University of Pisa

Department of Computer Science

Master's Degree in Computer Science

M.Sc. Thesis

Automating the deployment of cloud-native applications over multiple platforms

Candidate:
Leonardo Frioli

Supervisors:
Antonio Brogi
Jacopo Soldani
Michael Wurster

Academic Year 2019/2020

Abstract

Deployment automation is crucial nowadays, and a plethora of deployment automation technologies have recently been released. The Essential Deployment MetaModel (EDMM) and the ecosystem of its accompanying tools were proposed to ease the migration from one technology to another. EDMM allows to describe multi-component application deployments in a technology-agnostic way. The EDMM Modelling tool enables editing the specification of the deployment in a graphical environment. The EDMM Transformation Framework is capable of transforming the EDMM specification of a deployment into the specific files required by a chosen declarative deployment technology to actually enact the application deployment. A Decision Support System (DSS) is also available, to support developers in selecting the technologies that can actually support a specified deployment. Unfortunately, the ecosystem of tools around EDMM was not integrated, hence requiring developers to set up different environments for different tools, and to switch from a tool to another. In addition, the DSS only allowed to understand which deployment automation technologies supported the current deployment, without any further suggestion. The thesis presents two contributions tackling the aforementioned issues. We first discuss the design and development of a standalone solution integrating the ecosystem of tools accompanying EDMM. We then present an extension of the DSS. It now allows developers to automatically adapt the deployment of a multi-component application to enable its transformation to a technology-specific model, if the deployment automation technology was not already supporting the modelled deployment. We finally assessed the integration and the enhanced DSS by a case study comparing different deployments of a third-party multi-component application performed with and without the integrated EDMM ecosystem.

Contents

List of Figures

Chapter 1

Introduction

Cloud platforms feature virtually infinite computing resources to run applications. These resources can be provisioned and released on demand to scale applications and satisfy customers needs. At the same time, manually deploying and configuring multi-service applications on cloud platforms is a complex and error-prone task.

Various deployment automation tools have been released to support developers and operators in automating the deployment and configuration of cloud-based applications. Such deployment automation tools follow the DevOps philosophy by trying to automate as much as possible the application development and delivery [16]. Developers are hence no more responsible of manually provisioning and orchestrating the resources needed to run the services forming their applications, as this is fully automated by deployment automation tools. Deployment automation tools can be divided in two categories, i.e., *imperative* and *declarative* [11]. The idea behind the first ones is to describe a detailed delivery process, specifying all the technical tasks to be executed, their implementation, and order. Declarative tools follow a different approach: Developers only provide a model describing the desired configuration and requirement of an application. Given the desired application configuration, declarative tools automatically determine and execute the operations required to reach such con-

figuration. This thesis focuses on the declarative tools because the declarative approach resulted the most accepted and used in the industry [24].

Each deployment automation technology has its own peculiarities and require a certain amount of expertise to be used. In addition, given that they share the same purposes, different deployment automation technologies offer similar features in a different way. The above makes it not easy to port the deployment of an application from a technology to another. Developers should hence choose carefully among them and they should perform the choice as late as possible in order not to have an application bounded to a specific deployment tool. The risk is to get into a form of lock-in due to the difficulty of changing the deployment tool once it has been used for the first deployment [24].

The Essential Deployment MetaModel (EDMM) was introduced [24] to tackle the issue mentioned above. EDMM allows to specify a technology- and vendor-agnostic deployment model, in order to have a uniform view of the application deployment state with the essential elements and without the overhead introduced by a specific technology. EDMM is accompanied by an ecosystem of tools, i.e., the EDMM Modelling and Transformation tools [23], and a Decision Support System (DSS) [22]. An application developer can edit the EDMM specification of her multi-component application deployment in the web UI featured by the EDMM Modelling tool. While modelling, the DSS can be exploited to check which declarative technologies can support the modelled deployment. The application developer can then export a YAML file specifying the modelled deployment in EDMM. The EDMM Transformation Framework can be fed with the exported file to automatically generate the technology-specific files required by the target technology in order to provision the infrastructure running the modelled deployment. The idea is to have a single deployment model, describing the state of a multi-component application, which can be transformed to technology-specific models just by running the EDMM Transformation Framework.

While the EDMM Modelling tool and the DSS were two standalone tools, they were already capable of interacting each other via REST. The EDMM

Transformation Framework was instead a standalone application accessible only via command-line interface. A developer was hence required to separately deploy the Modelling tool and the DSS, and to configure them to allow them to intercommunicate. She could then exploit them to graphically edit the specification of the desired application deployment and to understand which deployment automation technologies could enact the specified deployment. Afterwards, she was required to export the EDMM specification of the deployment and to switch to the EDMM Transformation CLI to obtain the technology-specific files capable of enacting the deployment. It would have been better to have an unique integrated environment allowing the modelling, the decision support and the transformation without changing tools and performing additional steps.

Moreover, the feedback given by the DSS was only allowing developers to understand which deployment automation technologies were supporting a modelled application deployment. For each technology not supporting a modelled deployment, developers were also informed on the reasons causing the technology not to support such deployment. No further support was given, hence asking developers to understand on their own how to adapt the deployment of a multi-component application in order to deploy it with a desired deployment automation technology (if it was not supporting such deployment yet). A support for automatically adapting an application deployment to enable deploying it with a given technology was hence needed [22].

The aim of this thesis is precisely to help solving the aforementioned issues, as well as to assess the support given by EDMM and the accompanying tools, if compared with manually performing the corresponding tasks. In particular, we provide the following three main contributions:

1. We present the design and development of the integrated EDMM environment, a standalone solution integrating the EDMM Modelling tool, the DSS, and the EDMM Transformation Framework. With the integrated EDMM environment, developers can now edit the EDMM specification of

a multi-component application deployment, exploit the DSS, and export the technology-specific files in the same graphical environment.

2. We present an extension of the support given by the DSS, which now suggests developers which transformations to apply to a specified application deployment, in order for such deployment to get supported also by a given deployment automation technology.

3. We assessed the usefulness of EDMM and of the integrated EDMM environment with a case study. We analysed different deployments of a third-party multi-component application in order to evaluate the effort required by an application developer adopting, or not, the integrated EDMM ecosystem.

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background on EDMM and its accompanying toolset. Chapter 3 introduces a scenario motivating the need for our contributions. Chapter 4 and Chapter 5 describe the design, implementation, and testing of the integrated EDMM environment and of the extension of the DSS, respectively. Chapter 6 presents a case study to analyse the usefulness of using the integrated EDMM ecosystem with respect to manually specifying application deployments. Finally, Chapters 7 and 8 discuss related work and draw some concluding remarks, respectively.

Chapter 2

Background

In this chapter we present the Essential Deployment MetaModel (EDMM) and its accompanying tools: Modelling tool, Transformation Framework, and Decision Support System. The discussion includes also a simple usage case.

EDMM [24] tries to answer to the necessity of having a common denominator among all the declarative deployment technologies. Tools like Ansible, Puppet and AWS CloudFormation share the same purposes and allow to accomplish similar goals: Use code to provision infrastructure resources. They however offer different features, mechanisms, and deployment specification languages. There is no systematic way to compare them, and it gets really difficult to migrate from one tool to another: This requires the ability to translate the deployment specifications from one technology to another one [23]. The aim of EDMM [24] is to define a single way to specify the application deployment and to implement a tool that maps the deployment specification into a target technology deployment model. The first goal is accomplished by the Essential Deployment MetaModel itself [24], while the second is fulfilled by the EDMM Transformation Framework. The Modelling tool, which is an extension of the Eclipse Winery project [15], allows to specify graphically the deployment while receiving feedbacks from the Decision Support System (DSS). The feedbacks list the declarative technologies accepting the modelled application deployment. A YAML file specifying such

deployment in EDMM can then be exported from the EDMM Modelling tool and fed to the EDMM Transformation Framework (Figure 2.1). The latter then enables automatically generating the technology-specific files allowing to enact the specified deployment with the selected deployment automation technology.

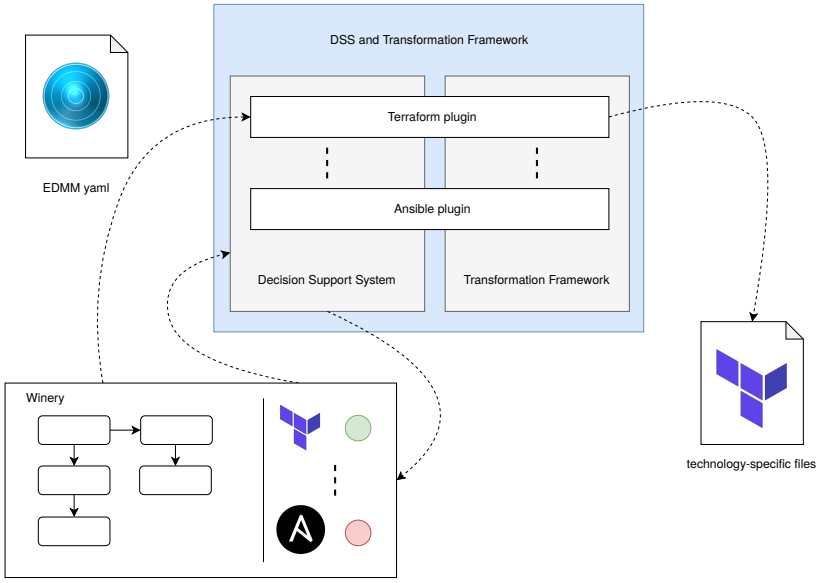

Figure 2.1: EDMM architecture [22]

2.1 The Essential Deployment MetaModel

EDMM is a meta model that allows to describe deployments of multi-component applications. The described model is given by a topology graph [4], whose nodes models the components of the application and whose edges models the dependencies occurring among such components. The modelled specification includes only the essential part of the deployment, in order to provide a uniform and common way of analysing a deployment, independently from the technology-specific representations.

A systematic review [24] of the most popular declarative deployment technologies was performed in order to analyse the common functionalities and core

aspects of these tools. The result was the Essential Deployment MetaModel defining entities (to describe an application deployment) and mappings between the entities and the abstractions used by the technologies reviewed. The metamodel follows the principles of the declarative deployment models: Describing the desired state and structure of the application including the components, configurations and relations. We hereafter recall the definitions of the entities that are exploited in the rest of this thesis:

Component Type: A *component type* is a reusable entity that specifies the semantics of a component. Figure [2.2] shows the component types supported by the tools accompanying EDMM, giving a general overview of them and of their inheritance structure.

Component: A *component* is an instance of a component type, like an object is an instance of its class.

Relation Type: A *relation type* is a reusable entity that specifies the semantics of a relation. At the time of writing, the tools accompanying EDMM support the following three relation types: *ConnectsTo*, *DependsOn*, *HostedOn*.

Relation: A *relation* is a directed physical, functional or logical dependency between exactly two components.

Operation: An *operation* is an executable procedure performed to manage a component or a relation.

Artifact: An *artifact* implements a component or operation and it is required for their execution. The artifacts related to the *Operations* are files containing the logic to install, configure, start and stop a certain component. While artifacts implementing a component may be compressed source code or binary files containing the business logic.

Deployment Model: A *deployment model* is a graph describing the *Components* and the *Relations* among them. Each node has its *Component*

Type and it can specify *Operations* and *Artifacts*. Each edge has a proper *Relation Type* and connects exactly two components. The model is used to describe the desired target state of an application as is done in the declarative approach.

The *Deployment Model* graph structure derives from the notion of application topologies [4]. By describing the components and the dependencies of the application, it is possible to specify a *Deployment Model* that represents a uniform and common specification of the desired deployment state. A deployment model can be mapped to any target deployment tool. EDMM hence provides a technology-independent way of designing deployments allowing to get rid of the *technology lock-in*: It is not needed to commit to a specific deployment technology, because it is sufficient to specify the model which is guaranteed to be converted to any of the reviewed tools. The technology can be selected as late as possible and migrated at any time, without portability issues.

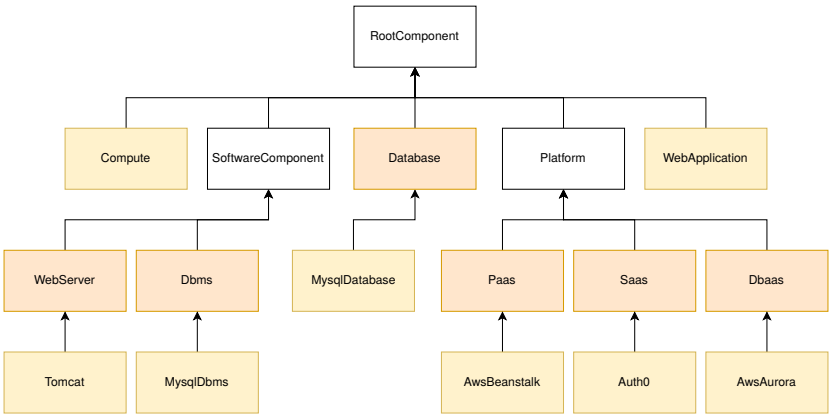

Figure 2.2: Component Types currently implemented in EDMM. There is the possibility to extend the *orange* classes with new *yellow* component types. In a deployment model, most of times, we are going to see the *yellow* types, even if it is possible to instantiate the *orange* ones too.

EDMM has a YAML specification[1] that helps to write deployment models in an easy and human readable way, leveraging the entities defined by EDMM itself. The graph structure of the deployment model is represented in YAML format by having a list of *Components*, each specifying the *Relations* starting from that component and ending to another node described in the file. The YAML file represents a machine readable element that can be processed by the EDMM Transformation Framework to get the technology-specific files needed for enacting the specified deployment with a given deployment automation technology.

2.2 The EDMM Modelling tool

Developers can graphically edit the EDMM specification of a multi-component application deployment and export the corresponding YAML file thanks to the EDMM Modelling tool. The latter is an extension of the Winery modelling tool [15], which is a web-based environment giving utilities to graphically model TOSCA-based application topologies. Even if EDMM defines its entities while Winery works with OASIS TOSCA standard [18], the Essential Deployment MetaModel entities can be mapped to TOSCA elements [24] (i.e., the TOSCA *Service Template* corresponds to the *Deployment Model*). This makes it possible to work with EDMM types inside the Winery environment.

