the *Manages* relationship type is used. For each

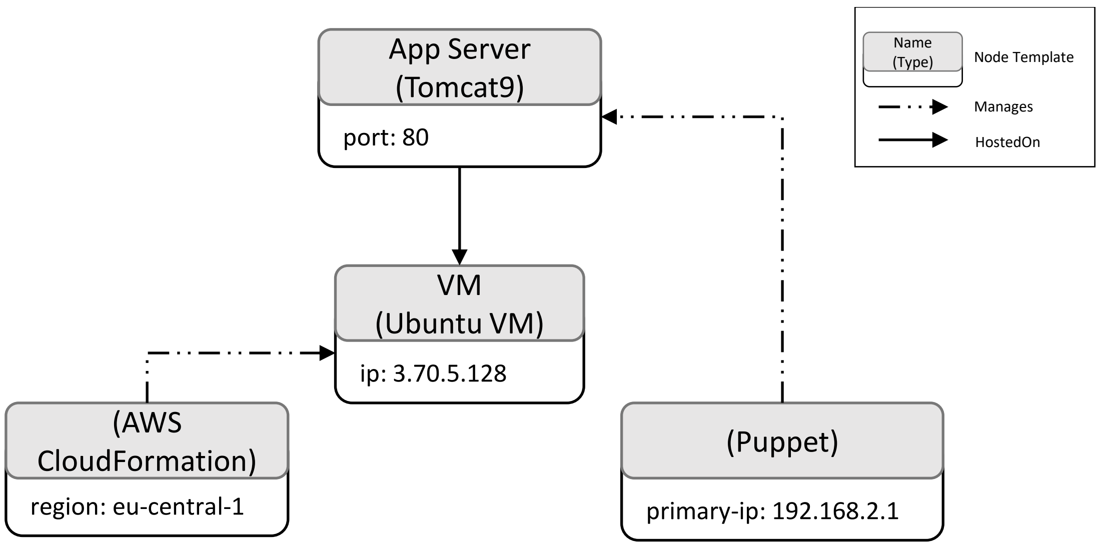
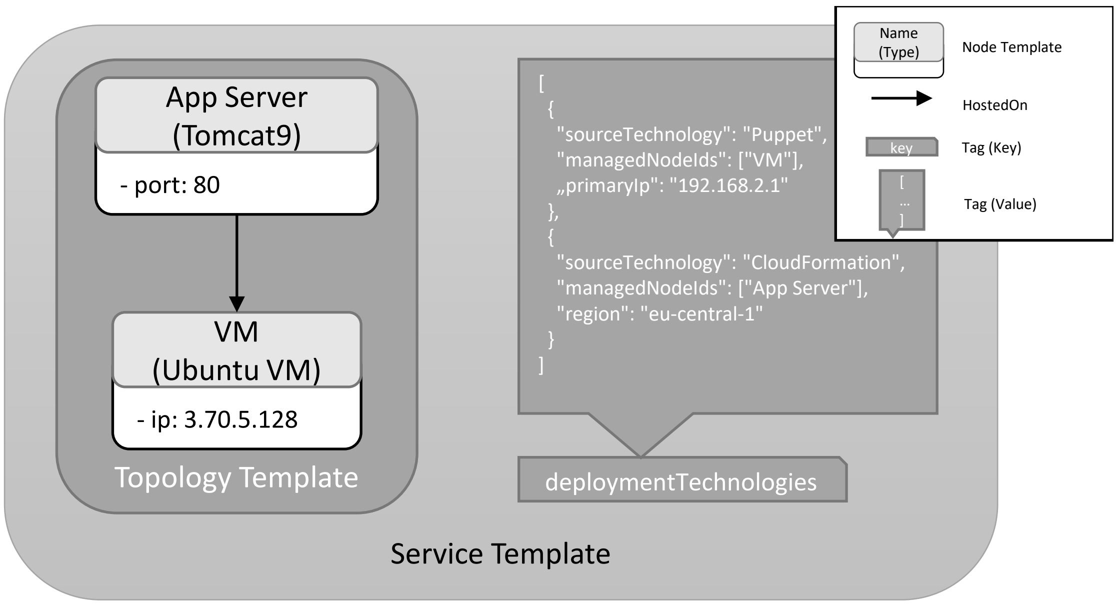


Figure 4.7: Example for representing deployment technologies as dedicated node templates

component, that is managed by a deployment technology, a relationship template of type *Manages* is added. The source of this relationship template is the node template of the deployment technology and the managed node template is the target. Figure 4.7 shows an example service template for the running example application. For the sake of brevity only the VM and the Tomcat application server are depicted. The service template contains dedicated node templates for each deployment technology used, i. e., AWS CloudFormation and Puppet. These node templates provide additional information about the deployment technology instance in their properties. For example, the IP address of the Puppet primary server is specified only once as a property of the *Puppet* node template. If a workflow targeting the VM needs access to the AWS CloudFormation API, it can search for incoming relationship templates of type *Manages* and backtrack them to the node template that represents AWS CloudFormation. It can then read the necessary properties and connect to the API. While this option avoids cluttering the model with redundant information, it still “pollutes” the instance model with information that is not part of the actual application. Although the deployment technologies are important components for deploying the application, they not actually are application components. Thus, they should not be depicted as such in the instance model.

The last option combines the usage of a single service template with the encoding of information in its tags. Figure 4.8 depicts an example service template for the running example application. Again only the VM and the Tomcat application server are depicted for the sake of brevity. The instance components are modeled in the topology template without any information about the involved deployment technologies. Thus, the topology template is a pure representation of instance information of the target application. The necessary information about the involved deployment technologies is encoded in the tags of the service template. Arbitrary methods may be used to encode information in one or multiple tags. However, this work uses a JSON structure with all necessary information and serializes it into a single tag with the key *deploymentTechnologies*. The JSON structure is essentially a list of deployment technology descriptors. Each descriptor has the mandatory property *sourceTechnology* specifying the type of deployment technology, e. g., Puppet.



```
[
  {
    "sourceTechnology": "Puppet",
    "managedNodeIds": ["VM"],
    „primaryIp": "192.168.2.1"
  },
  {
    "sourceTechnology": "CloudFormation",
    "managedNodeIds": ["App Server"],
    "region": "eu-central-1"
  }
]
```

Figure 4.8: Example for representing deployment technologies as deployment technology descriptors

The second mandatory property is *managedNodeIds*, which is a list of node template ids. Each application component, that is represented by a node template in that list, is assumed to be managed by the deployment technology represented by the descriptor. Moreover, the descriptor may contain arbitrary additional properties, e. g., the IP address of the Puppet primary server. A workflow, that needs information about the deployment technology for a specific node template, must decode the descriptors from the tags and iterate all node id lists to find the id of the target node template. If the id is found in a descriptor, the workflow can extract the required information from the remaining properties of the descriptor. The decoding of deployment technology information increases the complexity of management workflows and imposes a performance penalty. Nonetheless, this option provides many benefits. For once, there is no redundant information as all information regarding the deployment technologies is stored in a central location. Moreover, the JSON structure allows the usage of arbitrary property values instead of simple string based key-value pairs. At last, the topology template itself is kept clear of “polluting” information that does not describe the application itself. Thus, this work uses the last option.

4.3.3 Merging Instance Information from multiple Deployment Technologies

The previous sections described how to discover involved deployment technologies, how to map deployment technology specific information to a TOSCA model and how to include information about the deployment technologies in the model. This section explains how the instance information that is retrieved from multiple deployment technologies can be merged into a single TOSCA instance model.

