Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Framework for Orchestrating Application Deployments Using Multiple Deployment Technologies

Felix Diez

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Michael Wurster, M. Sc. Karoline Wild, M. Sc.
Commenced:	December 2, 2019
Completed:	July 28, 2020

Abstract

With current trends like continuous delivery the application deployment process is automated as much as possible. Manual steps would slow down this process substantially and are prone to mistakes, so they are reduced where the possibility exists. To achieve a higher level of automation many tools are in use with varying features. In practice, often more than one technology is used to get the best of all worlds. However, with the usage of more tools the complexity of managing their interactions increases. This thesis provides a framework that enables and manages the deployment with multiple deployment technologies. This is achieved by first defining a technology agnostic model on the basis of the Essential Deployment Meta Model. Afterwards, concepts for the automatic transformation to technology specific models and then the automatic execution with the desired tools are described. As a validation for the concept a prototypical implementation is provided.

Contents

List of Figures

List of Listings

Acronyms

BPMN Business Process Model and Notation.

CLI Command Line Interface.

CPSM Cloud Provider Specific Model.

DB Database.

DBMS Database Management System.

DTSM Deployment Technology Specific Model.

EDMM Essential Deployment Metamodel.

JSON JavaScript Object Notation.

OS Operating System.

SSH Secure Shell.

TOSCA Topology and Orchestration Specification for Cloud Applications.

VM Virtual Machine.

YAML YAML Ain't Markup Language.

1 Introduction

With principles like Continuous Delivery becoming more prevalent, the deployment process needs to be highly automated [15, 17]. Additionally, with the trend of microservices a lot of services have to be managed [25]. Nonetheless, it must be possible to release a new version within hours or even minutes. Without strong automation, this approach would be very error prone [21] and not as quick as required. To allow this reliable and continuous delivery, many different tools and techniques were created. The topic is also known under the term DevOps [31]. One big pillar is the reoccurring key concept to describe infrastructure and configuration as reproducible and reusable code [4, 18]. In other words, a model holds all the information, which can be executed. There are many tools that work with this principle. Some examples are Terraform[13], Chef[22] and CloudFormation[2]. Apart from this commonality, they have many differences as well. The modeling languages and the feature sets may vary widely. Furthermore, some are restricted to a specific cloud provider, others require a running infrastructure node or depend on a specific platform.

In practice, this often leads to the usage of more than one technology to fit all the requirements. Additionally, it's likely that different teams in the same company use a different set of tools. This increases the difficulties for developers vastly. Therefore, it could lead to a increased complexity of the whole development process. Some reasons are that all the used tools have to be learned to some degree and the communication with other teams becomes more difficult because the differing tools may have slightly different terms and approaches. Getting a comprehensive overview over the deployment landscape is more difficult as well, because different parts are described in different tools or languages. The "big picture" has to be reverse engineered from the different tools by hand. In practice this may often lead to a model that is outdated or doesn't describe the real deployment at all. Furthermore, the orchestration is harder. When different tools are used there is not an easy way to see what needs to be executed in sequence to start a component. For example, in a simple case an Application is deployed with one technology and the database with another one. The application does not work without the database so it needs to be started first. Therefore, to guarantee a flawless deployment the order of execution needs to be coordinated and developers need to keep track of it. Looking at even bigger deployments with more possibly even more technologies the coordination can clearly become a big problem.

To reduce the difficulties that come with using different tools a solution could be to define a model that is agnostic of the underlying tech. This model is then the basis for reasoning about the complete deployment without relying on too many tool specific traits. At the end this can then be transformed to an executable model that is technology specific. A first possible step towards this solution represents the Essential Deployment Metamodel [29]. This model extracts the fundamental modeling entities that 13 different deployment technologies have in common. Through this model teams can integrate their different tools and nonetheless have a general overview that is independent of specific technologies. Based on this model the EDMM Transformation Framework [28] was built that transforms the model to one of the 13 target technologies. But, so far, there is no possibility

defined to split the deployment into multiple technologies. To achieve our goals of unifying a heterogeneous landscape, this framework needs to be extended. It needs the ability to transform a model where different components in the model can have a varying deployment technologies.