Winery is structured around the *winery-repository*: A directory containing the files describing the templates and the types of the Winery elements. The repository is managed, accessed and populated by a Java backend. The backend is composed by many packages, but the only one useful for the discussion is *org.eclipse.winery.repository.rest*, which contains the REST endpoints called by the web UI. The frontend is divided in two Angular applications, i.e., the TOSCA Management UI and the Topologymodeler. The first web application allows to graphically manage the *winery-repository* by adding, modifying and deleting types and templates; while the Topologymodeler presents a canvas to

[1] `https://github.com/UST-EDMM/spec-yaml`

draw *Service Templates* by dragging and dropping the previously defined elements.

EDMM has its own modelling-repository[2] that has already all the entities defined as well as some artifacts and deployment topologies. If Winery knows the EDMM modelling-repository location, it will provide to an application developer the capability of instantiating the EDMM types from the graphic interface. Thanks to the Topologymodeler, we can draw an application deployment using the types defined by EDMM, while relying on the Winery internal data model. The modelled deployment has the usual graph structure. An application developer can draw it by creating the nodes and wiring them with the *Relations*. Each node, which is an EDMM *Component*, may have additional information like configuration parameters or *Artifact* files. An application developer can add this information from the Management UI and they will be associated to the nodes in the topology graph. Both the backend and the frontend have been extended to export the modelled *Service Template* into an EDMM compliant *Deployment Model* written as a YAML file. The Winery template, which already is a graph, is translated to the EDMM YAML description representing a *Deployment Model*. In this way, an application developer does not have to write the file manually, but she can just draw the application deployment and then download the file needed by the EDMM Transformation Framework.

2.3 The EDMM Transformation Framework

The EDMM Transformation Framework [23] takes as input the EDMM specification of a multi-component application deployment and it automatically generates all artefacts needed to enact such deployment with existing deployment automation technologies. It supports 13 popular deployment automation technologies, i.e., Ansible, Azure Resource Manager, AWS CloudFormation, Chef, Cloudify, Docker Compose, OpenStack Heat, Kubernetes, Puppet, Terraform, Juju, SaltStack, CFEngine. The support for each of such technologies is imple-

[2] `https://github.com/UST-EDMM/modeling-repository`

mented by a separate plugin, so that their logic and mechanisms are decoupled from the core of the EDMM Transformation Framework and can be easily extended.

Once we have the EDMM YAML file describing the *Deployment Model* in a technology-independent way, we can feed it as input to the EDMM Transformation Framework command-line interface (CLI). The *Deployment Model* is parsed into a Java graph object, where each component is represented by a node and each relation is represented by an edge. Maintaining the graph representation is useful because allows the plugins to visit the deployment model with graph algorithms or visitors. The selected technology plugin will traverse the graph and produce a directory with the technology-specific files describing the model, as well as the *artifacts* files needed for the operations and components. A plugin may implement the logic to manage a certain *component type* or it may rely on the *create*, *configure*, *start*, *stop*, *delete* operation files provided by an application developer.

Migrating from one deployment technology to another is just a matter of passing the same EDMM YAML file describing the application deployment to the EDMM Transformation Framework, while at the same time instructing it to generate the deployment artefacts for different deployment automation technologies. This hence saves application developers from having to rewrite (a portion of) the deployment specification when wishing to migrate from a deployment automation technology to another, i.e., when wishing to change the deployment automation technology actually used to enact the deployment of a multi-component application. Suppose, for instance, that we wish to deploy a Tomcat server firstly with Juju and then with Kubernetes. Juju allows deploying application components on virtual machines, while Kubernetes allows to deploy them on containers. Using EDMM, we are only required to define once a deployment model describing a *Tomcat* component *hosted on* a *Compute* node. This model will be then automatically mapped to a virtual machine running a Tomcat sever, when transforming the model to deploy it on Juju. Instead, when

targeting Kubernetes, the model will be automatically transformed in a Docker container downloading the image of Tomcat.

2.4 The EDMM Decision Support System

The EDMM Decision Support System [22] analyses the topology while the developer is modelling it and returns a list of the deployment automation technology supporting that model. While application developers edit the EDMM specification of a multi-component application deployment in Winery, they receive a feedback on the deployment automation technologies that can actually support such deployment. If the *Service Template* has one or more components that cannot be translated to something supported by a given deployment automation technology by the corresponding plugin, it is useless to try to transform the EDMM YAML specification to such technology. A concrete example in this direction is given by EDMM models including *PaaS* or *SaaS* components (Figure 2.2), which are not supported by all the deployment technologies. For instance, Juju does not support *PaaS* or *SaaS* components, so it is not possible to transform an EDMM YAML file containing an *AwsAurora* node into a valid Juju model. The Decision Support System will inform the application developer about this while graphically editing the EDMM specification of the deployment.

The DSS [22] shares part of its logic with the EDMM Transformation Framework tool because it exploits some of the mechanisms of the latter[3] (Figure 2.1). Each plugin had a function, *checkModel*, that took the deployment model represented as a Java graph in input, visited it and outputted the unsupported components. The function simply traversed the graph and marked each component as supported or not, just by looking at its type. Each plugin has his set of supported components. For instance, while Terraform supports all the components; AWS CloudFormation does not support *Auth0* and Ansible does not support any *Platform* component. The DSS exposed a REST API allowing

[3]The given description of the Decision Support System refers to the former version of the system, before all the updates applied with this thesis.

Winery to access its functionalities. Winery was then continuously interacting via REST with the DSS to provide developers with live feedbacks while modelling *Service Templates*, i.e., reporting the components not supported by each technology and highlighting them in the modelling canvas.

Chapter 3

Motivating scenario

Consider the PetClinic application[1]. The core components involved are two: A Java web application, based on Spring, and a MySQL database. The Java component is a simple REST service that implements operations to manage pets and their owners in the MySQL database. The service provides a web interface based on HTML templates to graphically add and search the pet owners. The Java component needs to be installed on a *Web Server* and requires a connection to a *MySQL database* hosted on a *Database Management System* or leveraging a *Database as a Service* Cloud utility.

Figure 3.1 shows the modelling of a possible deployment for such application in Winery. For doing it, we first need to have all building blocks (i.e., all component and relation types) available in Winery itself. Notably, most of the component types are already available in the EDMM modelling-repository[2]. The only exception is the *PetClinic* component, representing the Spring based service, that should be specified in Winery by "subclassing" a *WebApplication* component type. The relation types are instead all already defined in the modelling-repository. We also need to add some *Artifacts* to the component types, using again the Winery UI. The PetClinic needs a *.war* file containing the source code

[1] `https://github.com/UST-EDMM/spring-petclinic`

[2] `https://github.com/UST-EDMM/modeling-repository`

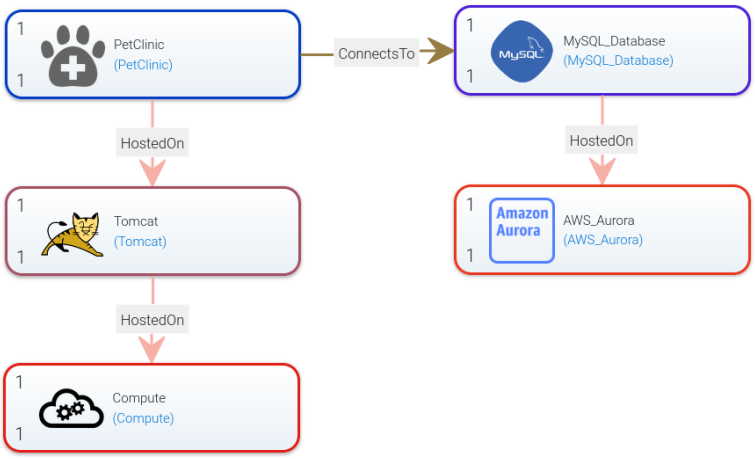

Figure 3.1: PetClinic Deployment

of the app; the database has a *.sql* schema file and Tomcat requires some files concerning the *create* and *start* operations. After this initial preparation, the deployment can be modelled in Winery by dragging and dropping the components from the Winery palette and by wiring them as shown in Figure 3.1. It is possible to see the components of the infrastructure like *Tomcat*, *Compute*, *AwsAurora* and the components specific to the application, i.e. the *PetClinic* and the *MysqlDatabase*. The arrows linking them are the relations. The *HostedOn* means that the source component needs to be installed on top of the target component. The *ConnectsTo* relation means that a connection should be setted up between the two components.

Once we have structured our deployment, we may wish to check which deployment automation technologies support such deployment. We can hence use the DSS. It tells us that *Terraform* and *Aws CloudFormation* are the only tools able to transform this deployment because the *AwsAurora* component type is not supported by all the other technologies.

We can now export the EDMM specification of our deployment, obtaining the YAML file in Listing 3.1. In the file, each component is listed, associated

```yaml
---
components:
  Tomcat:
    operations:
      start: modelling-repo/artifacttemplates/tomcat/start.sh
      create: modelling-repo/artifacttemplates/tomcat/create.sh
    type: tomcat
    relations:
    - hosted_on: Compute
    properties:
    port: '8080'
  PetClinic:
    operations:
      start: modelling-repo/artifacttemplates/petclinic/start.sh
      configure: modelling-repo/artifacttemplates/petclinic/configure.sh
    type: web_application
    relations:
    - connects_to: MySQL_Database
    - hosted_on: Tomcat
    artifacts:
    - war: modelling-repo/artifacttemplates/petclinic/petclinic.war
```

Listing 3.1: Snippet of a portion of the EDMM YAML file specifying the application deployment in our motivating scenario. The full YAML file can be found at `https://github.com/wikilele/master-thesis/blob/master/motivating-scenario/petclinic-xaas.yaml`.

with a *type*, and defines its relations. The target of the relation is another component, which is used to satisfy some dependency of the source component. Each component also specifies its operations and artifacts by providing the paths to the files implementing them. Each component may specify properties specific to its type (e.g. the *port* for the *Tomcat* component).

Suppose now that we saved the obtained EDMM YAML specification in the file `petclinic-xaas.yaml`, and that we wish to generate the technology-specific files for enacting the application deployment in Terraform. We can generate such files by invoking the command-line interface of the EDMM Transformation Framework as follows:.

```
edmm transform terraform petclinic-xaas.yaml
```

The command produces a directory called *terraform* with a *.tf* file and the copy of the artifact and operation files specified in the YAML[3]. The *.tf* file can then be given as input to the Terraform CLI.

In this scenario, we were hence able to proceed from the editing of the EDMM specification of the desired deployment for PetClinic to its actual deployment in Terraform. This however required us to first set up and configure the editing environment, and then to exploit it to edit the EDMM deployment of PetClinic. We were then asked to export the YAML file specifying such deployment, and to separately run the EDMM Transformation Framework to obtain the files for enacting the deployment in Terraform. Why not enabling to run an integrated environment, not needing to get configured, and allowing to directly obtain the Terraform-specific files from its graphical UI?

In addition, we selected Terraform as target deployment automation technology since the DSS informed us that it was supporting our deployment. What if we wished to deploy PetClinic with Kubernetes or Juju, for instance? We would have been required to manually edit the specification and adapt the deployment so that it get supported by Kubernetes and Juju as well. Why not supporting application developers also in reasoning on how to adapt their deployment specifications to get them deployed also by unsupported deployment automation technologies?

In the following chapters we will explain how we designed and obtained an integrated environment to tackle the necessities explained above. We will talk also about the new version of the DSS, which allows to semi-automatically reshape the deployment model in order to use it with the chosen technology.

[3] `https://github.com/wikilele/master-thesis/tree/master/motivating-scenario`

Chapter 4

The integrated EDMM environment

This chapter describes how the EDMM Modelling in Winery, the EDMM Transformation Framework, and the Decision Support System were originally related. The design and implementation of the integrated EDMM environment is then presented.

4.1 Starting configuration

Even if the DSS and the EDMM Transformation Framework were part of the same Java package, they did not interact with Winery in the same way. It is also important to specify that they shared some logic and code (in the *edmm-core* module[1]), but they were constituting separate, non-integrated tools. They used the same classes to parse EDMM YAML files and to obtain the Java representation of the *Deployment Models* specified in such files. Beside that, they were acting in different environments, as they were implemented differently and separately.

[1] `https://github.com/UST-EDMM/edmm/edmm-core`

The EDMM Transformation Framework was designed as a standalone Java application exposing its services via a command-line interface, so that it could be used in CI/CD pipelines. It was not integrated with Winery because they were executed in two completely different environments, i.e., while Winery is web-based, the EDMM Transformation Framework was intended to run through a command-line interface. The only common element between Winery and the EDMM Transformation Framework was that Winery provided a way to edit and export the EDMM-compliant specifications in YAML files, which could then be fed to the EDMM Transformation Framework. The Winery backend imported some of the classes specified in the *edmm-core* (the module also used by the EDMM Transformation Framework), required to initialize a data structure that could be converted to a YAML string. The lack of integration was requiring application developers to export the EDMM YAML files, from the Winery graphical environment, and then to pass them to the EDMM Transformation Framework (through its command-line interface).

The DSS was instead capable of interacting with the Winery frontend. Winery was indeed continuously interacting with the DSS to provide application developers with live feedbacks: Each time the specified deployment for an application was updated in Winery, the DSS was queried checking its deployability. This was done through an ad-hoc REST API offered by the DSS, which allowed to access the logic checking the support for a specified application deployment. The DSS had hence to be up and running while application developers were interacting with Winery, so that its REST endpoint could have been called and a list of results, one for each technology, could have been retrieved. Developers were provided with a pre-configured Docker Compose file (publicly available on the EDMM GitHub repository[2]), which allowed to run both Winery and the DSS and to interconnect them, so that they could communicate and that the live feedback from the DSS could be get.

[2] `https://github.com/UST-EDMM/getting-started/blob/master/docker-compose.yml`

4.2 Design of the integrated EDMM environment

For integrating Winery, the DSS, and the EDMM Transformation Framework together, two possible solutions were analysed, i.e, a library-based integration and a REST-based integration. Intuitively speaking, the library-based integration was intended to include the DSS and the EDMM Transformation Framework as a Java dependencies for Winery. The REST-based integration was instead intended to engineer the DSS and the EDMM Transformation Framework in a web service exposing their functionalities through a REST API. Both solutions are described hereafter, together with their pros (+) and cons (−). At the end of the section, we discuss the chosen approach and explain the rationale behind it.