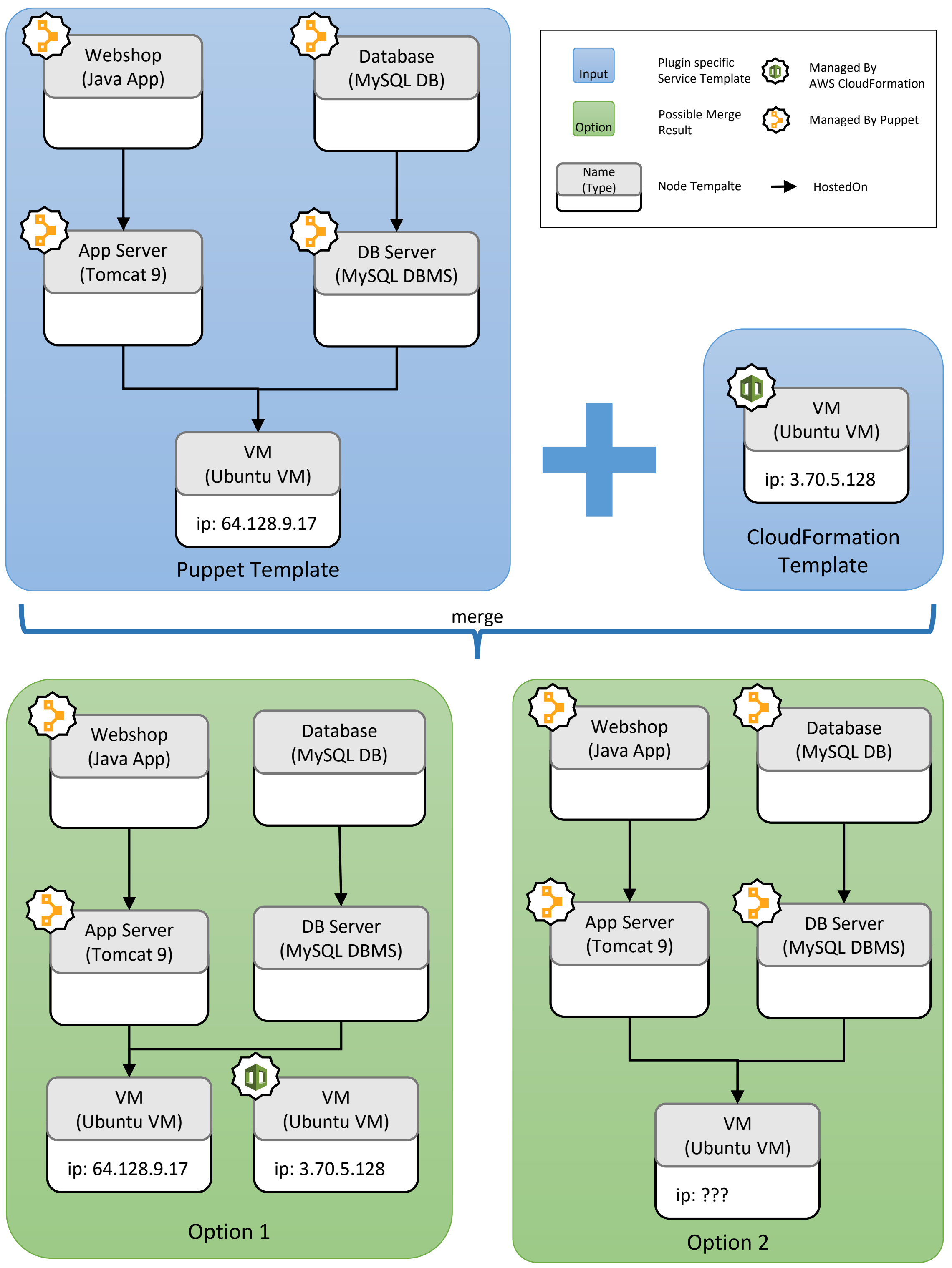


Figure 4.9: Example for merging service templates, that shows different options for the merge result

The Instance Model Retriever has a dedicated plugin for each supported deployment technology. Taking the list of deployment technologies, as of Section 4.2, one possibility is to start an independent plugin run for each entry of the list. This results in independent service templates for each plugin, all of which need to be merged into a single service template. However, the merging of these service templates is complex and cannot be performed in a deployment-technology-agnostic way. Again consider the running example application. Executing the AWS CloudFormation plugin and the Puppet plugin independently yields the two service templates depicted at the top of Figure 4.9. At the bottom the figure shows two possibilities for merging the two templates. Merging the lists of deployment technology descriptors is a simple union, thus only the topology templates are depicted. The Puppet plugin discovers all components managed by the primary server, as indicated by the icon in the top left corner of each node template. In addition, the Puppet plugin discovers the VM, as the agent is installed on this machine. However, the VM is not marked as managed by Puppet, since Puppet has not been used to deploy it. The AWS CloudFormation plugin discovers only the VM, since it is the only resource deployed by AWS CloudFormation. Both technologies provide information about the VM, e. g., its IP address. However, the IP address discovered might differ. For example, EC2 instances have two IP addresses: a public address and a private address. While the private address is registered with the OS of the VM, the public address is unknown to the OS. Thus, the Puppet plugin can retrieve the private address, while the public address can only be retrieved by AWS CloudFormation. The simplest merging approach is to create the union of the set of node templates and the set of relationship templates. The resulting merged service model for this option is shown as *Option 1* at the bottom left side of the figure. Yet, this leads to a wrong instance model with two VMs, while the application consists only of one. Consequently, the merge operation must detect, that the two VM node templates actually represent the same application component as depicted in *Option 2*. Merging the two node templates requires merging their properties, but since both specify the property *ip* with different values, they cannot be trivially merged. As the *ip* property is defined to contain the IP address on which the VM can be reached, the public IP address is the best option to choose. However, only the AWS CloudFormation plugin can determine, which of the addresses actually is the public one, since it is the only one that knows both.

As shown, merging independent service templates again requires the usage of deployment technology specific knowledge. Thus, this work lets all plugins operate on the same service template. To achieve this, the Instance Model Retriever creates an empty service template and runs the plugins for the specified deployment technology instances consecutively. Every plugin takes the current service template and adds the information it can retrieve from its deployment technology. Note, that the same plugin might run multiple times, if multiple instances of the same deployment technology have been specified. This approach does not eliminate the need to join the information. However, every plugin can use its context and deployment technology specific knowledge to prevent duplicate entries from the start. Before adding a new node template to the service template, the plugin must first check the existing node templates if they represent the same component. This can be done by comparing the assigned node type and properties of the node templates. For example, assume the Puppet plugin in the running example is executed first. As the initial service model is empty, all discovered components are directly added to the instance model, including the VM, as depicted in Figure 4.9. The subsequently running AWS CloudFormation plugin would also discover a VM, but needs to check if a corresponding node template already exists. In the case of a VM, all node templates that represent some type of software component can be ignored, while the IP address poses as a unique identifier to find identify a matching node template. In the example, the AWS CloudFormation plugin knows the public and private IP address of the VM. Thus, it can

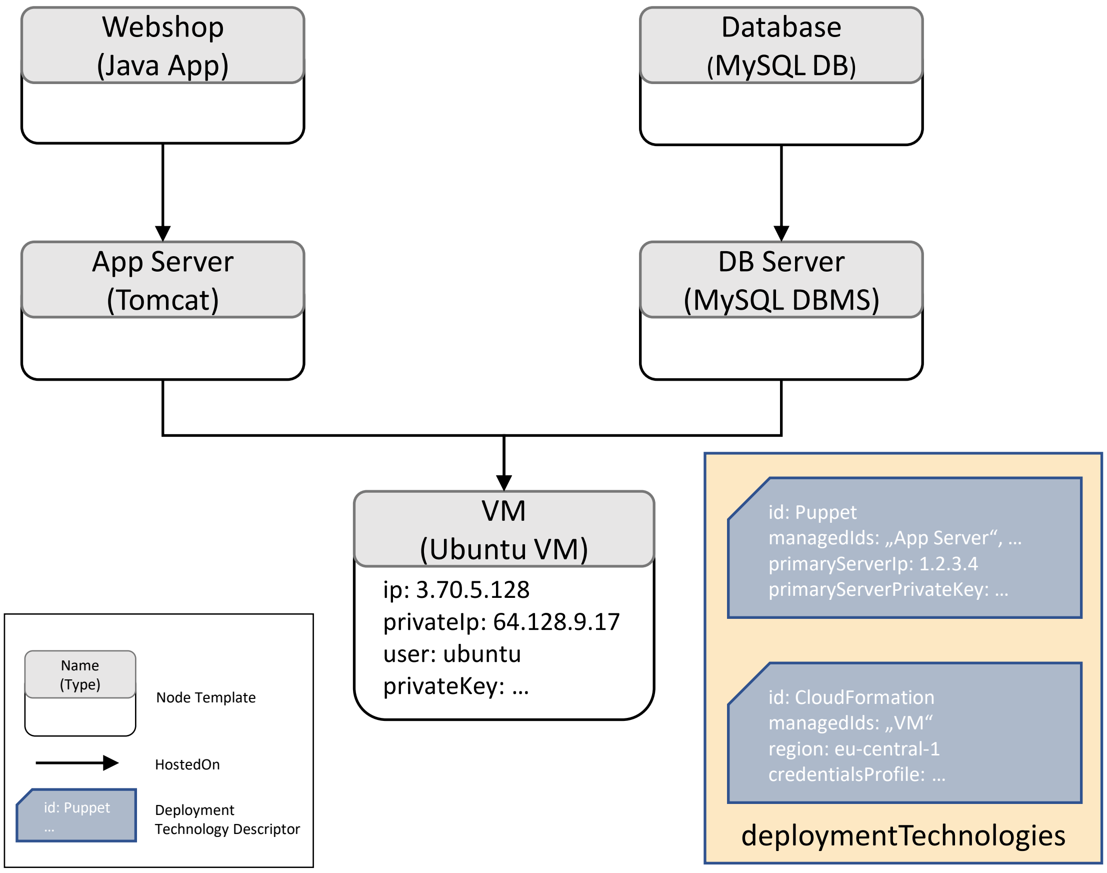


Figure 4.10: Instance model retrieved by the Instance Model Retriever for the running example application

search for a node template, that represents a VM and check its *ip* property. As the property value, provided by the Puppet plugin, will match the private address, a suitable candidate is found. The AWS CloudFormation plugin does not need to create a new node template, but can work on the existing one. In case, the plugin execution order is reversed, the AWS CloudFormation plugin also adds the private ip address to a different property, called *privateIp*. Using this property, the Puppet plugin is able to identify the node template representing the VM the Puppet agent runs on.

While the approach so far equals *Option 2*, the AWS CloudFormation plugin knows which of the two possible IP addresses is more suitable, as it is aware of the fact, that the VM is only accessible on the public one. Thus, it would correctly choose the public address as the correct value for the *ip* property. After running all plugins, the Instance Model Retriever can simply output the resulting service template without applying any additional logic. The advantage of this approach is, that the necessary merging of service template contents can be directly done by the respective plugins, using context information about the currently investigated deployment technology.