Furthermore, an automatic executor would be desirable as well that has an overview which components depend on each other and deploys them in the correct order with the chosen technologies.

This thesis will first explain the backgrounds and visualize a motivating scenario in Chapter 2. Subsequently, other work in this area is examined in Chapter 3. The main part builds the introduction of a concept with the possibility to split the deployment in Chapter 4 horizontally and vertically in different technologies. Additionally, it automatically orchestrates the deployment process. At last, a prototype is shown in Chapter 5 and a conclusion is given Chapter 6.

2 Background and Motivation

To get a common of understandings of basics that are used throughout the thesis, small introductions are given, followed by a motivating scenario.

2.1 Deplyoment Technologies

In the center of all the work, there are the different deployment technologies. For large scale services, 1000 or more machines are a common occurrence [24]. Additionally, different components have to work together and are interdependent in a complex system. This cannot be handled manually anymore and would be extremely error prone. For example, some servers could be forgotten or applications are started in the wrong order. The complexity is handled through describing them as code that is then automatically executed by a tool. As a result, the components are reusable, better maintainable and it is reproducible. Only the code has to be checked to see the current configuration. Not every machine needs to be checked individually anymore. On a very basic level deployment technologies are divided between declarative and imperative principles. In the declarative approach the desired goal state is described by a user. If a command is issued the tool will execute all necessary steps to get to the goal state. This state is described in a model that contains all components, configurations, their dependencies and their interactions. What steps are necessary to reach it is not of concern for the user [9].

In contrast, with an imperative approach, every step needs to be written down. Regarding a scenario where first a virtual machine must be deployed and then software is installed on top of it. Imperatively, the solution is to write a procedure that step by step lists what has to be done. First, call the API to start a VM, then access the machine with SSH, then install software and so on. In comparison with the declarative approach, the deployment model is enough. The tool figures out the concrete operations and executes them on its own. A more comprehensive comparison between the two categories was made by Endres et al. [9]

Declarative methods have proven to be well suited for deployment and configuration management tasks and many tools use this approach [14, 24]. The model the thesis uses and the used tools are declarative, therefore they are looked at in more detail. In the systematic review from Wurster et al. [29] the declarative tools were further separated into three different categories:

- General Purpose
- Provider-Specific
- Platform-Specific

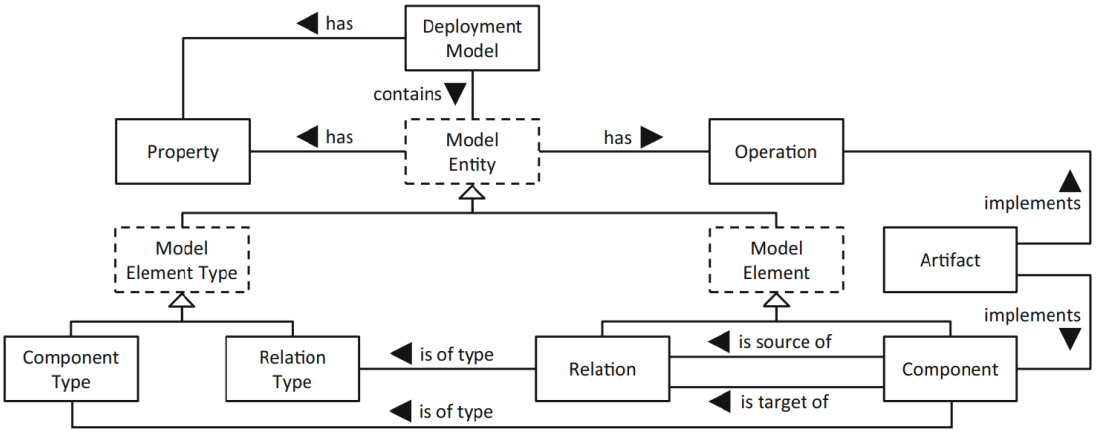

Figure 2.1: The Essential Deployment Metamodel by Wurster et al. [29]

The first category encompasses all tools, which are not restricted to a specific cloud or platform. They are universally usable and contain modules or plugins for many different platforms. As an example, Ansible contains plugins that enable connections to target hosts with ssh, docker, kubectl and many more[1]. Other tools in this category have similar options that allow them to be operated in various environments. Some examples are Terraform, SaltStack and Juju [29].