4.2.1 Library-based integration

The DSS and the EDMM Transformation-Framework could both have been integrated as Java dependencies of Winery. This would have required to import their source code in the Winery backend as a library. Both tools have their core functionalities implemented in the *edmm-core* module[3] that can be packaged and added to the Winery dependencies. Once imported, some glue-code should be added to Winery in order to access the functionalities of the DSS and of the EDMM Transformation Framework. The backend is in charge of importing the library, instantiating its classes, calling its functions, and then exposing the result via the REST API used by the frontend. This integration is transparent from the Winery frontend point of view, because it already knows how to call the backend REST endpoints. The frontend needs to adjust the GUI and the way it calls the endpoints offered by the backend. This approach comes with the following pros and cons:

+ The Winery backend is a Java application that uses Maven as building tool. The DSS and the EDMM Transformation Framework core logics

[3] `https://github.com/UST-EDMM/edmm/tree/master/edmm-core`

are implemented under the same Java module, *edmm-core*. This enables packaging the *edmm-core* code and importing it from the Winery backend as a Maven dependency.

+ The Winery *Service Template* is entirely managed by the backend that has already the functionalities to parse it. Notice that a *Service Template* can be mapped to a *Deployment Model*. The Winery backend is able to pass the *Deployment Model* to the DSS and to the EDMM Transformation Framework without having to convert it to a representation more suitable for a REST communication. For instance, there is no need to convert the Winery *Service Template*, which has a graph structure, into a YAML string and then reconvert the string into the internal EDMM graph representation of the *Deployment Model*. The backend could directly process the *Service Template* graph to get the *Deployment Model* graph.

+ Without the library integration, the DSS should perform two API calls: One to the Winery backend and another to the Decision Support System or the EDMM Transformation Framework endpoint. The first call retrieves the YAML string describing the *Deployment Model*; the second call passes the YAML to the DSS or to the EDMM Transformation Framework. Using the library solution, just one call needs to be performed. It is sufficient to call the Winery backend that has already all the information to build the object used by the DSS and the EDMM Transformation Framework. This object, which is a Java graph describing the *Deployment Model*, can be directly passed to the *edmm-core* library functions to get the requested results.

+ Independently from which integration we used, the Angular interface should be updated in order to provide "buttons" to interact with the EDMM Transformation Framework. The UI elements to call the DSS functionalities are already present. With the library approach, the DSS and the EDMM Transformation Framework functionalities will be always accessible, so we do not need to hide or show these buttons depending on the

availability of an external service as it was done for the DSS. The UI should hence result cleaner and more accessible from the viewpoint of application developers.

+ The logic shared by the DSS and the EDMM Transformation Framework, implemented in the *edmm-core* module, is already imported in Winery. To generate the EDMM YAML file, Winery uses some classes and constructs implemented in the *edmm-core* module. We are already able to ship that module as a packaged library, so implementing the library-based approach should only require to add the glue-code in Winery. The 'export YAML' functionality cannot be removed and Winery needs some of the *edmm-core* classes also in the REST based approach, because it has to know how to convert a *Service Template* into a valid EDMM YAML. Since the *edmm-core* should be imported anyway, we might as well expose the functionalities of the DSS and of the EDMM Transformation Framework to implement the integration.

+ The integration testing should be easier to perform. Winery is a Maven based project, so the testing can be automatically performed leveraging the Maven utilities. It is simpler to test an imported library with respect to two services, i.e., Winery and the REST service resulting from the second approach. The integration tests, between the Winery code and the imported library, can be performed by directly writing them in Winery and running them via Maven. The test of two separated services requires to execute them in parallel to obtain separate log files, which have then to be analysed separately, hence complicating the debugging [19].

− The development should respect the Winery rules for pushing and writing code. Winery is an open source project, but some precise steps are required to work on a topic [10]. For instance, before merging a pull request we should have just one single commit in our branch. Having such guidelines improves the code quality, while at the same time it can result in slowing

down the coding or in complicating the application of changes to the project.

- The libraries shared by Winery, the DSS and the EDMM Transformation Framework should be aligned. We can not add to Winery two dependencies referring to the same library, but with different version. Maven will download both of them and it is not a good practice of having mismatches in the versioning. If we encounter shared libraries between Winery and the *edmm-core* we must fix the issue. This might be not easy because one version of a library might differ from the other. The source code of Winery, of the Decision Support System, and of the EDMM Transformation Framework might need to be changed and the changes might bring problems, especially if the new common version adopted is quite different from the previous ones and the code assumptions are not valid any more.

- The *edmm-core* module exposes the EDMM Transformation Framework logic that is used by the *edmm-cli* module[4]. We must preserve the possibility of using the EDMM Transformation Framework via command-line interface. If some fixes are required in the *edmm-core*, we should pay attention not to make assumptions valid only for the Winery integration, but breaking the code when using the command-line interface.

- The EDMM modelling repository should be shipped with Winery, because without that it is impossible for an application developer to instantiate the correct EDMM types. A *Service Template* that is not modelled with the EDMM types cannot be processed by the DSS or by the EDMM Transformation Framework. The library approach would hence require application developers to download the EDMM modelling repository.

- The entry barrier for the library-based integration approach may be higher if compared to other approaches, as we would be required to deeply understand Winery before applying updates to it. The size of Winery itself,

[4] `https://github.com/UST-EDMM/edmm/tree/master/edmm-cli`

and the fact that some parts of Winery are not fully documented, may hence hamper our understanding and slow down the process of enacting the integration.

4.2.2 REST-based integration

With the REST-based integration, the DSS and the EDMM Transformation Framework can be both included in a RESTful service, e.g., in Docker containers, running and exposing a REST API. The communication should be performed passing the YAML file to the service and getting back the results, i.e. the feedbacks from the DSS or the technology-specific files. The YAML file gets more importance because it is the contract on which the two services must agree. To have the full capabilities of EDMM, Winery and the REST service (packaging the DSS and the EDMM Transformation Framework) should be executed in parallel, so we have to think about the consequences of using Winery without EDMM. This approach can be considered a natural extension of the former implementation of the DSS.

+ The implementation should be easier, because we do not need to modify the backend code. Winery already allows to export a valid EDMM YAML file. It should be sufficient to modify the Winery frontend in order to let it call the backend for the YAML string generation and then the EDMM service (i.e the service exposing the Decision Support System and Transformation Framework functionalities).

+ There should be less code to write, because the DSS is already implemented in this way. In the EDMM *getting started* GitHub repository[5] there is already a docker compose file starting Winery and the DSS service implemented by the *edmm-web* module[6]. The endpoint for the EDMM Transformation Framework is already coded in the *edmm-web* package. We just need to adjust the interfaces and test the API calls.

[5] `https://github.com/UST-EDMM/getting-started`
[6] `https://github.com/UST-EDMM/edmm/tree/master/edmm-web`

+ The development should be easier with respect to the library approach. The Winery code should only be slightly modified, hence making us less coupled to Winery's code constraints. We have more freedom working on the EDMM repository.

– This approach gives to the integrated environment a microservices-like architecture. Adopting the microservices philosophy might be good, but only under certain conditions. One of the principles of microservices is to divide the application in services if and only if it is strictly required [19]. In the case of Winery and EDMM there is no real need to adopt a solution like that, because both can be considered part of the same domain since their business capability is to edit and manage the EDMM deployment model.

– The DSS needs a continuous interaction with the Winery interface. Keeping it in a service that may or may not be up-and-running seems a bit counterintuitive. The service should be queried multiple times during the modelling phase to provide a live feedback. This may result in unnecessarily wasting network resources and in latency problems, which can both be avoided with the other approach. With the DSS the most natural solution seems hence to be the one library-based, because of the nature of the system itself.

– The classes needed by Winery to implement the 'export YAML' functionality must be imported anyway. We can not remove the Winery code implementing the conversion from the *Service Template* to the EDMM YAML string for two reasons. An application developer should still be able to export the YAML file and pass it to the EDMM Transformation Framework. The endpoint generating the YAML file is fundamental, because the frontend needs to query it before passing the YAML to the EDMM service (running the Decision Support System and Transformation Framework). The *edmm-core* code should be maintained both to be

used as a library and to be called by the *edmm-web* code implementing the REST service.

- The Winery Angular interface needs in both cases (library-based or REST-based integration) to be modified in order to provide the "buttons" that trigger the calls to the DSS or the EDMM Transformation Framework. In this case, the buttons are useful only if the EDMM service is up and running. Even if it is not a big problem, we should pay attention to hide completely the EDMM related UI if the service has not been started by an application developer.

- Winery and the *edmm-core* are both written in Java. Since the language is the same there is not the need of communicating between application written in different languages. The REST service approach is usually more useful if it is not possible to import one application code into the other because of language incompatibilities.

- The fronted needs to perform two REST queries: One to the backend, obtaining the YAML file; the other to the EDMM service. We are performing one more communication with respect to the other approach. Even if the time wasted is negligible, it is always an additional operation that can be avoided.

- An application developer can start Winery with or without EDMM. We must provide an easy way to use Winery with or without the modelling repository. If in the library-based approach an application developer is forced to download the EDMM Winery repository, in this case she has more freedom. The environment is really integrated if it considers also this possibility of having or not the repository. We should think how to manage the repository in order to automate the steps that the application developer should do to plug or unplug it. It might not be easy and in the library-based solution it is not necessary to do this reasoning.

– EDMM becomes really separated from Winery and its reputation. Since Winery is a bigger and more known project, EDMM might benefit if it is integrated as a library. Every time an application developer works with Winery she will find the possibility of using EDMM. The more developers will be adopting Winery, the more users will be hence knowing EDMM (if the latter is always coming with Winery - as with the library-based integration approach). If we use a separated service, the application developer might not be tempted to start the EDMM service and use it with Winery.

4.2.3 Chosen approach

After analysing the pros and cons of both solutions, and also following the desiderata of the University of Stuttgart (who is leading the project), we decided to go for the library-based integration approach. One of the main reasons motivating our choice was that the second approach gives to the integrated environment a microservices-like architecture that is not strongly required in our case, because both Winery and EDMM share the same domain by allowing to edit and manage the application deployment model. Another critical point was about the logic required to create the YAML file. As we already said, the Winery backend needs that logic in both cases because the 'export YAML' functionality cannot be removed. Some of the classes implemented in the *edmm-core* module, managing the different *Deployment Model* representations (i.e. as a YAML or as a graph), must be imported by Winery because they implement the mapping between the Winery data model and the EDMM types. The capability of parsing a Winery *Service Template* into a YAML file, compliant with the EDMM specifications, is needed also in the REST-based integration, so with the second solution we would have imported the EDMM core module as a library anyway. Since the *edmm-core* was already imported as a packaged library, the first approach was the most natural. It is true that there is more code to write with respect to the REST approach and we are

quite limited by the Winery coding guidelines, but the code maintainability is higher. The library can be tested easily because the integration testing can be automated leveraging the Maven building environment. The *Service Template* can be immediately converted to the graph data structure accepted by the DSS and the EDMM Transformation Framework. The direct conversion produces a cleaner code and avoids a useless operation. An application developer is forced to download the modelling-repository and use Winery with EDMM, but it is not a huge problem because the types and templates defined in the repository are well structured and are just a few. Having the EDMM decision support and transformation functionalities always available is surely an advantage. The problem of aligning the shared libraries still remains and we will need to pay attention to that.

4.3 Implementation

To overcome the difficulty of working directly in the Winery official repository[7], we forked it and all the commits where performed in a new branch of the forked code[8]. Only at the end a pull request was issued to the main repo, following the guidelines of Winery.

We created one new Java package (*org.eclipse.winery.edmm*[9]), in the Winery backend, importing the *edmm-core* module[10] as a Maven dependency and providing the necessary logic and mechanisms to call the functions in the library. EDMM uses Java Spring and Spring Boot to build the command-line interface and to find and instantiate the plugins implementing the transformation from EDMM to the supported deployment automation technologies. We decided not to use the same mechanism because this would have required to add an additional dependency to Winery, which is not importing Java Spring at all.

[7] `https://github.com/eclipse/winery`

[8] `https://github.com/UST-EDMM/winery`

[9] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.edmm`

[10] `https://github.com/UST-EDMM/edmm/edmm-core`

The plugin instantiation and wiring is now done by plain Java[11], but still relying on the same configuration file used by Spring[12]. Every time a plugin is added to or removed from the configuration file, the update will be reflected in Winery, which will use the up-to-date set of plugins. The list of plugin objects is necessary to instantiate two fundamental classes, i.e., the *PluginService*[13], which exposes a method to obtain the DSS feedbacks and the *TransformationService*[14], which is responsible for the transformation. These classes are the same that were used by the REST endpoints already present in the DSS and in the EDMM Transformation Framework (under the *edmm-web* package[15]). We reused their logic to avoid code duplication. The DSS does not need any further element as input; while the *TransformationService* needs a *context* object containing the *Deployment Model*, the target technology, the source directory of the artifacts and the target directory of the technology-specific files. In the *org.eclipse.winery.edmm* package we added a class to instantiate properly the *TransformationService* with all the information we just described[16]. The *transform* command, when used via command-line interface, returns a directory with the target technology files, but we need a zipped directory because it will be downloaded by the user of the Winery frontend. Fortunately, the *edmm-core* provides a zip utility that we reused[17]. Before going on, we spend a couple of words about the zip code. In the Winery backend project there is at least

[11] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.edmm/src/main/java/org/eclipse/winery/edmm/plugins/PluginManager.java`

[12] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/resources/pluginContext.xml`

[13] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/core/plugin/PluginService.java`

[14] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/core/transformation/TransformationService.java`

[15] `https://github.com/UST-EDMM/edmm/tree/master/edmm-web/src/main/java/io/github/edmm/web/controller`

[16] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.edmm/src/main/java/org/eclipse/winery/edmm/TransformationManager.java`

[17] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.edmm/src/main/java/org/eclipse/winery/edmm/utils/ZipUtility.java`

a couple of classes providing zip utilities similar, but not equal, to the ones needed by the transformation. This code duplication is not good at all and a common class providing zip and unzip functionalities is surely required. Since the decision concerns the entire Winery project and the goal of our branch was to integrate Winery and EDMM, we opened an issue[18] on the Winery main repository and decided, in the meanwhile, to reduce as much as possible the duplicated code by using the zip function already implemented in the EDMM library.