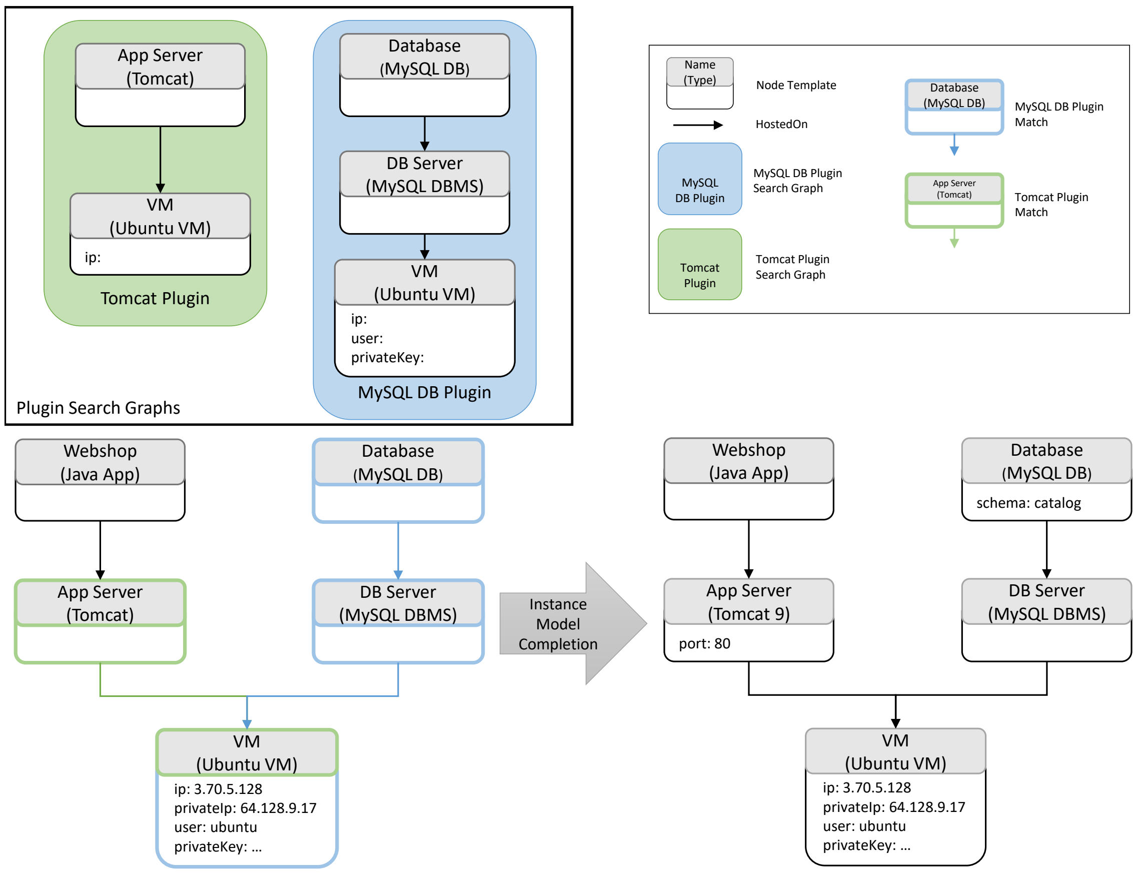


Figure 4.11: An example refinement performed by the Instance Model Completer

4.3.4 Retrieved Instance Model for the Running Example

Figure 4.10 depicts the service template that is retrieved by the Instance Model Retriever for the running example application. All components have been discovered and the deployment technology descriptors correctly define which component is managed by which deployment technology. In addition, all information necessary to connect to the Puppet primary server as well as to the AWS CloudFormation API are contained in the respective deployment technology descriptors. Moreover, the public and private IP addresses of the VM as well as the necessary credentials for establishing a SSH connection to the VM have been discovered. However, no properties could be discovered for the components deployed by Puppet. In addition, the exact version of the Tomcat application server could not be determined, thus the generic *Tomcat* node type is used. Also, the horizontal *ConnectsTo* connection between the Java application and the MySQL database was not discovered.

4.4 Instance Model Completion

As described by Harzenetter et al. [HBB+21], the information, retrieved from the deployment technologies, varies in detail and may not produce a complete model of an application. For example, the information provided by the APIs may not be sufficient to derive the exact version of a installed software component. Thus, this work reuses the plugin-based *Instance Model Completer* of Harzenetter et al. [HBB+21] and extends it. The Instance Model Completer takes the retrieved instance model and tries to retrieve additional information. To achieve this, every plugin specifies a TOSCA topology sub-graph as a search pattern. If the sub-graph matches any part of the instance model, the plugin is executed to refine the information about the respective component or to discover additional components. When executed, the plugin may use arbitrary methods to collect additional information, for example issuing HTTP requests or opening SSH sessions.

Figure 4.11 shows an example refinement of the Instance Model Completer for the running example application. The Instance Model Completer has two plugins, the Tomcat plugin and the MySQL DB plugin. The search graphs for both plugins are depicted in the top left corner of the figure. The search graph for the Tomcat plugin, depicted in green, defines a node template with the assigned node type *Tomcat* without any properties specified. This node template must be hosted on a node template with the assigned node type *Ubuntu VM* which has the property *ip* filled. Comparing the search graph with the retrieved instance model, depicted in the bottom left corner, the match highlighted in green is found. Thus, the Tomcat plugin is applied to the instance model. The plugin issues HTTP requests to a list of predefined ports, e. g., port 80, in order to determine the port the Tomcat application server is listening on. The IP address to use for these requests is extracted from the *ip* property of the hosting VM. Having discovered the correct port, the plugin adds the *port* property to the node template that represents the Tomcat application server. In addition, the plugin extracts the concrete version from the received HTTP responses and changes the assigned node type of the node template from the generic *Tomcat* node type to the correct *Tomcat 9* node type. The resulting instance model is displayed in the bottom right corner of the figure. The search graph for the MySQL DB plugin, depicted in blue, defines a node template with the assigned node type *MySQL DB* which is hosted on a node template with the assigned node type *MySQL DBMS*. The *MySQL Server* in turn must be hosted on a node template with the assigned node type *Ubuntu VM*. In addition, the *Ubuntu VM* must have the properties *ip*, *user*, and *privateKey* defined. Comparing the search graph with the retrieved instance model, the match highlighted in blue is found. The MySQL plugin is applied to the instance model and opens a SSH connection to the VM using the IP address and credentials defined in the properties of the node template of the VM. Using th SSH connection the plugin retrieves the schema name for the MySQL database and adds it to the *schema* property to the node template representing the MySQL database.

In addition, the Instance Model Completer marks the components of the instance model with the id of the plugin that was able to refine them. This information can be used to trace where the instance information came from. This will help the user to spot and debug errors in the instance model as well as increase the comprehensibility of the instance model. Analogously to the deployment technology descriptors, this work proposes to use a tag with the key *discoveryPlugins* to store a JSON list of discovery plugin descriptors. Each descriptor contains the *id* of the plugin and a list of *discoveredNodeIds*. The id of every node template, that was altered by the respective discovery plugin, is contained in this node id list. The plugins of the Instance Model Retriever, also create and manipulate node templates. Thus, each of these plugins can also create a discovery plugin descriptor

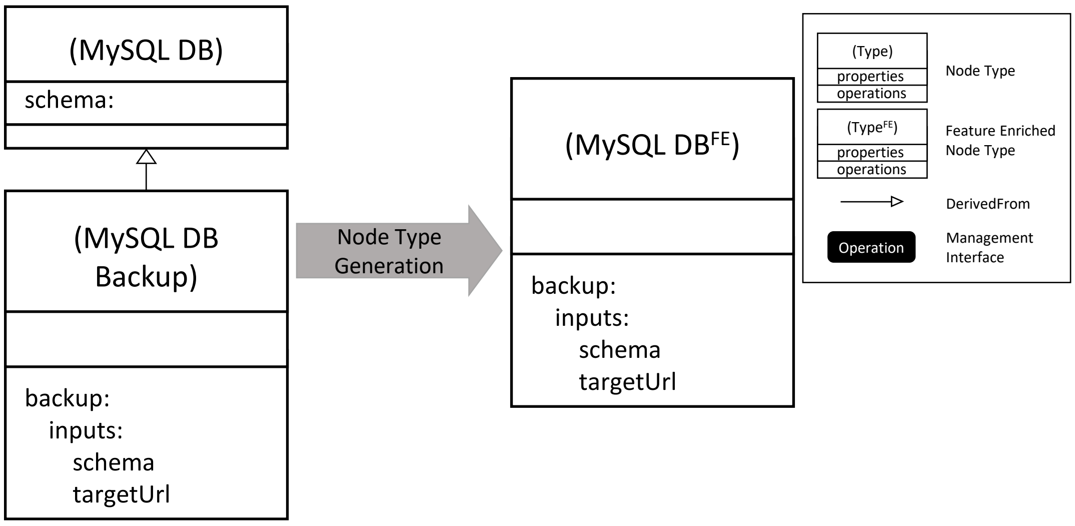


Figure 4.12: Example feature management representation for backing up a MySQL Database

alongside its deployment technology descriptor. Every node template can only be managed by a single deployment technology and thus, its id should only be contained in a single deployment technology descriptor. However, multiple discovery plugins may alter a node template and there is no upper limit for discovery plugin descriptors that contain a specific node template id.