The REST endpoints for the Winery frontend were created in the *EdmmResource*[19] class under the new *org.eclipse.winery.repository.rest.edmm* package. Each entity defined by Winery is accessed by an URL starting with the type of the entity. The created endpoints can be queries calling the URL relative to the *servicetemplates*, appending at the end the *edmm* string (e.g. *servicetemplates /https%3A%2F%2Fedmm.uni-stuttgart.de%2Fservicetemplates/PetClinic-Cloud/edmm*). There are an endpoint for the DSS and another one for the EDMM Transformation Framework. The first one simply calls the *PluginService* and returns a response containing a list of *PluginSupportResult*[20] objects. The list is taken from the *PluginService* and forwarded to the typescript frontend without being modified by the backend. In this way, if the object changes its internal structure, we need to update only the corresponding typescript interface, without touching the code of the backend. The transformation endpoint calls the class, defined in the *org.eclipse.winery.edmm* package, which sets-up the *TransformationService* and gets the technology-specific files[21]. The endpoint receives the target technology and returns a zip containing the produced files. Both the endpoints have direct access to an object, in the backend, representing the drawn topology. The

[18] `https://github.com/eclipse/winery/issues/499`

[19] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.repository.rest/src/main/java/org/eclipse/winery/repository/rest/edmm/EdmmResource.java`

[20] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/model/PluginSupportResult.java`

[21] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.edmm/src/main/java/org/eclipse/winery/edmm/TransformationManager.java`

mentioned object is used to produce the EDMM YAML after a pre-processing operation, but it is also converted to a *Deployment Model* given in input to the DSS and the EDMM Transformation Framework procedures. The object is directly converted to the EDMM internal representation without being transformed in a YAML string and without being re-parsed to get the graph with components and relations. Doing this, we are saving an unnecessary step with respect to the previous implementation and we get a cleaner conversion between the Winery representation of the topology and the EDMM representation of the model.

The Winery fronted was simply modified to call the new endpoints provided by the backend [22]. It does not require any more an environment variable telling if the EDMM Angular components can be displayed because the DSS and the EDMM Transformation Framework utilities are always available now. The UI concerning the Decision Support System was not changed at all, we added only a transform button per technology. By clicking this button, an application developer immediately gets the zip containing the target technology files.

The *edmm-core* did not require heavy changes, except for a bug fix discussed in Appendix A.1 . As we mention in one of the cons of the library approach, we could not forget about the CLI code because an application developer should have the possibility to execute EDMM also via command-line. We refactored a function in the *TransformationService*[23] so that it is called both by the CLI and by the integration package of Winery (*org.eclipse.winery.edmm*). In this way, the semantic of the transformation is preserved and we do not have duplicated code. Another cons listed in the design section was the alignment of the libraries versions. Both Winery and EDMM use JGraphT[24], but in the Eclipse project the version was ahead with respect to the EDMM one. We could not push to

[22] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.frontends/app/topologymodeler/src/app/edmmTransformationCheck/edmmTransformationCheck.service.ts`

[23] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/core/transformation/TransformationService.java`

[24] `https://jgrapht.org/`

the main repository a code with this mismatch, so we fixed it[25]. This time the version alignment was easy, but it still remains a problem and we should pay attention in the future.

4.4 Testing

Before describing the integration testing, we present how the testing of the backend is performed in Winery because we followed the same pattern used also by other packages. As the backend relies entirely on the winery-repository, to test its functionalities, it is necessary that a valid repository is present and that the backend knows its path. Winery provides an abstract class, *TestWithGitBackedRepository*, that downloads a testing repository from GitHub[26] into a *temp* directory and sets the backend to use it. The test repository has different branches. The *TestWithGitBackedRepository* abstract class allows to checkout a branch before running the test. We created another branch[27] in the testing repository and added there the files present in the EDMM modelling repository[28]. The new branch is automatically checked out before the integration tests execution. We have been able to test the *Service Templates* already present in the EDMM modelling-repository still relying on the Winery test infrastructure.

We tested both the endpoints with the two models stored in the repository[29] trying to avoid duplicated tests[30]. The tests were automatically ran by the Maven test utility (i.e., `mvn test`). For each model, we simulated two REST API calls from the frontend and checked the response. The first call invokes the DSS passing the *Service Template* stored in the EDMM modelling repository.

[25] `https://github.com/UST-EDMM/edmm/commit/4519a2aa13648d4fd779241492300fc35a9777c1`

[26] `https://github.com/winery/test-repository`

[27] `https://github.com/winery/test-repository/tree/edmm`

[28] `https://github.com/UST-EDMM/modeling-repository`

[29] `https://github.com/winery/test-repository/tree/edmm/servicetemplates/https%253A%252F%252Fedmm.uni-stuttgart.de%252Fservicetemplates`

[30] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.repository.rest/src/test/java/org/eclipse/winery/repository/rest/edmm/EdmmResourceTest.java`

We then checked that the returned list of *PluginSupportResult*, used by Winery to display the DSS feedbacks, had been correctly populated. The second call executes the transformation, passing the *Service Template* and the target deployment automation technology, returning a zip file with the technology-specific artefacts. We unzipped the file and checked the number of files contained. The transformation tests were performed passing as target technology *Terraform* and *AWS CloudFormation*. The glue-code, in the Winery *org.eclipse.winery.edmm* package, sets-up some of the classes in the *edmm-core* (*TransformationService* and *PluginService*), then calls directly the function of the *edmm-core* library without adding any further testable logic. It has hence been sufficient to test the API endpoints without repeating the same tests in the *org.eclipse.winery.edmm* package.

Some unit tests were also performed in the *org.eclipse.winery.edmm* package, in particular concerning the plugin instantiation and wiring[31]. We checked that the size of the plugin list was equal to the number of plugins declared in the Spring configuration file.

[31] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.edmm/src/test/java/org/eclipse/winery/edmm/EdmmTests.java`

Chapter 5

Enhancing the DSS

The second contribution of this thesis is an enhancement of the DSS. The original version of the DSS was only informing developers on whether their deployment model was supported by a given deployment automation technology, also highlighting not supported components if this was not the case. The presented enhancement of the DSS supports application developers by also suggesting them how to adapt their deployment models so that they get supported by a given deployment automation technology. The enhanced DSS can also automatically apply adaptations to a deployment model, while at the same time generating the scaffolding code needed to implement the management of newly inserted application components. This chapter presents the design of the enhanced DSS, its implementation, and testing.

5.1 Design

The DSS, in the previous implementation, traversed the graph of the deployment model using a visitor pattern and, for each component, decided if adding it to the list of unsupported components based on the component type. As a result, the list of unsupported components was visualized in Winery without any further information.

Our objective here is to empower the Decision Support System in such a way that it can help application developers to migrate from one technology to another by automating or semi-automating the steps to be done. In particular, an application developer should receive a feedback suggesting how to transform the current model into another that is supported by the chosen technology. The goal has been to ease the application developer decision process, reducing the time wasted to understand which is the right substitution.

5.1.1 Rule System

The solution proposed is a system based on rules defined by the plugin developer: Each technology can define some rules to transform the current topology into another one that is supported by the technology itself. Each rule describes a sub-topology that is not supported by the plugin and provides a possible replacement,i.e., "every component of type *PaaS*, or subtype of it, is not supported; replace it with a *WebServer* hosted on a *Compute*" (Figure 5.1).

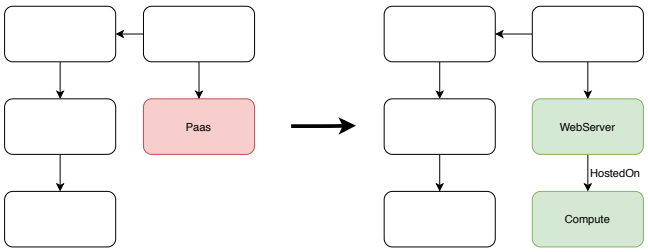

Figure 5.1: Example of a sub-topology into sub-topology replacement. The red box is the sub-topology declared as not supported by the rule. The green boxes represent the sub-topology proposed as replacement by the rule.

The system should search in the graph of the deployment model a sub-graph that matches the sub-topology specified as not supported by the plugin developer. Once it finds the sub-graph, it returns the suggested replacement that Winery will display. The search in the graph, for the sub-topology match, does not use the Visitor pattern any more, as it was done in the previous version.

The search should take a component in input and the entire *Deployment Model* graph. It should be performed for each component of the graph and for each rule. The search should start from the input component and check if in its neighbourhood there is a sub-graph matching the sub-topology of the rule. The match should compare the component types of the nodes. A component sub-type in the current *Deployment Model* should match with a component super-type in the sub-topology defined by the rule.

Each plugin should have its own set of rules, but three *default* rules were individuated. This rules may be applied in every situation because they concern a transformation supported by all the plugins: From a *Platform* type to a topology hosted on a *Compute* node. If the plugin does not specify anything, the default behaviour is applied, so that an application developer is able, at least, to perform a correct, even if quite simple, substitution. The three default rules can be stated as follows:

- *Saas* can be replaced by a *WebApplication* hosted on a *WebServer* hosted on a *Compute* node.
- *Dbaas* can be replaced by a *Dbms* hosted on a *Compute* node.
- *Paas* can be replaced by a *WebServer* hosted on a *Compute* node

The rules should look for component types and the sub-topologies defined should be able involve the *Saas*, *Paas* and *Dbaas* types for sure; but also *WebServer*, *Dbms* and *Database*. The last three node types are supported by all the plugins. The rules should be implemented to be general and should not be specific only to the first three listed types because it is better to treat all the component types in a uniform way. If, in the future, there will be a component type that derives from a *WebServer* or *Dbms* or *Database*, but it is not supported by a certain plugin, the rule system will be able to manage it. The rules should target specific component types like *Tomcat* or *AwsAurora* (yellow boxes in Figure 2.2) as well as general component types like *WebServer* or *Dbaas* (orange boxes in Figure 2.2) because the rule system should not be biased by the types currently

defined. We want it to be maintainable and to scale to new node types. A plugin should be able to define a general rule that says it is not supporting a general component type, like *Saas*, so that if a new component type, sub-type of *Saas*, is added, the plugin should not be modified in order to consider the new type. We should avoid to refactor all the plugins rules every time a new node type is added. A rule should allow to define a general component type as not supported (i.e. *PaaS*), providing also some "special cases"[1] (i.e. *AwsBeanstalk*). The rule will match every component instance of the general class except if its type is in the specified "special cases". When looking for a "special case" the match performed by the graph search must check for the equality between the component types, in the current sub-graph, and the types in the rule "special case". The *AwsBeanstalk* "special case" must match only an *AwsBeanstalk* node, and must not be confused with another node that has *Paas* as super-type. The search for the "special case" sub-topologies should be performed by the same algorithm searching for the not supported sub-topology, but with a different strategy. The component sub-type (i.e. *Auth0*), in the current *Deployment Model*, must not match with the component super-type (i.e. *Saas*), in the sub-topology specified by the rule. Only exact type matching should be allowed (i.e. *Auth0* matches *Auth0*).

The rules must allow *sub-topology into sub-topology* transformations. We should not implement a system considering only *single component into sub-topology* cases, like in the *default* rules. If the rules is able to generally define *sub-topology into sub-topology* replacement, the rule system expressiveness increases because there are more possible rules to define. Sticking the rule system only to *single component into sub-topology* rules strongly limits the possibilities of transformation. We should provide to the plugin (rule) developer a simple way to specify *sub-topology into sub-topology* rules. One possibility is to describe the sub-graph leveraging the EDMM YAML notation and specifications.

[1] The correct terms are *exception*, *rule exceptions*, *exception sub-topologies*. We use "special case" to avoid confusing them with the Java runtime Exceptions (e.g. NullPointerException).

The rule should have a *priority* sorting the order of their evaluation and the order of the displayed results. We may want to check first a plugin defined rule instead of a default one; an application developer may want to see first the replacement suggested directly by the plugin, instead of the default transformation. The application developer should be able to see all the possible replacements and not only the one with highest priority suggested by the plugin developer. The application developer may want to choose the replacement also considering other factors like the cost, so we cannot provide just the solution of the plugin developer because it might be the most expensive. The rule should have a *reason*, stating why the plugin developer has written it and why it should be applied. The *reason* should have three values: *unsupported*, *partly-supported* and *preferred*. The first means that the sub-topology specified by the rule is completely unsupported and it must be replaced. The *partly-supported* means that the sub-topology specified by the rule contains some component types not supported and some other supported. An application developer must anyway replace the entire sub-topology to get a valid deployment model. The last value, *preferred*, means that rule is just suggesting a possible replacement, but it is not forcing it. The topology provided by the application developer is supported, but there is the possibility to apply a replacement that the plugin developer thinks is better. The rule *reason* should be used to compute a *support distance measure* approximately indicating how far is the current topology with respect to the topology supported by a given plugin. The higher is the distance, the bigger the difference between the current topology and the topology supported by the plugin. If the current topology, related to a certain plugin, triggers many *unsupported* or *partly-supported* rules, the distance between the current and the supported topology will be higher; if the *unsupported* or *partly-supported* rules are few the two topologies are closer.

The Winery interface should be updated too, in order to display all the new information provided by the DSS. An application developer needs to be notified about the portion of the topology that is not supported and she should see the replacement suggested. Since we are aiming for more automation, it would

be really good if the replacement is done semi-automatically: The application developer should just press an "apply-rule" button without having to manually delete the nodes in the topology, add the new ones and link them with the relations. There might be the possibility that the application developer does not have the artifact files necessary for the newly inserted components. Adding them in Winery, for each component generated by the rule application, it is time consuming and can be done automatically. For each component operation (i.e., *create*, *configure*, *start*, *stop*, *delete*), an *artifact template* should be generated and it should be wired to the *component type*. We should be able to provide a mechanism that generates placeholder scripts for the components inserted by the rule, and that adds them to Winery. The scripts should be empty because it is not always possible to migrate the *artifact* files. The Winery backend already provides all the functionalities to manage the winery-repository and to add the *artifact*. Our automatic generation should use the backend classes in order not to write duplicated code.

5.2 Implementation

The idea behind the Rule System is to have a mechanisms that takes the drawn deployment model and suggests the replacement that should be done in order to get a model supported by a chosen plugin. The system will search for the portion of the topology that is not supported by the plugin and propose a sub-topology as substitute. Figure 5.2 shows the classes involved in the Rule System and displays the methods and the fields necessary to understand the discussion[2]. The *TransformationPlugin*[3] class was already present. We just added the *getRules* method that should be implemented by every plugin and should return the rules list of the plugin. The *PaasDefault* class is just an

[2] `https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/main/java/io/github/edmm/plugins/rules`

[3] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/core/plugin/TransformationPlugin.java`

example of rule that extends the *Rule* abstract class and that is instantiated by the *TransformationPlugin*.