4.5 Feature Enrichment and Workflow Generation

The concept of feature enrichment and automated workflow generation has been introduced by Harzenetter et al. [HBL+19]. While the original approach was limited to enriching deployment models, it was extended to allow the enrichment of instance models by Harzenetter et al. [HBB+21]. This work reuses this approach and extends it to support instance models of applications that have been deployed by multiple deployment technologies.

The feature enrichment of instance models is based on the concept of management operations in the TOSCA standard. Every node type, may define an arbitrary number of management operations. Each operation defines an arbitrary number of input and output parameters and is implemented by an implementation artifact. To enable the enrichment of instance models, special feature node types are defined [HBL+19]: Each feature node type defines a single management operation, e. g., the backup of a database. To identify the node types the management feature can be applied to, the feature node type must inherit from its target node type. For example, consider the *MySQL DB Backup* feature node type depicted on the left side of Figure 4.12. The feature node type defines a operation with the name *backup* that takes to input parameters *schema* and *targetUrl*. These input parameters define the target schema to backup and the target url to which the backup file is uploaded. To indicate that the backup feature, represented by the feature node type, can be applied to MySQL databases, the feature node type inherits from its target node type *MySQL DB*.

The feature enrichment of the completed instance model is performed by the *Instance Model Enricher*. For each node template in the instance model, the Instance Model Enricher loads all applicable feature node types. To actually enrich the node template with the management feature, the Instance Model Enricher generates a new node type. For this, the Instance Model Enricher merges the properties and operations of the target node type with the feature node type. For example, the *MySQL DB*^{FE}, depicted in Figure 4.12, is generated by adding all properties of the target node type *MySQL DB*, i. e., *schema*, and all operations of the feature node type, i. e., the *backup* operation. Note, that there can be multiple feature node types applicable to a node template, as management features can target the same node type. For example, in addition to the backup feature, there may also be a test feature defined for the *MySQL DB* node type. Thus, the Instance Model Enricher may loop the node type generation by merging the previously generated node type with the next feature node type.

In addition, each feature node type may specify deployment technologies that it supports. The component, represented by the enriched node template, may be managed by a deployment technology. Thus, the management operation may need to connect to the deployment technology to perform state-changing operations on the component, e. g., a software update. The Instance Model Enricher parses the deployment technology descriptors from the instance model and determines for each node template the deployment technology that manages the represented component. Based on the managing deployment technology, the Instance Model Enricher filters the list of applicable feature node types.

The enriched instance model is used to automatically derive executable management workflows. The *Management Workflow Generator* generates an executable workflow for each management operation defined in the enriched instance model [HBB+21]. For example, the instance model may contain multiple node templates with the assigned node type *MySQL DB*^{FE}. The backup operation for all these databases will be executed in a single workflow. To provide values to the input parameters of each management operation, the information contained in the instance model can be used. Values can either be extracted from the properties of the node templates or from the deployment technology descriptors contained in the tags of the instance model.

For the instance model of the running example, we assume the Instance Model Enricher finds two applicable feature node types: First the *MySQL DB* component can be enriched with the backup operation defined in Figure 4.12. Moreover, the *Ubuntu VM* component can be enriched with an update operation that installs software updates onto the VM. Provided with the enriched example instance model, the Management Workflow Generator generates two management workflows: one backup workflow and one update workflow.

5 Implementation

To prove the feasibility of the presented approach, a prototypical implementation based on the OpenTOSCA ecosystem Breitenbücher et al. [BEK+16] and the Instance Model Retrieval Framework Mathony [Mat20] is provided. The prototype includes an implementation of the Instance Model Retriever as well as extensions to the Instance Model Completer and the Instance Model Enricher. The Instance Model Retriever is implemented inside the TOSCin[1] framework and consists of a command line application for the retrieval process and plugins for the following deployment technologies: Kubernetes, Puppet, AWS CloudFormation and Terraform. The Instance Model Completer is part of Winery, cf Section 2.3, and is extended with the creation of discovery plugin descriptors as described in Section 4.4. The Instance Model Enricher, also part of Winery, is extended to extract information about the used deployment technologies from the deployment technology descriptors. All the mentioned components are implemented in Java.

The Instance Model Retriever as of Section 4.2 takes a list of deployment technology instances, that should be queried for information. The list is supplied as part of a configuration file in YAML format. Listing 5.1 shows an excerpt of an example configuration file. The *model-name* property defines the name of the service template that is output by the Instance Model Retriever. The *technology-instances* property defines the list of deployment technology instances, grouped by type. The example specifies two Puppet instances. The *puppet-primary* instance is reachable at the IP address defined by *ip* and is configured to retrieve instance information for the *backend* environment. In

Listing 5.1 Excerpt of an example configuration file for the Instance Model Retriever

```
model-name: puppet-terraform-retrieved
technology-instances:
    puppet:
        puppet-primary:
            environment: backend
            ip: 1.2.3.4
            ...
        another-primary:
            environment: frontend
            ...
    cloud-formation:
        cfn:
            stack-name: app
            region: eu-central-1
            ...
```

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

Listing 5.2 Lifecycle interface for deployment technology specific retrieval plugins

```
public interface InstancePluginLifecycle {
    void setInitialInstanceModel(ServiceTemplate serviceTemplate);

    void prepare();

    void transformToTOSCA();

    void cleanup();

    ServiceTemplate getRetrievedInstanceModel();
}
```

addition, the second Puppet instance, *another-primary*, is configured to retrieve instance information for the *frontend* environment. Moreover, the example specifies one AWS CloudFormation instance with the AWS region specified by *region*. It is configured to examine the stack with the name *app*.

The deployment-technology-specific plugins are Java classes. For every deployment technology in the configured list, an instance of the respective plugin class is created. Thus, the same plugin can be instantiated multiple times, targeting different instances of the respective deployment technology. Every plugin implements the lifecycle interface depicted in Listing 5.2. The very first execution step is to provide the plugin with the current state of the retrieved instance model. For the first executed plugin, the specified service template will be empty, while all subsequent plugins receive all information discovered by their predecessors. Having an instance model to operate on, the plugin might need to execute some preparing operations, e. g., establishing a communication channel with the target API. After successful preparation, the actual transformation can be performed. In this step the plugin retrieves information from the deployment technology and adds it to the provided service template. Hereby, it uses the mapping rules defined in Section 4.3.1. During the transformation process, the plugin also joins the information retrieved from its deployment technology with the instance model already provided by the previous plugins. After the transformation, each plugin may perform cleanup operations. In this step, the plugin releases all resources allocated during preparation or transformation, e. g., closing established communication channels. As a last step, the Instance Model Retriever requests the retrieved instance model from the plugin. Note, that the service template returned by the plugin must not be the same object as the initially provided service template. The plugin may perform arbitrary operations on the initial service template or create a completely new instance, as far as all necessary information from the previous plugin runs is preserved. In the following, the technical details plugins for every investigated deployment technology are breifly described.

Kubernetes Plugin The Kubernetes plugin utilizes the official Kubernetes Java Client[2] for accessing the Kubernetes API. The information, necessary to connect to the Kubernetes cluster, is provided to the plugin in form of a kubeconfig file [The21c]. The plugin expects a path to that file as input. As described in Section 4.3.1, the target namespace must also be provided as input

[2]https://github.com/kubernetes-client/java

to the plugin. During the preparation phase, the plugin loads the kubeconfig file and initializes the api client. For the transformation, the plugin uses the api client to retrieve all pods of the given namespace and iterates the defined containers. It then maps the containers to node templates according to the mapping specified in Section 4.3.1. As the Kubernetes api client does not need to be explicitly terminated, the cleanup phase is a no-op for the plugin.

Puppet Plugin The Puppet plugin utilizes the Puppet DB API to retrieve reports, that have been sent by Puppet agents. That API is often only accessible on the localhost for security reasons. Thus, the plugin opens a SSH connection to the master. The Puppet DB API is queried using *curl* commands that are executed on the Puppet primary server. To open the SSH connection the plugin requires credentials, consisting of a username and the path to a private key file as input. Moreover, the plugin requires the target environment as input, as described in Section 4.3.1. During the preparation phase, the plugin opens the SSH connection to the primary server. That connection is used in the transformation phase to retrieve the list of managed nodes and the reports for each node. Using the reports and the facts about the nodes, the plugin updates the service template according to the mapping specified in Section 4.3.1. The cleanup phase is used to close the SSH connection.

AWS CloudFormation Plugin The AWS CloudFormation plugin utilizes the AWS SDK for Java[3] to access different AWS APIs. AWS strictly separates all resources by geographic regions, thus the target region is the first input required, by the plugin. In addition, the plugin requires the credentials to authenticate against the AWS API. This is achieved by using authentication profiles. Thus, the plugin also requires the name of the authentication profile to use. Moreover, the plugin requires the name of the target stack as input, as described in Section 4.3.1. During the preparation phase, the plugin initializes the AWS SDK and authenticates against the API. At the beginning of the transformation, the plugin queries the AWS CloudFormation API to retrieve information about the target stack and about the contained stack resources. In a second step the plugin queries resource specific APIs for each resource, depending on the resource type. For example, if the stack contains an EC2 instance as a resource, the plugin queries the EC2 API to retrieve detailed information about the instance. The initially provided service template is updated with all the retrieved information according to the mapping specified in Section 4.3.1. As the AWS SDK for Java does not leak any resources, the cleanup phase is a no-op for the plugin.

Terraform Plugin The Terraform plugin parses a Terraform state file and retrieves all information exclusively from that file. The plugin assumes, that the state file contains all instance information for the target application and thus, requires the path to the state file as its only input. As the plugin only reads the state file from disk, the Instance Model Retriever must either run on the same machine as Terraform or the state file must be manually copied to the machine that runs the retriever. During the preparation phase, the plugin loads the specified state file and parses the JSON content into a Java object structure. In the transformation phase, the plugin interprets the state file contents and tries to map the specified resources to node templates as defined by the mapping in Section 4.3.1. As the plugin holds no communication channels and does not leak any other resources, the cleanup phase is a no-op for the plugin.

[3]`https://aws.amazon.com/de/sdk-for-java`

Listing 5.3 Abstract description of the implementation of the Instance Model Completer

```
public void completeInstanceModel(ServiceTemplate serviceTemplate){
    CompleterPlugin plugin=findApplicablePlugin(serviceTemplate);
    while(plugin != null){
        PluginDescriptor descriptor=extractDiscoverPluginDescriptor(serviceTemplate,plugin);

        List<String> refinedNodeIds=plugin.apply(serviceTemplate);
        descriptor.addDiscoveredNodeIds(refinedNodeIds);

        updateDiscoveryPluginDescriptors(serviceTemplate,descriptor);
        saveServiceTemplate(serviceTemplate);
        CompleterPlugin plugin=findApplicablePlugin(serviceTemplate);
    }
}
```

After the plugins have completed, the Instance Model Retriever stores the populated service template in the Winery repository for further processing. The Instance Model Completer is implemented as a part of Winery. Thus, either the Winery API or the Winery UI can be used to start instance model completion. The Instance Model Completer loads the service template from the repository and applies its completer plugins. Listing 5.3 shows the abstract workflow of the completer. The first step is to find an applicable plugin for the specified service template. This is done by taking the search pattern defined by each plugin and searching for a match in the topology template graph. If an applicable plugin was found, the corresponding discovery plugin descriptor must be extracted from the service template, i. e., the JSON structure must be parsed from the *discoveryPlugins* tag and a descriptor with the respective plugin id must be searched. If no descriptor was found for the selected plugin, a new one is created. When applied to a service template, each plugin returns the list of ids of nodes that it has refined. These ids must be added to the list of in the respective descriptor. At last, the updated descriptor is added to the service template. Since the completion process may be applied iteratively, as described in Section 4.4, the process is looped.

After successful completion of the instance model, feature management enrichment can be performed. For this, the Instance Model Enricher of Harzenetter et al. [HBB+21] is reused and extended to support a service template derived from multiple deployment technologies. The enrichment logic is left unchanged, only the process of determining the deployment technology that manages a given node template is adjusted. Given a node template, the Instance Model Enricher must iterate over all deployment technology descriptors and check if any of them declares the id of the given node template as managed.

Having completed the feature enrichment, the enriched instance model can be loaded into the OpenTOSCA Container. The Management Workflow Generator also reuses the implementation of Harzenetter et al. [HBB+21] and extends it to handle deployment descriptors. The generated management workflows are defined using BPEL. Each execution of a management operation is added as an invocation to the generated workflow. To provide the necessary input data to the invocation, every property of a node template and every property defined by one of the deployment descriptors is added as a variable to the workflow. These property variables and descriptor variables are assigned the value that is specified in the service template. The values of these variables are then used to invoke the actual management operation.

6 Validation

The presented approach and its prototypical implementation are validated in a case study which is described in Section 6.1. The results of the case study and their implications are discussed in Section 6.2.

6.1 Case Study

The approach, presented in this work, is validated in a case study, based on the *Sock Shop* microservices demo[1]. A coarse-grained view on the Sock Shop architecture is presented in Figure 6.1. The shop consists of six backend services, i. e., user, order, catalogue, cart, shipping and payment. Each backend service consists of a software component, that provides the API of the service, and an optional database, that holds the data for the service. Different backend services use different technologies for their API and database. For example, the cart service uses a Java app to implement its API and stores its data in a MongoDB [Mon21], while the catalogue service uses a Go [Go 21] implemented API and a MySQL database. The central frontend service connects to all backend services and presents a unified UI for the user, which is implemented as a NodeJS [Ope21]

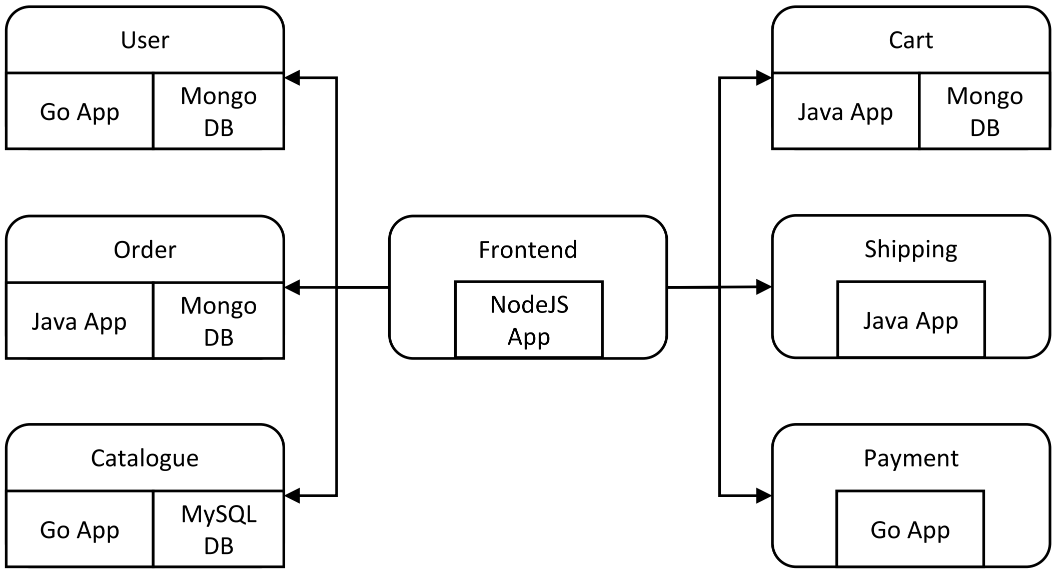


Figure 6.1: Structure of the example application *Sock Shop*

[1]https://github.com/microservices-demo/microservices-demo

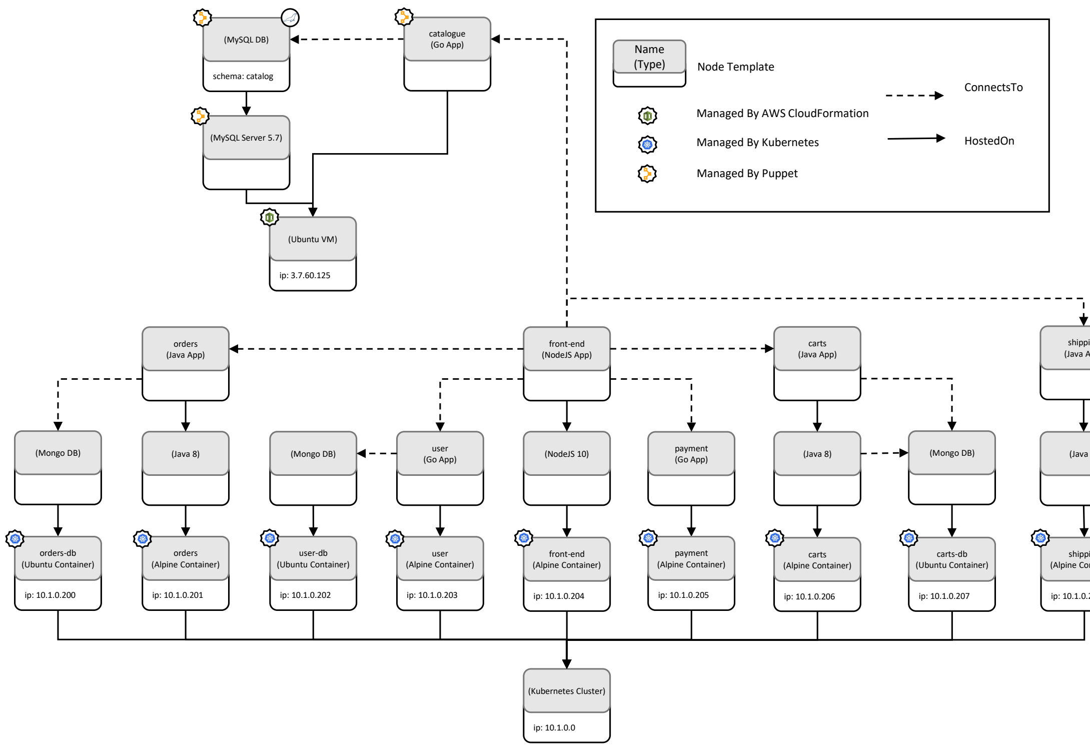


Figure 6.2: Validation scenario

application. The Sock Shop application uses some additional middleware components, e. g., for messaging purposes. However, for the sake of simplicity, these components are not considered further in this case study.