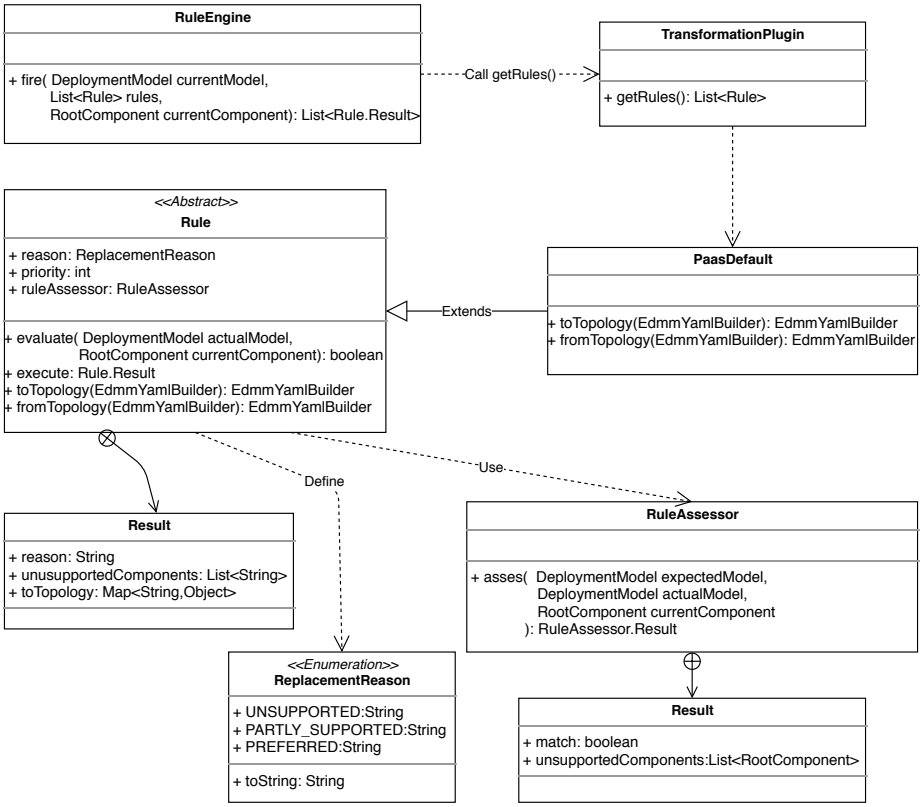

Figure 5.2: Rule System class diagram

The *EdmmYamlBuilder*[4], used in the *Rule* class, is not showed in the Figure 5.2. The *EdmmYamlBuilder* allows to describe EDMM models in a YAML-like way, but using a builder pattern (Listing 5.1). The developer has the feeling of writing a YAML file because the names of the functions provided by the builder mimic the keywords in the EDMM YAML specification[5]. She does not have to instantiate the Java object relative to the components or relations she is describing, but it is sufficient to provide their Java classes. The builder will

[4] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/model/support/EdmmYamlBuilder.java`

[5] `https://github.com/UST-EDMM/spec-yaml`

```
1        EdmmYamlBuilder yamlBuilder = new EdmmYamlBuilder();
2
3        yamlBuilder.component(WebServer.class)
4                   .hostedOn(Compute.class)
5                   .component(Compute.class);
6
7        String edmmYaml = yamlBuilder.build();
```

Listing 5.1: Simple usage of the *EdmmYamlBuilder* class. The generated YAML describes a *Deployment Model* with a *WebServer* component *HostedOn* a *Compute* node.

automatically create the YAML using a bidirectional Java Map that associates components and relations (Java classes) to the corresponding YAML string.

Each rule must declare a sub-topology that is not supported (also called *fromTopology*), and provide a sub-topology as possible replacement (also called *toTopology*). The first topology is involved in the rule *evaluation*, while the latter is included in the content of the rule result. The *Rule*[6] abstract class needs to be extended by the plugin developer to create a new rule. The developer must override two methods of the class and make them return the *fromTopology* and *toTopology*. The composition of both the topologies is performed using the *EdmmYamlBuilder*. For instance, Listing 5.1 shows how to build a *WebServer HostedOn Compute* sub-topology. Once the plugin developer has specified the *from* and *to* topologies, she just needs to add the rule *priority* and *reason* fields, and, if she wants, the rule *exceptions*. We will talk about them later, after explaining how the rule is evaluated and executed.

The responsibility of the rule evaluation and execution is of the *RuleEngine*[7]. This class traverses the topology drawn by an application developer and for each component evaluates every rule returned by the *getRules* function of the

[6] https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/plugins/rules/Rule.java

[7] https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/plugins/rules/RuleEngine.java

TransformationPlugin. If the rule evaluates to true, it is executed and its result is saved in a list. The rule evaluation takes as input the topology drawn by the application developer and the component currently traversed by the *RuleEngine*. It returns true if in the input topology there is a sub-topology similar to the *fromTopology* containing the current component.

The search for the sub-topology match is delegated to another class, the *RuleAssessor*[8]. It performs a graph search in the *fromTopology* and input topology, starting from the node taken as input and following its relations like an expanding stain. The search is guided by the *fromTopology*, meaning that the algorithm follows the graph structure of the *fromTopology* and checks, starting from the input node, if the input topology has a sub-graph with the same structure and the same node types. Every time the algorithm finds a triple 'component-relation-component' that matches, it continues in the search. If there is a mismatch or the *fromTopology* has been completely visited, the algorithm ends returning false in the first case, true in the second. The algorithm implements two kinds of matches: An *equality* match and a *similarity* match. With the first one we are searching for a topology that exactly matches the *fromTopology*, meaning that the components and the relations in the matched topology must be of the same type of those ones in the *fromTopology*. If we have an *Auth0* component in the drawn topology and a rule searching for *Saas*, an equality match will return false because *Auth0* is a sub-type of *Saas*. The *similarity* match, instead, looks for a sub-topology that is similar to the *fromTopology*, meaning that its components and relations must be of the same type or sub-type of those ones in the *fromTopology*. Considering the *Auth0-Saas* example again, a similarity match will return true because the sub-type is allowed. The similarity match is used every time by the rule evaluation, because we want our rules to be flexible and we want to allow rules accepting groups of component types (orange boxes in Figure 2.2), not only fixed to single com-

[8] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/plugins/rules/RuleAssessor.java`

ponent types (yellow boxes in Figure 2.2). The equality match, instead, it is useful when we consider exceptions sub-topologies.

A plugin may not support every topologies of a certain type (e.g., *Paas*), *except* some particular cases (e.g., *AwsBeanstalk*). In this case the plugin (rule) developer cannot specify a rule to replace *Paas* components because such rule will trigger also with *AwsBeanstalk* nodes, that are supported instead. The developer needs a rule stating that *Paas* is not supported, but providing also some "special cases"[9] to the rule. The "special case" sub-topologies allows to fullfil such need. They are defined in the same way as the *from*/*to* topologies, but we search for them only if we found a match with the *fromTopology*. In this case, we perform the graph visit using the equality strategy. We want that the "special case" topology matches exactly and does not get confused with other similar topologies. If one of the "special case" topologies matches (e.g., *AwsBeanstalk*), the evaluate function will return false even if the similarity matching, performed with the *fromTopology* (e.g., *Paas*), returned true.

The execute function returns an object (*Rule.Result*[10]) containing a list of components to be replaced and belonging to the drawn topology; a Map describing the *toTopology* and the *reason* field specified in the rule. This field represents the reason why the rule was written and should be applied. It can have three values: *unsupported*, *partly-supported* and *preferred*. *Unsupported* means that the matched sub-topology is completely not supported by the technology and must be replaced. The number of unsupported rules evaluated true and executed are used to compute a distance measure between the current topology and the model supported by that plugin. In the original DSS implementation, this measure was computed just by counting the size of the unsupported components list. The *partly-supported* reason means that only some components of the sub-topology are not supported at all, but an application developer needs to replace

[9]The terms used in the implementation are *exception*, *rule exceptions*, *exception sub-topologies*. We use "special case" to avoid confusing them with the Java runtime Exceptions (e.g. NullPointerException).

[10]`https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/plugins/rules/Rule.java#L140`

the entire sub-topology to get a compliant model. The *partly-supported* rules are treated in the same way of the *unsupported* when computing the distance measure. The last value, *preferred*, means that the rule was written just to give a suggestion to the application developer proposing a preferred replacement, but not forcing it. If a technology displays only *preferred* rules, does not mean that the topology cannot be transformed, but it just means that the application developer has the freedom of choosing among different deployment models and they will be all equally supported. The number of *preferred* rules is not considered when counting the support distance measure, as could be expected. The *support distance measure*[11] is equal to $\frac{\#unsupported + \#partly\ supported}{\#components\ in\ the\ topology}$. If the number of *unsupported* plus *partly-supported* rules, evaluated true, grows, the distance becomes higher. If there are no *unsupported* and *partly-supported* rules, evaluated true, the distance is zero. We have not talked about the *priority* field. It simply sets the order in which the rules are applied and the results showed. The default rules have the lowest priority.

The rules currently supported by the enhanced DSS are the following:

- **PaasDefault**[12]: "*Paas* component type and its subtypes are not supported, use a *WebServer HostedOn Compute* instead". The *reason* field value is *unsupported*.
- **SaasDefault**: "*Saas* component type and its subtypes are not supported, use a *WebApplication HostedOn WebServer HostedOn Compute* instead". The *reason* field value is *unsupported*.
- **DbaasDefault**: "*Dbaas* component type and its subtypes are not supported, use a *Dbms HostedOn Compute* instead". The *reason* field value is *unsupported*.

[11] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/core/plugin/PluginService.java#L95`

[12] `https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/main/java/io/github/edmm/plugins/rules`

- **CfnPaas**[13]: "*Paas* component type and its subtypes (except *AwsBeanstalk*) are not supported, use a *WebServer HostedOn Compute* instead". The *reason* field value is *unsupported*.
- **CfnDbaas**: "*Dbaas* component type and its subtypes (except *AwsAurora*) are not supported, use a *Dbms HostedOn Compute* instead". The *reason* field value is *unsupported*.
- **Beanstalk**: "We suggest you to replace a *WebServer HostedOn Compute* with *AwsBeanstalk*". The *reason* field value is *preferred*.
- **Aurora**: "We suggest you to replace a *Dbms HostedOn Compute* with *AwsAurora*". The *reason* field value is *preferred*.

These rules should be returned by the *getRules* function, declared in the *TransformationPlugin* (Figure 5.2) and called when the *RuleEngine* is executed. If a plugin does not override that function, the default rules will be used; if a plugin supports every component it must override the function and return an empty list, so no rule will be evaluated. The results of the rules (*Rule.Result*) are packaged in the same object used in the previous implementation (*PluginSupportResult*) and a list containing one of these objects per plugin is returned to the Winery frontend. The *PluginSupportResult*[14] contains also additional information like the support distance measure and the name of the plugin; it is not just a wrap for the result of the rules.

The *checkModel* function and the visitor used before are not necessary any more, therefore they have been eliminated because the DSS is now entirely managed by the rules logic. Now, a plugin developer is only required to specify the rules instead of declaring, for each component type, if it is accepted or not. The concept of supported or unsupported is not bounded to a single component type, but to a sub-topology and can be inferred from the *reason* field of the rule.

[13] `https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/main/java/io/github/edmm/plugins/cfn/rules`

[14] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/main/java/io/github/edmm/model/PluginSupportResult.java`

The Winery UI calls the DSS every time the topology changes and gets the *PluginSupportResult* list. For each plugin it shows the support distance measure and the list of rules that evaluated to true. If the technology supports the model, a transform button is provided to trigger the EDMM Transformation Framework functionalities (as already described in Chapter 4). A new Angular component was added to gather in one place the rule information[15]. Each rule has a colour based on its *reason* field (red, yellow, green), displays the list of components to be replaced and highlights them in the canvas (Figure 5.3, Figure 5.4). The *toTopology* is presented using a YAML-like style, but the EDMM component types strings are replaced with the corresponding types defined in Winery. To do this conversion we query a new endpoint in the *EdmmResource*[16] that returns a Winery-EDMM types Map, where, for instance, the EDMM *Dbms* component type is mapped to the Winery *DBMS* node type. The rule can be applied just by clicking a button. The not supported sub-topology will disappear and a new sub-topology will be automatically created. an application developer has only to order the components graphically as she prefers most, and link that new sub-topology to the remaining part of the previous topology. If the replacement applied by the rule is too generic, the application developer can substitute a component of the generated sub-topology with one of its subtypes just picking it from the palette. The functions used to delete the old components and create the new sub-topology are the ones implemented by the Angular palette component and Angular canvas component (to create and delete the nodes). The Winery-EDMM types Map is fundamental in the creation of the new sub-topology, because the called Winery functions, generating nodes and relations, work with the Winery internal data representation and not directly

[15] `https://github.com/UST-EDMM/winery/tree/edmm-ng/org.eclipse.winery.frontends/app/topologymodeler/src/app/edmmTransformationCheck/edmm-replacement-rules`

[16] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.repository.rest/src/main/java/org/eclipse/winery/repository/rest/edmm/EdmmResource.java#L137`

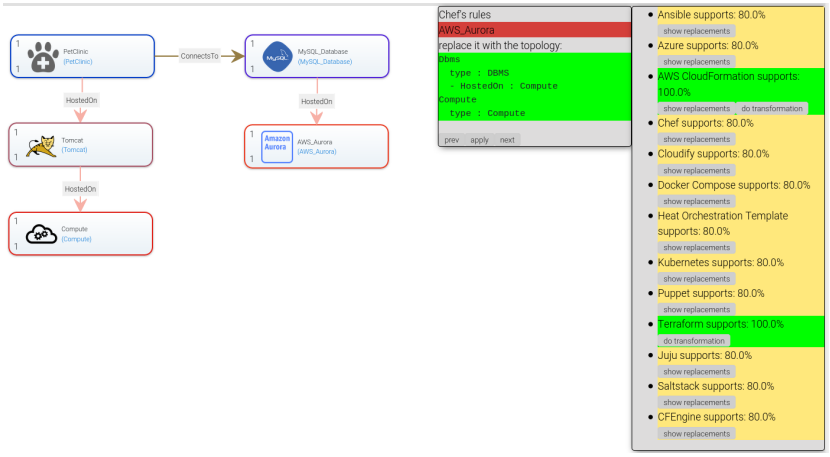

Figure 5.3: Winery snapshot of DbaaS rule. The *AwsAurora* component is highlighted when the mouse is over the rule box.