Figure 6.2 depicts the validation scenario. Apart from the catalogue service, all services of the Sock Shop are deployed onto a Kubernetes cluster. To achieve that, each API component and database is deployed in its own container. The containers use either Ubuntu or Alpine Linux as OS. To isolate the Sock Shop application from other components of the cluster the namespace *sock-shop* is used. The catalogue service, including its API and database, is deployed onto a VM running Ubuntu as the OS. The VM is provisioned in the AWS cloud using AWS CloudFormation as part of the stack *catalogue*. The MySQL database as well as the Go application that make up the catalogue service are installed on the VM using Puppet. To isolate the components of the catalogue service from other components managed by the Puppet primary node, the environment *catalogue* is used.

The first step of the case study is to run the Instance Model Retriever with the three deployment technology instances, i. e., Kubernetes, Puppet and AWS CloudFormation, as input. A shortened version of the respective configuration file is depicted in Listing 6.1. The name of the resulting service template – *sock-shop* – is defined by the *model-name* property. The deployment technology instances are defined by the *technology-instance* property. The instance of the Kubernetes retrieval plugin is assigned the id *kube-sock-shop* and is configured to retrieve instance information from the *sock-shop* namespace. The instance of the Puppet retrieval plugin is assigned the id *puppet-catalogue* and is configured to retrieve instance information for the target environment *catalogue*. The instance of the AWS CloudFormation plugin is assigned the id *cloud-formation-catalogue* and it is configured

Listing 6.1 Configuration file for the Instance Model Retriever in the case study

```
model-name: sock-shop
technology-instances:
    kubernetes:
        kube-sock-shop:
            target-namespace: sock-shop
            ...
    puppet:
        puppet-catalogue:
            environment: catalogue
            ...
    cloud-formation:
        cloud-formation-catalogue:
            stack-name: catalogue
            ...
```

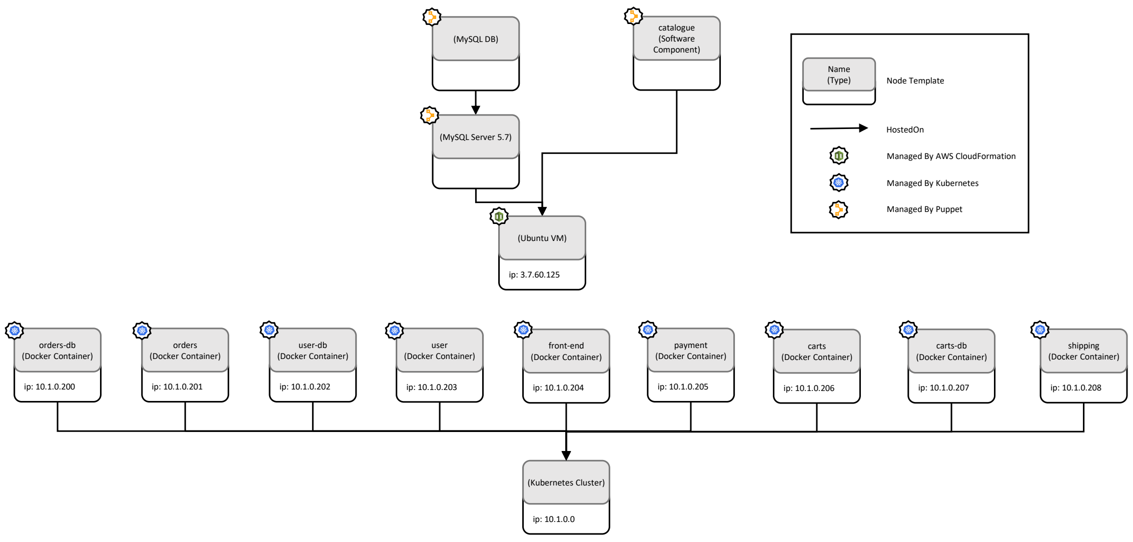


Figure 6.3: Resulting instance model after executing the Instance Model Retriever

to retrieve instance information for the target stack *catalogue*. In addition, for all plugins the necessary credentials are configured, as indicated by the ellipsis. However, they are not depicted for the sake of brevity.

Figure 6.3 shows the resulting instance model, after the Instance Model Retriever has completed. For simplicity the figure does not show the JSON deployment descriptors. Instead, the icon in the upper left corner of each node template indicates which deployment technology manages the respective component. If no icon is depicted, no deployment technology manages the respective component. The Instance Model Retriever discovered all containers running on the Kubernetes cluster and marked them as managed by Kubernetes. For each container, the properties *ContainerID*, *ImageID*, *PodName* and *ip* have been retrieved. The prototypical implementation of the Kubernetes plugin currently cannot determine the concrete node type for a container. Thus, all node templates, representing a container, are assigned the generic *Docker Container* node type. For the sake of

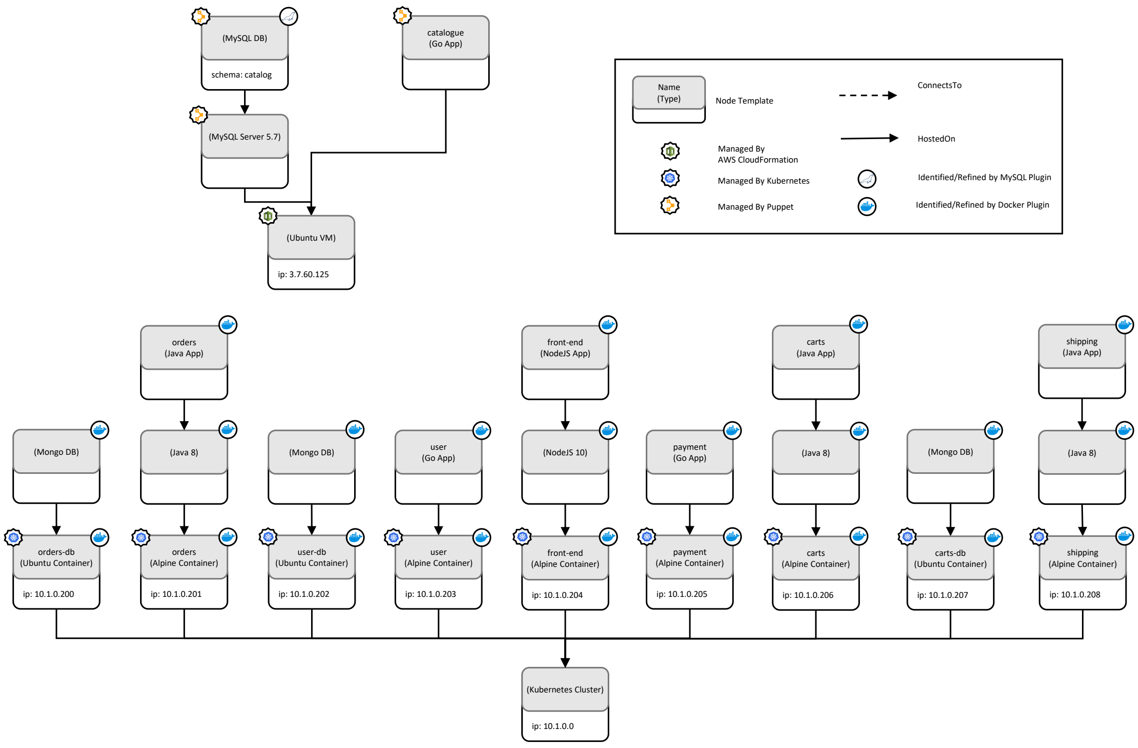


Figure 6.4: Resulting instance model after executing the Instance Model Completer

brevity, only the *ip* property is depicted. The Instance Model Retriever also discovered the VM and marked it as managed by AWS CloudFormation. The AWS CloudFormation plugin discovered the *ip* address of the VM as well as the *PrivateKey* used to connect to the VM. In addition, the Puppet plugin discovered the *UserName* which can be used to connect to the VM. Again, for the sake of brevity, only the *ip* property is depicted. The Instance Model Retriever also discovered the components installed on the VM by Puppet and marked them as managed by Puppet. This includes the installed MySQL server with the created MySQL database as well as the Go application that provides the API of the catalogue service. However, the Puppet plugin is not able to recognize the API service is a Go application. Thus, it falls back to the generic *Software Component* node type. Moreover, the Instance Model Retriever was not able to discover the *ConnectsTo* relation between the API and the MySQL database, as the connection information is compiled into the binary of the API. The resulting service template also includes the deployment technology descriptors for Kubernetes, Puppet and AWS CloudFormation. The deployment technology descriptors contain all properties necessary to establish a connection with the API of the respective deployment technology

In the next step, the Instance Model Completer tries to refine the retrieved instance model with additional information. The completed instance model is depicted in Figure 6.4. Node templates that were refined by the Instance Model Completer are marked with an icon in the upper right corner. The icon represents the plugin that was used to refine the node template. Two completer plugins have been executed on the model, i. e., the Docker plugin and the MySQL plugin. The Docker plugin can be applied to any node template that has the assigned node type *Docker Container*. The plugin uses the value of the *ImageID* property and maps it to a predefined sub graph. It replaces the original node template with the sub graph. For example, consider the *carts-db* container. The node

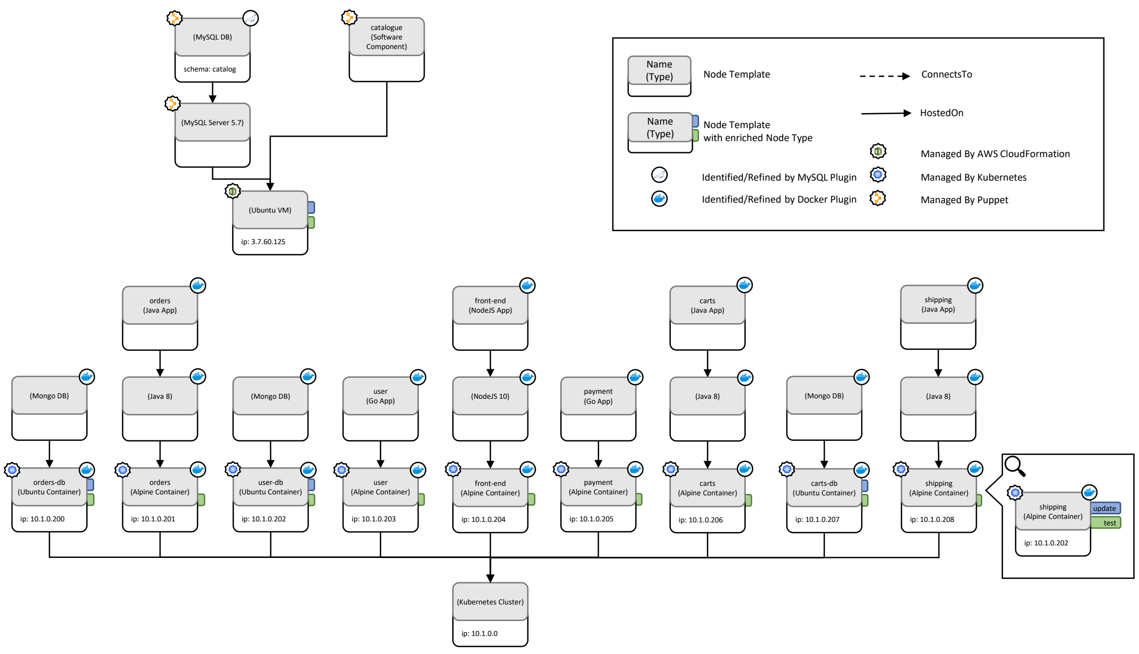


Figure 6.5: Resulting instance model after executing the Instance Model Enricher

template has the assigned type *Docker Container* and the *ImageID* property of value *mongo*. The Docker plugin replaces the original *Docker Container* node type with the more specific *Ubuntu Container* node type and adds a new node template of type *Mongo DB* to the model that is *HostedOn* the container. The MySQL plugin can be applied to a node template with the assigned type *MySQL DB*. When applied, the plugin tries to determine the name of the *schema* that the node template represents. The database of the catalogue service is the only one node template in the retrieved instance model that can be refined by the MySQL plugin. The plugin correctly retrieves the schema name *catalog* for the catalogue database.

Given the completed instance model, the Instance Model Enricher checks for available management features and applies them to the instance model. For this case study, two management features are available: *update* and *test*. The *update* feature updates installed software packages, while the *test* feature checks for the availability of a component. In this case study, there are five feature node types that provide the above management features: *Ubuntu Container Update*, *Ubuntu VM update*, *Ubuntu Container Test*, *Alpine Container Test* and *Ubuntu VM Test*. Using these feature node types, the Instance Model Enricher performs the enrichment depicted in Figure 6.5. Every node template that has an assigned node type that was enriched with the *test* feature is marked with a green management operation. Analogously, every node template that has an assigned node type that was enriched with the *update* feature is marked with a blue management operation. To prove that the management operations can access the deployment technology, the *update* and *test* operations for containers are implemented to connect to the Kubernetes API, in order to run commands inside the container. It can be seen, that all *Ubuntu Containers* are enriched with both management features, as is the *Ubuntu VM*. However, the *Alpine Containers* are only enriched with the *test* feature, since there is no *update* feature node type targets the original *Alpine Container* node type. Using the enriched instance model, the Management Workflow Generator generates two management workflows: an *update* workflow and a *test* workflow. When executed, the workflows invoke the respective management operation on each component.

6.2 Discussion

The case study, described in Section 6.1, validated that the approach and the prototypical implementation allow the retrieval of instance models for applications that have been deployed by multiple deployment technologies. Moreover, the resulting instance model can be used to generate management workflows in an automated manner. However, there still exist open challenges: First, the approach still requires manually configured input. The initial list of deployment technology instances has to be specified manually, while all following steps may be executed in a purely automated fashion.

Furthermore, the approach relies on the accessibility of the deployment technologies. The Instance Model Retriever gathers information from the deployment technologies, either from their APIs, e. g., Puppet, or directly from the data storage of the deployment technology, e. g., the Terraform state file. However, there might be deployment technologies that do not provide access to the necessary instance information. For example, deployment technologies that are provided as a SaaS offering may be not offer an API. Missing access to the Terraform state file is another example. If the Instance Model Retriever lacks access to the state file, e. g., if it is located on a different machine, no instance information can be retrieved.

In addition, the approach only discovers components of an application that have been deployed with a deployment technology. However, there exist applications that consists of components that have been deployed with deployment technologies as well as components that have been manually deployed. For example, an enterprise application can have services that are deployed into the cloud with deployment technologies, while some legacy components are still deployed manually. These manually deployed components can neither be discovered nor managed with the proposed approach.

Comparing the completed instance model from Figure 6.4 with the full validation scenario depicted in Figure 6.2 shows, that all components of the application have been successfully discovered. Moreover, all *HostedOn* relations have been correctly identified. However, none of the *ConnectsTo* could be discovered by the prototypical implementation. This is due to the fact, that none of the investigated deployment technologies models connections between components. The deployment technologies provision the infrastructure, install necessary packages and start the software components. However, every component itself is responsible for establishing connections to other components. As an example, the API and the database of the orders service are deployed as separate containers by Kubernetes in the case study. While Kubernetes takes care of provisioning and starting the containers, the orders API is responsible for locating the database and for establishing a connection.

Lastly, the proposed approach only generates management workflows for management operations that target a single component. For example, the test operation in the case study can only check the availability of a single component. The approach allows combining multiple execution of the same management operation targeting different components into a single workflow. Nonetheless, there are management operations that target multiple components. For example, a database backup might require to stop all services that access that databases. In this case, the services must be stopped before performing the backup and must be started again after completing the backup. As this backup operation involves action on multiple components in a specific order, it cannot be realized by the current approach.

7 Summary and Future Work

This work presented an approach to retrieve an instance model of an enterprise application that was deployed with multiple deployment technologies. This included the manual discovery and definition of the involved deployment technologies. Moreover, for every of the investigated deployment technologies, i. e., Kubernetes, Puppet, AWS CloudFormation and Terraform, a method to isolate components specific to the target application as well as a mapping strategy to TOSCA models has been described. In addition to the instance information, the proposed approach also describes different possibilities for including information about the involved deployment technologies and discussed their benefits and drawbacks. To actually retrieve the models, this work introduced the *Instance Model Retriever* a plugin-based component, that uses the list of deployment technologies and the specified mapping strategies to create a single TOSCA service template representing the instance state of the target application. In order to retrieve information, that is not provided by the deployment technologies, the presented approach also includes the *Instance Model Completer*, which may use arbitrary methods to refine the instance information. Based on the retrieved instance model, the *Instance Model Enricher* can add management features to the application components which can be used to generate automated management workflows. To prove the feasibility of the presented approach, this work also provided a prototypical implementation based on the OpenTOSCA ecosystem and the Instance Model Retrieval Framework. To validate the presented approach and its prototypical implementation, a case study has been performed, which applied the prototype against an example application. The results of the case study proof that the approach can be used to retrieve instance models from multiple deployment technologies and that these model can be used to generate automated management workflows.

Future Work

Nonetheless, challenges for future work have been identified. The presented approach only retrieves instance information from deployment technologies, ignoring manually deployed components. Thus, the approach can be extended to query additional sources for instance information. Moreover, the approach relies on the accessibility of the APIs of deployment technologies. Future work should investigate how deployment technologies can be included that do not offer APIs for instance information retrieval. Moreover, the approach should be refined to improve the discovery of relations between components. Lastly, the generation of management workflows can be extended to allow management operations which target multiple components.

Bibliography

[Ama21] Amazon Web Services. *AWS CloudFormation*. 2021. URL: https://aws.amazon.com/cloudformation (cit. on pp. 13, 19, 22).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. "A Systematic Review of Cloud Modeling Languages". In: *ACM Computing Surveys* 51.1 (2018), pp. 1–38. ISSN: 0360-0300. DOI: 10.1145/3150227 (cit. on p. 13).