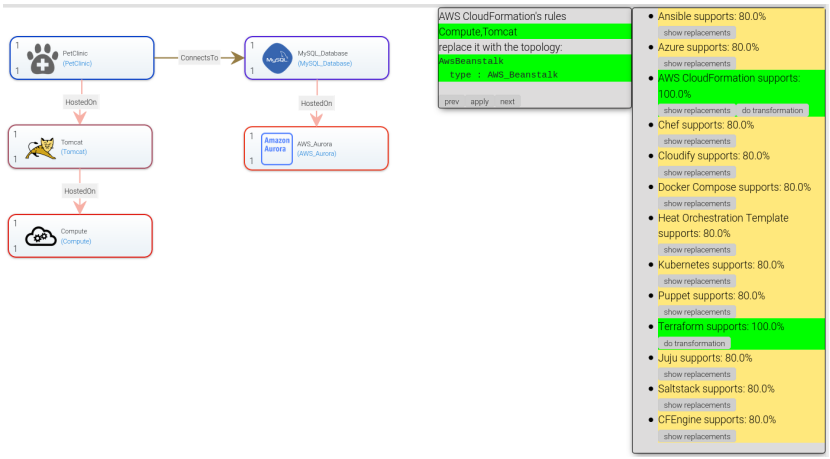

Figure 5.4: Winery snapshot of Beanstalk rule. The *Tomcat* and *Compute* components are highlighted when the mouse is over the rule box.

with the EDMM types. There is no duplicated code nor logic, since the create-delete functions are exposed using an Angular service [17].

Every time a new node is created by the rule application, we call another backend endpoint (exposed by the *EdmmResource*[18]) that creates the placeholder scripts. This operation is repeated for every new node, except the *Compute* one that does not require artifact scripts to be created, configured, or started, for instance. The endpoint code interacts with the winery-repository to create the necessary files for each of the *Operations* defined by EDMM. The steps automatically performed are the following:

1. Define a *lifecycle* in the *Node Type* with the list of *Operations*. The *lifecycle* specifies the "interface" of the node. The *Node Type* corresponds to the EDMM *Component Type*.

2. Create the *Artifact Template* and upload the placeholder script. The *Artifact Template* is a Winery elements used to hold artifact files and metadata. The bash script is empty and needs to be filled by an application developer because it is not always possible to migrate the scripts.

3. Create a *Node Type Implementation*. It acts like a bridge between the *Node Type* and the artifacts. In the *Node Type Implementation* we can register also the deployment artifacts like packaged code or binaries.

4. Link the *Artifact Template* to the *lifecycle* operation in the *Node Type Implementation*.

If the resources, that are automatically created, were already present in the winery-repository, they are not overwritten. If an application developer does not fill the placeholder scripts, there will not be any problem because, once they will be called, no action will be executed.

[17] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.frontends/app/topologymodeler/src/app/services/manage-topology.service.ts`

[18] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.repository.rest/src/main/java/org/eclipse/winery/repository/rest/edmm/EdmmResource.java#L157`

We will now show how the above presented extension of the DSS was used in our motivating scenario Figure 3.1. Our aim is indeed to use it as an example to give a comprehensive view of the features described in this chapter. The focus will hence be on the DSS and we will show how a couple of rules may be applied to that scenario. The application deployment considered is supported by Terraform and by AWS CloudFormation (Figure 5.3, Figure 5.4). The first plugin has an empty rule list since it supports everything, while the second has no rule for *AwsAurora* and does not inherit the *Dbaas* default rule. The boxes, relative to this technologies, are coloured green and present a transform button to retrieve the zip file. The Cfn rules offer to an application developer a possible replacement represented by the *preferred Beanstalk* rule. The rule suggests to replace the *Tomcat* server *HostedOn* the *Compute* node, with an *AwsBeanstalk* component (Figure 5.4). If we performs such substitution the resulting topology is still supported by the AWS plugins.

Consider now another type of substitution involving a deployment automation technology that does not currently support the deployment model. The technologies not supporting the deployment are coloured in red because each of them declares the *Dbaas* default rule that marks as unsupported the *AwsAurora* node and provides as replacement a *Dbms* over a *Compute* node (Figure 5.3). Suppose, for instance, that we wish to deploy with Chef. We can press the "apply" button next to the displayed rule and the *AwsAurora* will be deleted and replaced by the described sub-topology (Figure 5.5). We would now need to manually link the *MysqlDatabase* to the generated *Dbms* component, or we may wish to substitute the *Dbms* with a *MysqlDbms* component and then adjust the relations. The last thing to be done is filling the placeholder scripts created for the *Dbms* with the code necessary to manage the lifecycle of that component. Since the resulting topology is entirely IaaS based, it is supported by all the plugins. This can be seen by checking the colour of the plugins UI boxes that will become green (Figure 5.6).

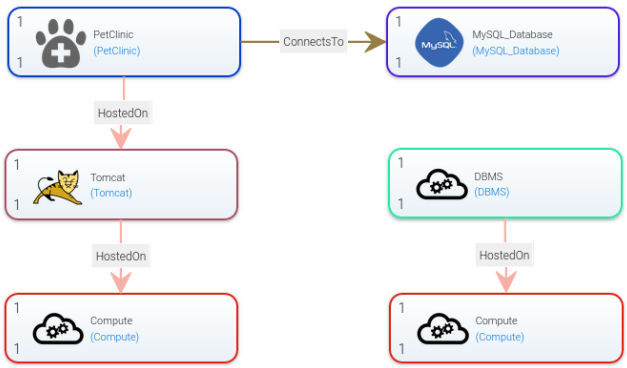

Figure 5.5: Winery snapshot of Dbaas rule replacement

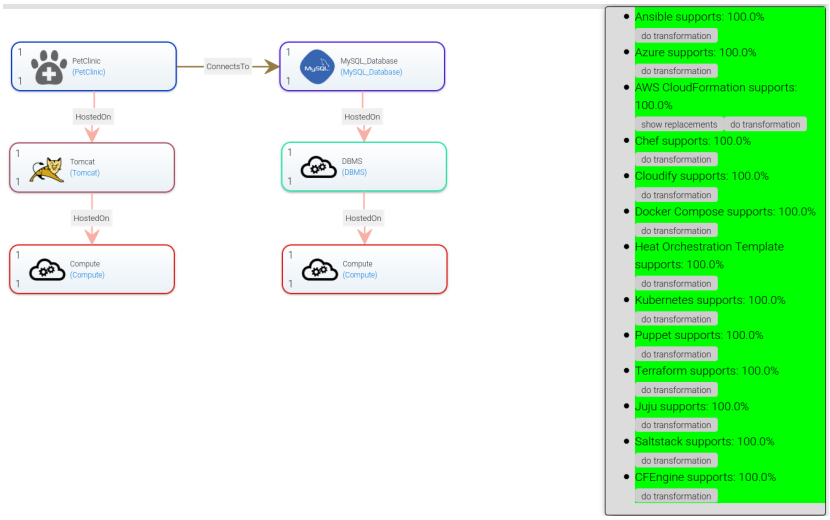

Figure 5.6: Winery snapshot of topology after rule application. The *MysqlDatabase* has been manually linked to the *Dbms* with an *HostedOn* relation.

5.3 Testing

All newly introduced classes (Figure 5.2) were unit tested[19]. We tested the single parts of the mechanism: *Rule*, *RuleAssessor*, *EdmmYamlBuilder*[20]; and then we tested the entire system thanks to the *RuleEngine*. The *EdmmYamlBuilder* was widely used during the testing of the *RuleEngine* and the *RuleAssessor* classes because it allows to build many different topologies, in order to test also edge cases.

The *EdmmYamlBuilder* was tested by using it to create an EDMM YAML string. The string was then converted to a *Deployment Model* and we checked that the model graph was structured as we expected.

The *RuleAssessor* required a more precise testing because the algorithm, searching for the sub-topology match, is not trivial and may hide bugs. The *RuleAssessor* takes in input a component and two topologies, i.e., the *actual* and the *expected* topology. The *actual* topology refers to the one drawn by an application developer; while the *expected* should be the sub-topology defined by the rule (i.e., *fromTopology*). Thanks to the *EdmmYamlBuilder* we instantiated couple of similar topologies, meaning topologies with the same graph structure but different component types, and fed them to the *RuleAssessor*. The *actual* topology given in input had the components of the same type or sub-type of the components in the *expected* topology. We first tested the similarity match with this configuration of topologies, then we swapped the them (i.e, the *actual* topology was passed to the *expected* topology function parameter, and vice-versa) to check that the *similarity* matching returned false. The *RuleAssessor* was directly tested with this edge cases, because, since it is involved in every rule evaluation performed by the *RuleEngine*, the tests of the common usage cases were performed together with the *RuleEngine* tests. We were able to fix

[19] `https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/test/java/io/github/edmm/plugins/rules/RulesTests.java`

[20] `https://github.com/UST-EDMM/edmm/blob/master/edmm-core/src/test/java/io/github/edmm/model/EdmmYamlBuilderTest.java`

some bugs in the *RuleAssessor* graph search algorithm, thanks to some tests that were not passing.

The *RuleEngine* was tested with realistic lists of rules composed by those ones implemented. We checked that the right rules were applied in the right order and we analysed also the results. Thanks to the *EdmmYamlBuilder* we were also able to test a topology similar to the one described in the motivating scenario, so we not only considered small configurations, but also a realistic model. We instantiated the topologies, simulating the deployment model drawn by the application developer, with the *EdmmYamlBuilder* and a list of rules that we wanted to test. We fed the *RuleEngine* with the topologies and the rules and we checked that it produces the *Rule.Result* objects relative to the rules matching the input topology. We verified that the *exception* topologies were correctly matched and checked the rule evaluation order given by the *priority* field.

The two added endpoints in the *EdmmResource* of the Winery backend were tested too. These tests are executed by the Maven test utility, because they are placed in the same test class used for the integration testing described in Chapter 4[21]. The test for the Winery-EDMM types Map, just calls the relative endpoint and checks that the Map contained in the response is well formed. The test for the automatic placeholder generation calls the relative endpoint two times in a row, checking if the files are correctly created and no duplicated file is generated.

[21] `https://github.com/UST-EDMM/winery/blob/edmm-ng/org.eclipse.winery.repository.rest/src/test/java/org/eclipse/winery/repository/rest/edmm/EdmmResourceTest.java`

Chapter 6

Case Study

This chapter analyses four different realistic cases for the deployment of the PetClinic application. The four cases were studied to demonstrate, with metrics, how the integrated EDMM ecosystem (including the enhanced DSS) can simplify the application deployment. The goal was to consider, for each example, the manual writing of the technology-specific files needed to deploy PetClinic with a given deployment automation technology, and to compare it with the case of specifying the deployment in EDMM and automatically generating such files with the integrated EDMM environment. In particular, the four examples correspond to four subsequent deployments of the PetClinic application, starting from the first deployment and going through its migration to different target technologies.

The Key Performance Indicators used to evaluate the developers' effort in each case are the following: Number of *lines of code*, number of *files* and programming/markup *languages* used. Concerning the first two metrics, we counted also the *added* (a), *changed* (c), *deleted* (d) lines of code or files. The KPIs were used also in the case analysing the deployment with the integrated EDMM ecosystem, even if the entire example is performed leveraging the Winery UI and not directly writing files. The reason for this is that comparing UI interactions (integrated EDMM ecosystem) with lines of code ("manual" deploy-

ment) would have not been fair. The KPIs values for the integrated EDMM ecosystem approach were then counted by referring to the EDMM YAML file specifying the deployment. Please also note that the *Artifact* files were not considered in the count. Each of the four cases compares two kind of deployments, with or without EDMM, as we already said. If an artifact files is needed in the "manual" deployment, it will also be needed in the EDMM-based one, and vice-versa. We assumed to already have the winery-repository configured with the EDMM node types and relation types, because the repository is shipped with EDMM and an application developer is not supposed to insert the types by herself. Notice that the modelling-repository had already defined all the *Artifact* files and the *Component Types* required for this case study. We had, however, to treat the case as a realistic scenario, so we considered the repository as it was filled only with the types not strictly specific to the application.

With the above preliminaries, we analysed the following four cases:

1. First deployment of the PetClinic application, with Kubernetes (Sect. 6.1)
2. Migrating the deployment from Kubernetes to other deployment automation technologies (Sect. 6.2)
3. Adapting of the deployment of PetClinic to run on PaaS platforms, and deploying it with Terraform (Sect. 6.3)
4. Migrating the latest deployment of PetClinic to Puppet, therein included its adaptation to enable it to be processed by Puppet (Sect. 6.4)

6.1 First Deployment

We again considered the PetClinic application[1], but with a different deployment specification (Figure 6.1): The PetClinic is installed on a *Tomcat* server, as in the motivating scenario, while the *MysqlDatabase* is managed by a *MysqlDbms*.

[1] `https://github.com/UST-EDMM/spring-petclinic`

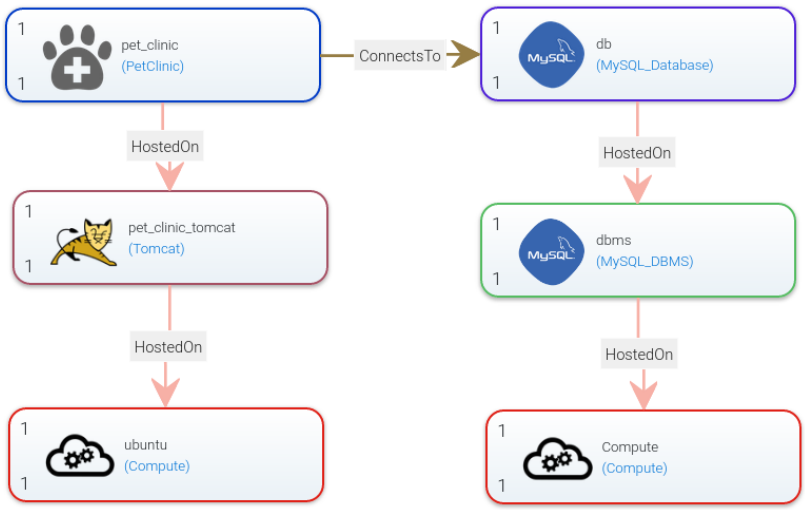

Figure 6.1: PetClinic IaaS deployment

The target technology chosen for the first deployment of the PetClinic application was Kubernetes[2].