[BBH+10] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications". In: *Service-Oriented Computing*. Ed. by P. P. Maglio, M. Weske, J. Yang, M. Fantinato. Vol. 6470. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 692–695. ISBN: 978-3-642-17357-8. DOI: 10.1007/978-3-642-45005-1_62 (cit. on p. 27).

[BBKL13a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "Automated discovery and maintenance of enterprise topology graphs". In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. 2013, pp. 126–134. DOI: 10.1109/SOCA.2013.29 (cit. on pp. 28, 29, 31, 38).

[BBKL13b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Pattern-based Runtime Management of Composite Cloud Applications". In: *Closer*. 2013, pp. 475–482 (cit. on p. 28).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "Migration of enterprise applications to the cloud". In: *it - Information Technology* 56.3 (2014), pp. 106–111. ISSN: 1611-2776. DOI: 10.1515/itit-2013-1032 (cit. on pp. 28, 32).

[BBKL14b] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. Ed. by A. Bouguettaya, Q. Z. Sheng, F. Daniel. New York, NY: Springer New York, 2014, pp. 527–549. ISBN: 978-1-4614-7535-4. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 17).

[BBKL14c] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Vinothek - A Self-Service Portal for TOSCA". In: *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*. Ed. by N. Herzberg, M. Kunze. Vol. 1140. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 69–72. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-25&engl= (cit. on p. 27).

[BCS17] A. Brogi, P. Cifariello, J. Soldani. "DrACO: Discovering available cloud offerings". In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 269–279. ISSN: 1865-2034. DOI: 10.1007/s00450-016-0332-5 (cit. on p. 31).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. "The OpenTOSCA Ecosystem - Concepts & Tools". In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*. SCITEPRESS - Science and Technology Publications, 2016, pp. 112–130. ISBN: 978-989-758-207-3. DOI: 10.5220/0007903201120130 (cit. on pp. 15, 17, 27, 55).

[BFGL20] A. Brogi, S. Forti, C. Guerrero, I. Lera. "Towards Declarative Decentralised Application Management in the Fog". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 223–230. ISBN: 978-1-7281-7735-9. DOI: 10.1109/ISSREW51248.2020.00077 (cit. on p. 32).

[BKH05] A. B. Brown, A. Keller, J. L. Hellerstein. "A model of configuration complexity and its application to a change management system". In: *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005*. 2005, pp. 631–644. DOI: 10.1109/INM.2005.1440836 (cit. on p. 13).

[Can21] Canonical. *Ubuntu*. 2021. URL: https://ubuntu.com/ (cit. on p. 18).

[Clo21] Cloud Native Computing Foundation. *Kubernetes*. 2021. URL: https://kubernetes.io (cit. on pp. 19, 20).

[Dea07] A. Dearle. "Software Deployment, Past, Present and Future". In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 269–284. DOI: 10.1109/FOSE.2007.20 (cit. on pp. 13, 17).

[Doc21] Docker, Inc. *Docker*. 2021. URL: https://www.docker.com/ (cit. on p. 20).

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

[FAB+11] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, I. Hanschke. "Automation Processes for Enterprise Architecture Management". In: *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*. IEEE, 2011, pp. 340–349. ISBN: 978-1-4577-0869-5. DOI: 10.1109/EDOCW.2011.19 (cit. on p. 31).

[Go 21] Go Lang. *Go*. 2021. URL: https://golang.org/ (cit. on p. 59).

[Has21a] HashiCorp. *Introduction to Terraform: What ist Terraform?* 2021. URL: https://www.terraform.io/intro/index.html (cit. on p. 13).

[Has21b] HashiCorp. *Terraform*. 2021. URL: https://www.terraform.io/ (cit. on pp. 13, 23).

[HAW11] H. Herry, P. Anderson, G. Wickler. "Automated planning for configuration changes". In: *LISA* (2011) (cit. on p. 18).

[HBB+21] L. Harzenetter, T. Binz, U. Breitenbücher, F. Leymann, M. Wurster. "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)*. SciTePress, 2021, pp. 99–110. DOI: 10.5220/0010477900990110 (cit. on pp. 13–15, 28, 33, 38, 44, 51–53, 58).

[HBL+19] L. Harzenetter, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder, M. Wurster. "Automated generation of management workflows for applications based on deployment models". In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. 2019, pp. 216–225. DOI: 10.1109/EDOC.2019.00034 (cit. on pp. 13–15, 32, 33, 52).

[HBLE14] H. Holm, M. Buschle, R. Lagerström, M. Ekstedt. "Automatic data collection for enterprise architecture models". In: *Software & Systems Modeling* 13.2 (2014), pp. 825–841. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0252-1 (cit. on pp. 31, 37).

[KBBL10] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery – A Modeling Tool for TOSCA-Based Cloud Applications". In: *Service-Oriented Computing*. Ed. by P. P. Maglio, M. Weske, J. Yang, M. Fantinato. Vol. 6470. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 700–704. ISBN: 978-3-642-17357-8. DOI: 10.1007/978-3-642-45005-1_64 (cit. on p. 27).

[LF09] F. Leymann, D. Fritsch. "Cloud computing: The next revolution in IT". In: *Proceedings of the 52th Photogrammetric Week* (2009), pp. 3–12 (cit. on pp. 13, 17).

[Mat20] T. Mathony. *Deployment-technology-agnostic management of running applications*. 2020. DOI: 10.18419/opus-11122 (cit. on pp. 27, 32, 33, 55).

[MDW+00] V. Machiraju, M. Dekhil, K. Wurster, P. Garg, M. Griss, J. Holland. "Towards generic application auto-discovery". In: *NOMS 2000. 2000 IEEE/IFIP Network Operations and Management Symposium 'The Networked Planet: Management Beyond 2000' (Cat. No.00CB37074)*. IEEE, 2000, pp. 75–87. ISBN: 0-7803-5928-3. DOI: 10.1109/NOMS.2000.830376 (cit. on p. 31).

[Mon21] MongoDB. *MongoDB*. 2021. URL: https://www.mongodb.com/ (cit. on p. 59).

[OAS07] OASIS. *Web Services Business Process Execution Language*. 2007. URL: https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (cit. on p. 25).

[OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications*. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 15, 17, 24).

[OAS13b] OASIS. *Topology and Orchestration Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. 2013 (cit. on p. 24).

[Obj10] Object Management Group. *Business Process Model and Notation (BPMN)*. 2010. URL: https://www.omg.org/spec/BPMN/2.0/About-BPMN/ (cit. on p. 25).

[Ope21] OpenJS Foundation. *NodeJS*. 2021. URL: https://nodejs.org/en/ (cit. on p. 59).

[Opp03] D. Oppenheimer. "The importance of understanding distributed system configuration". In: *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*. 2003 (cit. on pp. 13, 17, 38).

[Ora21] Oracle. *MySQL*. 2021. URL: https://www.mysql.com (cit. on p. 18).

[Pro21] Progress Software Corporation. *Chef*. 2021. URL: https://www.chef.io/ (cit. on p. 13).

[Pup21a] Puppet. *Developing Puppet code*. 2021. URL: https://puppet.com/docs/puppet/7/developing_code.html (cit. on pp. 13, 21).

[Pup21b] Puppet. *Introduction to Puppet*. 2021. URL: https://puppet.com/docs/puppet/7/puppet_overview.html#puppet_overview (cit. on pp. 21, 22).

[Pup21c] Puppet. *Puppet*. 2021. URL: https://puppet.com/ (cit. on p. 21).

[The19] The Open Group. *ArchiMate 3.1 Specification*. 2019. URL: https://pubs.opengroup.org/architecture/archimate3-doc/front.html (cit. on p. 31).

[The21a] The Apache Software Foundation. *Apache Tomcat*. 2021. URL: https://tomcat.apache.org/ (cit. on p. 18).

[The21b] The Kubernetes Authors. *Kubernetes Components*. 2021. URL: https://kubernetes.io/docs/concepts/overview/components/ (cit. on p. 20).

[The21c] The Kubernetes Authors. *Organizing Cluster Access Using kubeconfig Files*. 2021. URL: https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/ (cit. on p. 56).

[The21d] The Kubernetes Authors. *What is Kubernetes?* 2021. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (cit. on p. 20).

[WBB+20] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “The EDMM Modeling and Transformation System”. In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Ed. by S. Yangui, A. Bouguettaya, X. Xue, N. Faci, W. Gaaloul, Q. Yu, Z. Zhou, N. Hernandez, E. Y. Nakagawa. Vol. 12019. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 294–298. ISBN: 978-3-030-45988-8. DOI: 10.1007/978-3-030-45989-5_26 (cit. on p. 32).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The essential deployment metamodel: a systematic review of deployment automation technologies”. In: *SICS Software-Intensive Cyber-Physical Systems* (2019). ISSN: 2524-8510. DOI: 10.1007/s00450-019-00412-x (cit. on pp. 13, 17–20).

[WBK+20] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. “Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models”. In: *Advanced Information Systems Engineering*. Ed. by S. Dustdar, E. Yu, C. Salinesi, D. Rieu, V. Pant. Cham: Springer International Publishing, 2020, pp. 20–35. ISBN: 978-3-030-49435-3. DOI: 10.1007/978-3-030-49435-3_2 (cit. on p. 28).

All links were last followed on October 31, 2021.

Declaration