KPI	Manual	With EDMM
loc	152 a:152,c:0,d:0	131 a:131,c:0,d:0
files	7 a:7,c:0,d:0	1 a:1,c:0,d:0
languages	Kubernetes YAML, dockerfile	EDMM YAML

Table 6.1: **Case #1:** deployment on Kubernetes without and with EDMM

Table 6.1 shows the comparison between the "manual" deployment and that one using the integrated EDMM ecosystem. From the table, it seems that manually specifying the Kubernetes deployment of PetClinic is already more

[2] `https://github.com/wikilele/master-thesis/tree/master/case-study/case1`

costly than with the EDMM environment. We must say that actually both approaches are equal. The effort, required by the integrated EDMM ecosystem approach, does not lie only in the EDMM YAML file, but in the configuration of Winery. This strategy uses a graphical environment that should be setted-up in a proper way. Winery requires to have all the component types defined and to know the location of the *Artifacts*. The *Compute*, *Tomcat*, *MysqlDbms* and *MysqlDatabase* are already present in the EDMM modelling-repository, hence they are immediately available in Winery. The *PetClinic* component should be defined using the Winery UI, by "sub-classing" a *WebApplication* type. Every *Artifact* file should be added too[3]. If, from one side, the table states that it is easier to adopt the integrated EDMM ecosystem approach, from the other side we have to consider the effort required to set-up Winery. The "manual" strategy might seem more time consuming, but it does not require any particular initial configuration.

In addition, the Kubernetes ecosystem is well known and there is a lot of available material online helping the composition of a Kubernetes deployment model. The EDMM project has been recently released. The material and the community around it cannot be compared with those ones of Kubernetes. For the mentioned reasons, it should be easier to write the Kubernetes-specific files with respect to the EDMM YAML.

6.2 Migrating to other technologies

This second case aims at evaluating the effort needed for migrating the deployment of PetClinic (shown in Figure 6.1) from the former deployment in *Kubernetes* to other deployment automation technologies, i.e., *Ansible*, *CloudFormation* and *Terraform*[4].

[3]The *PetClinic* and the *Artifacts* are already present in the modelling-repository on GitHub, but we must consider a realistic case where the repository does not contain the application-specific elements.

[4]`https://github.com/wikilele/master-thesis/tree/master/case-study/case2`

KPI	Manual	With EDMM
loc	241 a:89,c:0,d:152	0 a:0,c:0,d:0
files	8 a:1,c:0,d:7	0 a:0,c:0,d:0
languages	Ansible YAML	_

Table 6.2: **Case #2:** deployment on Ansible without and with EDMM

KPI	Manual	With EDMM
loc	422 a:333,c:0,d:89	0 a:0,c:0,d:0
files	4 a:3,c:0,d:1	0 a:0,c:0,d:0
languages	Cfn YAML, bash	_

Table 6.3: **Case #2:** deployment on Cfn without and with EDMM

The three tables 6.2, 6.3, 6.4 should be read as if an application developer starts with a *Kubernetes* deployment, then goes on *Ansible*, then on *Cfn* and finally on *Terraform*. It is not really important the effective number of lines of codes counted in the "manual" deployment because it is quite dependent on the chosen technology. What really matters is the magnitude. When "manually" migrating from one technology to the other, almost every old file cannot be reused because each deployment automation technology has its own specific language and way of describing the state of the model. This further highlights the lack of portability of deployment artefacts, which are inherently technology-

KPI	Manual	With EDMM
loc	514 a:181,c:7,d:333	0 a:0,c:0,d:0
files	4 a:1,c:1,d:2	0 a:0,c:0,d:0
languages	terraform, bash	_

Table 6.4: **Case #2:** deployment on Terraform without and with EDMM

specific. With the integrated EDMM ecosystem the file created at the beginning is instead reused in all cases, as just one click is needed to enact each migration. The effort required to set-up the *Artifacts* in the Winery UI is zero, because all the files inserted in the modelling-repository in case #1 are available.

In addition, the expertise required is very different from one deployment strategy to the other. Every time we "manually" change technology, we should know how to map the current resources with those ones required by the target technology. EDMM, instead, has a unique and uniform way of specifying the state of the model; the mapping is then performed automatically. The language metric helps to understand this point, as we first have to translate the Kubernetes deployment in a deployment specified with Ansible's YAML-based domain specific language (DSL); we then have to translate the latter to CloudFormation's YAML-based DSL; finally, we have to translate the deployment to Terraform's specific deployment files. Again, each time we migrate technology, we have to rewrite (almost from scratch) the deployment files for such technology in a different language, hence being required to learn and understand the language. With the integrated EDMM environment, instead, it is enough to learn EDMM and to specify the deployment only once.

6.3 PaaS deployment

In the third case, we fixed the technology (*Terraform*), but modified the topology[5]. The reshaping of the model was guided by the live feedback of the enhanced DSS. The application is always the same, what changes is the infrastructure that now is PaaS based instead of IaaS based. We replaced the *Tomcat* server with *AwsBeanstalk* and the *MysqlDbms* with *AwsAurora*. We used provider specific services, supported by *Terraform*, meaning that the deployment must go on AWS cloud (Figure 6.2). The displayed topology was modelled thanks to the Winery UI. An application developer, going for the "manual" deployment, can reproduce the modelling step on a paper sheet or using other non-integrated tools.

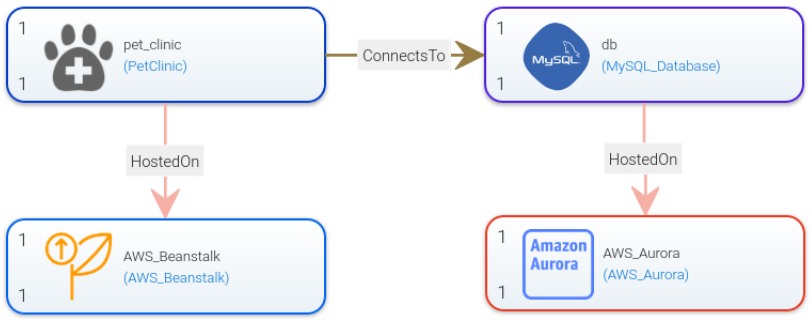

Figure 6.2: PetClinic PaaS deployment

The results are shown by table 6.5 and are quite similar to those ones presented in the second case. The table should be red with respect to table 6.4: An application developer once having deployed an IaaS based topology with *Terraform*, now wants to leverage PaaS offerings still keeping the chosen deployment technology. As in the previous case, also here the integrated EDMM ecosystem approach wins because just some UI interactions are sufficient to perform the migration. The EDMM YAML file needed to be modified in order to change the configuration of the infrastructure. Even if the table shows the changes using the KPIs, the file was not modified directly, but we changed graphically

[5] https://github.com/wikilele/master-thesis/tree/master/case-study/case3

KPI	Manual	With EDMM
loc	253 a:65,c:0,d:173	137 a:59,c:0,d:78
files	2 a:0,c:1,d:1	1 a:0,c:1,d:0
languages	terraform	EDMM YAML

Table 6.5: **Case #3:** deployment on Terraform without and with EDMM

the topology. These changes were reflected to the file by the integrated EDMM ecosystem.

With the integrated EDMM ecosystem it is easier to understand if the modelled topology can be supported by the chosen plugin. Every time the graph changes the application developer gets the feedback from the DSS and she can immediately verify if the new model is supported. The "manual" way requires instead to study the *Terraform* documentation to check which services are supported and how to instantiate them.

6.4 Migrating back to IaaS

The last case analyses the situation in which an application developer wishes to migrate to a deployment automation technology that is not supporting the current topology. Starting from the model of the previous case (Figure 6.2), we wished to switch to *Puppet*[6]. The DSS alerted us, showing that the technology we were choosing was not supporting the model because of the *AwsBeanstalk* and *AwsAurora* nodes. The integrated EDMM ecosystem suggested also one replacement for each of the components based on the *PaasDefault* and *DbaasDefault* rules. By applying such rules and adjusting a bit the component types,

[6] `https://github.com/wikilele/master-thesis/tree/master/case-study/case4`

we were able to get a topology analogous to that previously shown in Figure 6.1, that is supported by all plugins because it can be transformed to artefacts that can be deployed by all supported technologies. In the "manual" deployment approach, we were instead required on our own to understand that *Puppet* cannot handle the topology and should find a way to replace the unsupported components with resources available in the chosen technology. Table 6.6 shows

KPI	**Manual**	**With EDMM**
loc	248 a:168,c:0,d:80	137 a:78,c:0,d:59
files	12 a:11,c:0,d:1	1 a:0,c:1,d:0
languages	puppet	EDMM YAML

Table 6.6: **Case #4:** deployment on Puppet without and with EDMM

the KPIs comparison. The approach with the integrated EDMM ecosystem required to revert the changes done to the EDMM YAML file during case #3. The file was not modified directly, but we used the Winery interface and the rule application to do it.

Once again the integrated EDMM ecosystem allows to reuse the defined types and templates; in addition the modelling and the transformation are highly integrated and automated. What is however not highlighted by the table is the reduction of the effort required by an application developer, who does not need to figure out how to adapt the topology because it is the system that suggests the correct replacements.

Chapter 7

Related Work

Automating the deployment of applications on the cloud is a problem that has raised lot of interest. There are different solutions proposed engaging a wide spectrum of topics from standards, deployment tools, cloud modelling languages (CML), and provider-agnostic deployment models.

The OASIS standard TOSCA [17, 18] is probably one of the most popular efforts in this direction. TOSCA specifies a standardized language to describe multi-component applications in a portable way. The model can then be deployed on cloud if the provider supports the OpenTOSCA runtime [5]. Even if EDMM modelling is based on Winery, which allows to draw TOSCA topologies, our approach is quite different because we do not require additional support from the cloud platforms. We just transform deployment models to technology-specific files that can be used, later, to provision the cloud infrastructure, since they are already supported by the platforms.

Bergmayr et al. [3] reviewed and compared different cloud modelling languages. Vergara-Vargas and Umaña-Acosta [20] developed a new Architecture Description Language that includes software deployment based on a model-driven approach that features components and relations among them. Chen [7] describes a unified view to model data based on entities and relationships. GENTL [2] is a CML to express topologies. Breitenbücher [6] proposed a graph-

based description language to model management activities. EDMM has many similarities with the above approaches. It describes a modelling language and it uses a graph representation for the model based on component and relations. However, the other approaches lack a support for deploying modelled applications with production-ready deployment automation technologies

There exist other models and frameworks that ease the cloud-based deployment of multi-service applications, but none of them offers the same support for multiple production-ready deployment automation technologies as the integrated EDMM ecosystem does. Di Cosmo et al. [9, 8] and Guillén et al. [13], both share with us the idea of generating deployment models from provider-agnostic specification of a multi-component application and its expected state. The solution proposed by Di Cosmo et al. aim at having an high-level description of the application and its configuration to target OpenStack cloud deployments. Guillén et al. presented a framework to deploy multi-component applications to different cloud providers. Both the source code and the metadata representing the deployment model should be fed to the framework in order to get *Cloud Artifacts* that can be deployed on different clouds even if belonging to the same application. Their approaches can however be applied only to IaaS and PaaS cloud. The integrated EDMM ecosystem instead also supports the deployment of SaaS components.

Alipour and Liu [1] designed a model-driven method specific for managing auto-scaling strategies between multiple cloud platforms. It eases tracing the configuration and maintaining consistent the deployment. A Cloud Platform Independent Model is mapped to a Cloud Platform Specific Model that can finally be handled by deployment tools, resulting in a technique very similar to the EDMM one. The solution by Alipour and Liu hence complements ours, as they support application developers in configuring the auto-scaling strategies of their applications. EDMM and its accompanying integrated environment is instead intended to enable specifying application deployments only once, while at the same time enabling to actually enact such deployments with multiple different existing technologies. In addition, neither Alipour and Liu [1] nor Di

Cosmo et al [9, 8] or Guillén et al [13] provide an integrated system featuring also decision support while specifying multi-component applications.

It is worth mentioning also the work done by Wettinger et al. [21] about automating the deployment of cloud application in a middleware-oriented way. The paper presents a distinction between *application-oriented* deployment with respect to *middleware-oriented*. The first is specific to a particular application and it is not reusable. The deployment plans are tailored on the application components and on the PaaS or IaaS nodes they rely on. The *middleware-oriented* deployment, instead, is not bound to a specific application but aims at providing highly reusable deployment plans targeting middleware components (e.g. databases, web servers). EDMM provides application-specific deployment models because the files produced by the EDMM Transformation Framework derive directly from the modelled topology. Following the work of Wettinger et al. we could think about adopting a middleware-oriented approach to produce the technology-specific files by composing the reusable plans.

Other approaches worth mentioning are those focussing on providing some decision support for cloud-based application deployments, even if from a different point of view. Farshidi et al. [12] introduced a DSS to ease the selection process of the most suitable IaaS provider based on cost and security, looking at it as a multi-criterial decision making problem. Khajeh-Hosseini et al. [14] proposed a modelling tool that produces cost estimates derived from the migration to public IaaS clouds and a spreadsheet that outlines benefits and risks of using IaaS offerings. The tool allows to model the application, data, infrastructure requirements, and computational usage patterns. Both solution hence support developers in selecting the most appropriate IaaS clouds for their application deployments, while at the same time focusing on monetary costs. The objective of the integrated EDMM ecosystem spans over all the IaaS, PaaS and SaaS stacks. It has the objective of easing the migration to different declarative deployment technology and different infrastructure resources, without making considerations about the costs.

In summary and to the best of our knowledge, ours is the first solution that allows to automate the cloud-based deployment of multi-service applications targeting different production-ready deployment automation technologies, while at the same time supporting application developers in deciding how to adapt their deployment among IaaS, PaaS, and SaaS components, so that such deployment get supported by a given deployment automation technology.

Chapter 8

Conclusions

The Essential Deployment MetaModel allows to describe the desired state of a multi-component application in a declarative fashion. The specified deployment model is not related to any specific deployment automation technology. The EDMM ecosystem is composed by the EDMM Modelling tool, based on Eclipse Winery, the EDMM Decision Support System, and the EDMM Transformation Framework.

In this thesis, we presented an integration of the tools accompanying EDMM into a single EDMM ecosystem. The new integration gives to an application developer the possibility of modelling the desired deployment while having the DSS feedback always available and the transform utility just one click away. The Decision Support System was also improved and now implements a rule-based logic that allows to automatically adapt a specified multi-component application deployment so that it gets supported also by a chosen technology (if the latter was not supporting it yet, of course). The feedbacks displayed to an application developer contain suggestions on how to replace portions of the currently specified deployment with other portions that are supported by the chosen technology. What really stands out is the smoothness of the entire deployment process. An application developer has the freedom of changing the deployment model, checking if it can be deployed with a given deployment automation technology,

reshaping it or going back to a previous deployment model without having to commit to a single technology and focusing on the application rather than on the tool. One of the goal was to leave to the application developer only business logic related decisions by automating the other tasks as much as possible. We realised this with the integration, but also with the semi-automatic rule application and placeholder script generation.

To validate our work we discussed a case study that demonstrate how the integrated EDMM ecosystem eases the deployment and migration of applications across multiple deployment automation technologies, if compared with manually performing the same tasks. The case study used as KPIs (Key Performance Indicators) the following three metrics, i.e, the number of *lines of code* and *files* written, and the *languages* involved. The technology-specific files, required by the manual approach, were compared to the EDMM YAML file by using the KPIs, even if the latter is generated graphically by the Modelling tool. The results state that: If a developer does not wish to commit to a given technology but rather prefers to easily migrate from one deployment automation technology to another, the EDMM-based solution turns out to be much more helpful, especially for inter-technology migrations after the first application deployment.

At the same time, it is important to note that the integrated EDMM ecosystem currently features a limited number of rules, mainly because of the limited amount of supported component types. The rules, in particular *Default* ones, can be used to treat the component types in a uniform way. This is helpful when a new node type is added because it does not lead to re-write all the rules: The new type will be captured by the *similarity* matching. In the future, we aim to extend the set of supported component types in order to add more rules. With a larger number of component types and rules, the integrated EDMM ecosystem could become even more useful.

We also plan to work on improving the support provided by the integrated EDMM environment. For instance, the rules, exploited to adapt deployment models to work with given technologies, may be provided with a cost parameter, stating how much does it cost to apply the suggested replacement. We might

have different rule suggesting different replacements, and having costs associated with them would enable developers to select the "cheapest" adaptation.

In addition, it is often possible to adopt a middleware-oriented deployment instead of an application-oriented. The integrated EDMM environment could be adapted to also support deploying applications on top of middlewares [21].

Bibliography

[1] H. Alipour and Y. Liu. "Model Driven Deployment of Auto-Scaling Services on Multiple Clouds." In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2018, pp. 93–96.

[2] Vasilios Andrikopoulos et al. "A GENTL Approach for Cloud Application Topologies." In: *ESOCC*. 2014.

[3] Alexander Bergmayr et al. "A Systematic Review of Cloud Modeling Languages." In: *ACM Comput. Surv.* 51.1 (2018). ISSN: 0360-0300. DOI: 10.1145/3150227. URL: https://doi.org/10.1145/3150227.

[4] T. Binz et al. "Formalizing the Cloud through Enterprise Topology Graphs." In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 742–749.

[5] Tobias Binz et al. "OpenTOSCA — A Runtime for TOSCA-Based Cloud Applications." In: *Proceedings of the 11th International Conference on Service-Oriented Computing - Volume 8274*. ICSOC 2013. Berlin, Germany: Springer-Verlag, 2013, pp. 692–695. ISBN: 9783642450044.

[6] U. Breitenbücher. "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements." 2016.

[7] Peter Pin-Shan Chen. "The Entity-Relationship Model—toward a Unified View of Data." In: *ACM Trans. Database Syst.* 1.1 (1976), pp. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440. URL: https://doi.org/10.1145/320434.320440.

[8] Roberto Di Cosmo et al. "Automated Synthesis and Deployment of Cloud Applications." In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 211–222.

[9] Roberto Di Cosmo et al. "Automatic deployment of services in the cloud with aeolus blender." In: *Service-Oriented Computing - 13th International Conference, ICSOC 2015, Proceedings*. Ed. by Alistair Barros et al. Vol. 9435. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 13th International Conference on Service-Oriented Computing, CSOC 2015 ; Conference date: 16-11-2015 Through 19-11-2015. Springer-Verlag London Ltd, 2015, pp. 397–411.

[10] *Eclipse Winery GitHub Workflow*. URL: `https://winery.readthedocs.io/en/latest/dev/github-workflow.html`.

[11] Christian Endres et al. "Declarative vs. Imperative: Two Modeling Patterns for theAutomated Deployment of Applications." In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27.

[12] S. Farshidi et al. "A Decision Support System for Cloud Service Provider Selection Problem in Software Producing Organizations." In: *2018 IEEE 20th Conference on Business Informatics (CBI)*. Vol. 01. 2018, pp. 139–148.

[13] Joaquín Guillén et al. "A service-oriented framework for developing cross cloud migratable software." In: *Journal of Systems and Software* 86.9 (Dec. 2013), pp. 2294–2308.

[14] Ali Khajeh-Hosseini et al. "Decision Support Tools for Cloud Migration in the Enterprise." In: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. USA: IEEE Computer Society, 2011, pp. 541–548.

[15] Oliver Kopp et al. "Winery – A Modeling Tool for TOSCA-Based Cloud Applications." In: *Service-Oriented Computing*. Ed. by Samik Basu et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 700–704.

[16] Michael Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007. ISBN: 0978739213.

[17] *OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[18] *OASIS: TOSCA Simple Profile in YAML Version 1.2*. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csd01/TOSCA-Simple-Profile-YAML-v1.2-csd01.html.

[19] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review." In: *Journal of Systems and Software* 146 (2018), pp. 215–232. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.09.082.

[20] Jeisson Vergara-Vargas and Henry Umaña-Acosta. "A model-driven deployment approach for scaling distributed software architectures on a cloud computing platform." In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)* (2017), pp. 99–103.

[21] J. Wettinger et al. "Middleware-Oriented Deployment Automation for Cloud Applications." In: *IEEE Transactions on Cloud Computing* 6.4 (2018), pp. 1054–1066.

[22] Michael Wurster et al. "Technology-Agnostic Declarative Deployment Automation of Cloud Applications." In: *Service-Oriented and Cloud Computing*. Ed. by Antonio Brogi, Wolf Zimmermann, and Kyriakos Kritikos. Cham: Springer International Publishing, 2020, pp. 97–112.

[23] Michael Wurster et al. “The EDMM Modeling and Transformation System.” In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Ed. by Sami Yangui et al. Cham: Springer International Publishing, 2020, pp. 294–298.

[24] Michael Wurster et al. “The essential deployment metamodel: a systematic review of deployment automation technologies.” In: *SICS Software-Intensive Cyber-Physical Systems* (Aug. 2019).

Appendices

Appendix A

Bugfixes

This appendix reports some bug fixes performed during the development of the thesis. The aim is to show that our contribution was not only limited to the integration, improvement of the DSS and validation, but we corrected also some identified bugs.

The appendix is divided in two sections. The first one presents a bug concerning components with two or more relations of the same type and it requires an in-depth discussion because the solution is not trivial. The second section lists some issues found in the Terraform plugin and notifies that they are solved.

A.1 More relations, same type

The EDMM metamodel allows a component to feature two outgoing relations of the same type (e.g., two *ConnectsTo* relations modelling the fact that it connects to two different other components), but in the implementation that was not considered. An application developer could have hence fed the EDMM Transformation Framework with a YAML file describing a component with two relations of the same type, but, when the YAML got parsed, the first relation overwrote the second one. This bug was present also in the Winery backend

during the generation of the YAML file. The backend was only capable of producing YAML files with components having relations of different types.

In Section 2.3 we said that the YAML is converted to a graph with component as nodes and relations as edges. Between the YAML representation and the component-relation graph object there is and additional step. The YAML is first converted to an *EntityGraph*, then the graph is *normalized*, and finally the data structure with components and relations is instantiated. The *EntityGraph* is used to collect the YAML information and then it is traversed to instantiate the graph used by the plugins. Each node of the *EntityGraph* is an *Entity* which is identified by an *EntityId*. This last one is a string reflecting the position of the entity in the parsed YAML file. For instance, if we have a YAML describing a *WebApplication* connected to two *Saas* components, the *EntityId* of the relations would be something like *0.components.WebApplication.relations.connects_to*. It is possible to refer to Figure A.1 to understand how the ids follow the YAML structure.

The bug was caused by the impossibility of adding to the graph two entities with the same id, because, every time a new node is inserted in the *EntityGraph*, the graph library checks that its id is not equal to the ids of the already present nodes. The relations belonging to the same components were differentiated only by the last part of their id, corresponding to the relation type. It was impossible for the graph library to distinguish the *Entities* of relations with the same type and same source component because the ids were equal. The bug existed also in the Winery backend when creating the YAML file. The Java code takes the Winery topology object, converts it to an *EntityGraph* and then the graph is transformed to a YAML string (without being *normalized*). When Winery parsed two relationships belonging to the same node of the topology and inserted them to the *EntityGraph*, just the first one was added, for the reasons above.

The solution is not trivial because we needed to find a way to distinguish the relations without affecting the YAML generation[1]. The produced file should

[1] `https://github.com/UST-EDMM/edmm/commit/b32e1f531603cebb1b3a3c7f7be8fbe1ef1431d6`

not have been overburdened with additional information useful only to the EDMM Transformation Framework and meaningless from the application developer point of view. For instance, we could not add an index to each relation in the YAML file, because it influences the readability. To keep the YAML generation clean we overwrote the *Object.equals* function relative to the *Entities* managing the relations, in order to consider also the target of the relation and not only the id when adding the node to the graph. Once the *EntityGraph* has all the relation nodes correctly inserted, the '*EntityGraph* into YAML' transformation performed by Winery will produce correctly an EDMM YAML file devoid of unnecessary information. To solve the problem in the EDMM Transformation Framework we adjusted the graph *normalization*. This procedure is executed to expand the *EntityGraph* with additional information that is useful when instantiating the component-relation graph. The *normalization* is not performed when the *EntityGraph* is converted to a YAML string because we want a clean YAML describing only the essential information. The bugfix consisted in adding a unique number to the relation *EntityId*, every time the *normalization* parses a relation, obtaining something like: *0.components.WebApplication.relations.0.connects_to*, so that each relation can be distinguished from the others. Figure A.1 shows graphically the steps performed to convert a EDMM YAML string into a component-relation graph. The *EntityGraph* are represented by a printed list of the nodes *EntityIds*. The figure was obtained after the application of the bugfix, hence the two relations with the same type are correctly considered.

A.2 Terraform issues

This section lists the bugfixes performed on the Terraform plugins. The bugs and their solutions are quite simple. We report them here for sake of completeness[2].

[2] `https://github.com/UST-EDMM/edmm/commit/ee84224f41a467a83768f9635207482493179a1a`

- When transforming an IaaS based topology with two *Component* nodes, one of the produced files containing environment variables was overwritten so not all the variables were available. Now, all files are suitably created.
- When using the command-line interface of the EDMM Transformation Framework, the generated directory with the technology-specific file was not recreated. The plugin appended the produced content to the old files when the transform command was executed a second time. Now, each time the transformation is performed, the target directory is cleaned and the files do not have duplicated content.
- When transforming a topology with *AwsBeanstalk* and *AwsAurora*, the artifact files were not copied in the target directory. Furthermore, the Terraform file was not configuring the *MysqlDatabase* on top of the *AwsAurora*. Now, the artifact files (*petclinic.war*, *schema.sql*, *mysql-configure.sh*) are moved to the generated directory and the *.tf* file declares a resource to configure the database.

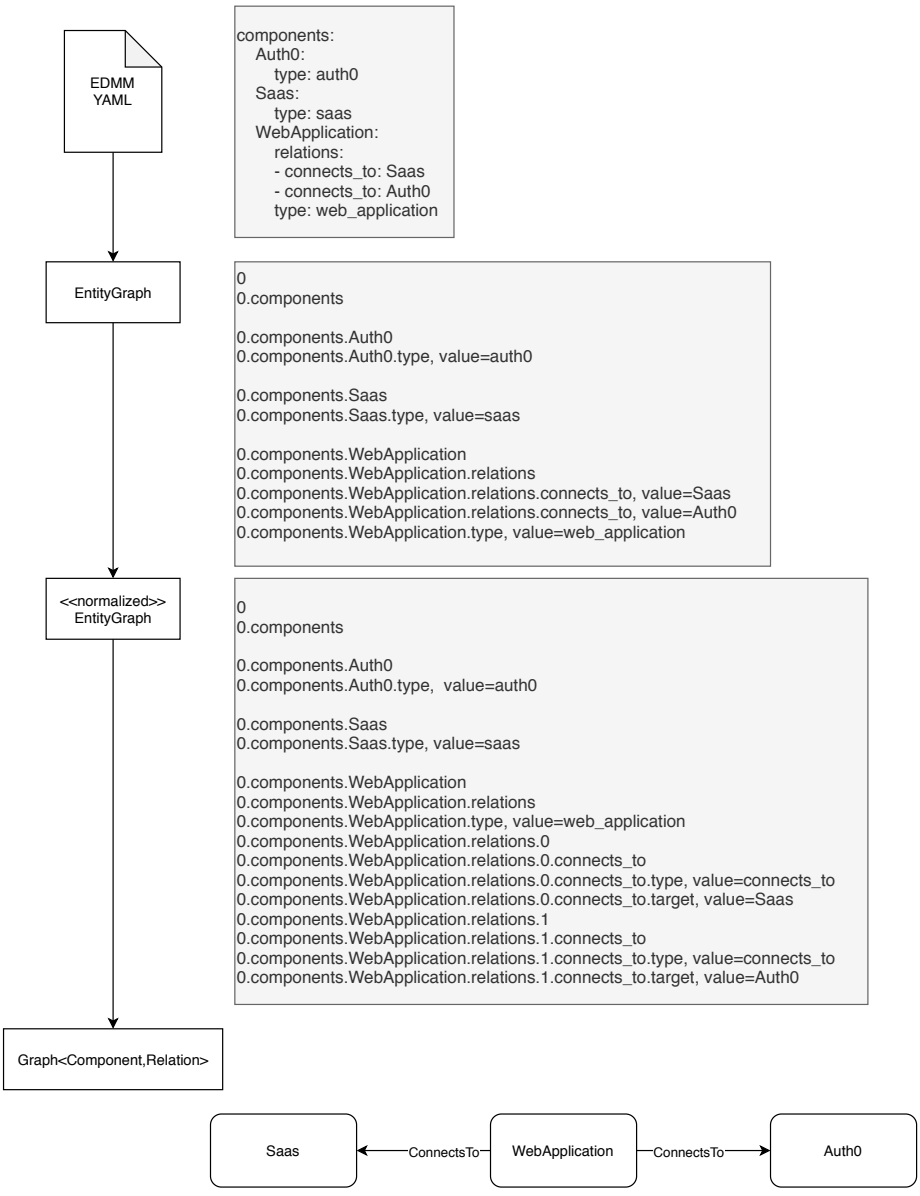

Figure A.1: Example of conversion from a EDMM YAML to a component-relation graph. The *value* attribute next to the *ids* is a field belonging to some *Entity* nodes. It used for additional specifications.