Provider-Specific tools on the other hand are bound to a specific provider. They are often offered by cloud providers. Examples for this category are AWS CloudFormation and the Azure Resource Manager. These tools are not intended to be used with other providers or environments. As an advantage, they will generally support all features provided by this specific provider, where more general tools may lack behind and not support everything.

Platform specific tools have a different trade-off. They may require a specific runtime environment and are not general purpose or extendable for other ones. Additionally, the components or applications that are deployed, need to conform to the specific format. For example, for Docker Compose and Kubernetes only applications in a container format are deployable. However, as an advantage they are not necessarily bound to a specific cloud provider.

The different categories exemplify various features and shortcomings these tools have. Therefore, often more than one tool is used to solve all the faced challenges. In turn this increases the necessity to have a way to express the deployment in a more comprehensive and tool agnostic way, one of them being the Essential Deployment Metamodel.

2.2 Essential Deployment Metamodel

The essential deployment model selected the essential concepts that are supported by 13 different declarative deployment tools. It can describe the structure of a complex deployment with all components, configurations and their relations.

[1] https://docs.ansible.com/ansible/latest/plugins/connection.html

The most basic entities are **components**. They represent a logical or physical part of the application. Examples are virtual machines or software like MySQL and Tomcat that are installed on a machine or part of a container image.

These components have a **component type**. A type gives semantic meaning to a component. As an example *mysql_dbms* could be defined as a base type for MySQL databases, which conveys some meaning that other components and the framework can expect this component to behave like a MySQL database. Additionally, properties, artifacts and so on can be inherited through the type system.

Components interact with other ones and are not autonomous. The model expresses these connections between components as **relations**. To give them some semantic meaning they have types as well, consequently called **relation types**. Relations are not restricted to connections over the internet. The installation of software on a compute instance is another scenario where a relation is shared. Some examples of definitions of types that will be used extensively throughout the thesis are hosted_on and connects_to. Hosted_on is used to describe a relation where one component is installed on top of another one. For example, a java web application requires Tomcat to be installed first on the machine. Then the hosted_on relation guarantees that tomcat is installed first. Connects_to represents a classical connection between to services. For example, an application needs a connection to a database, or service A needs to access an API offered by service B.

Besides connections for interactions, there needs to be a way to describe the desired state components should be in. This is represented by **properties**. For example, properties can describe a specific operating system(OS) or a database port that is wanted. Similarly, **operations** are imperative commands which are executed to get a machine to a targeted state. A bash script to install software or to start an application is an example.

At last, **artifacts** are files other operations need. For example, the previous mentioned install script is an artifact in the form of a compiled binary that is then started with an operation.

With all these together essential deployment models can be described. An example for such a model is shown in the next section.

2.3 Motivating Scenario

To further demonstrate the motivation for this work, a simple scenario is presented and a basic explanation, how the EDMM can be used, is given. The deployment will consist of the Petclinic[2] that represents a typical web application. The application can show information about pets from a connected database. For it to work correctly a database is necessary and a connection between them has to be configured correctly. It is easily imaginable that further restrictions are given. The application is hosted on a public cloud and the database is hosted on private machines. Additionally, in the cloud environment everything is deployed with Kubernetes and the use of other tools is discouraged or not compatible. The other environment uses Terraform for provisioning of virtual machines and Ansible is used to configure them. These restrictions clearly increase the complexity

[2] https://projects.spring.io/spring-petclinic/

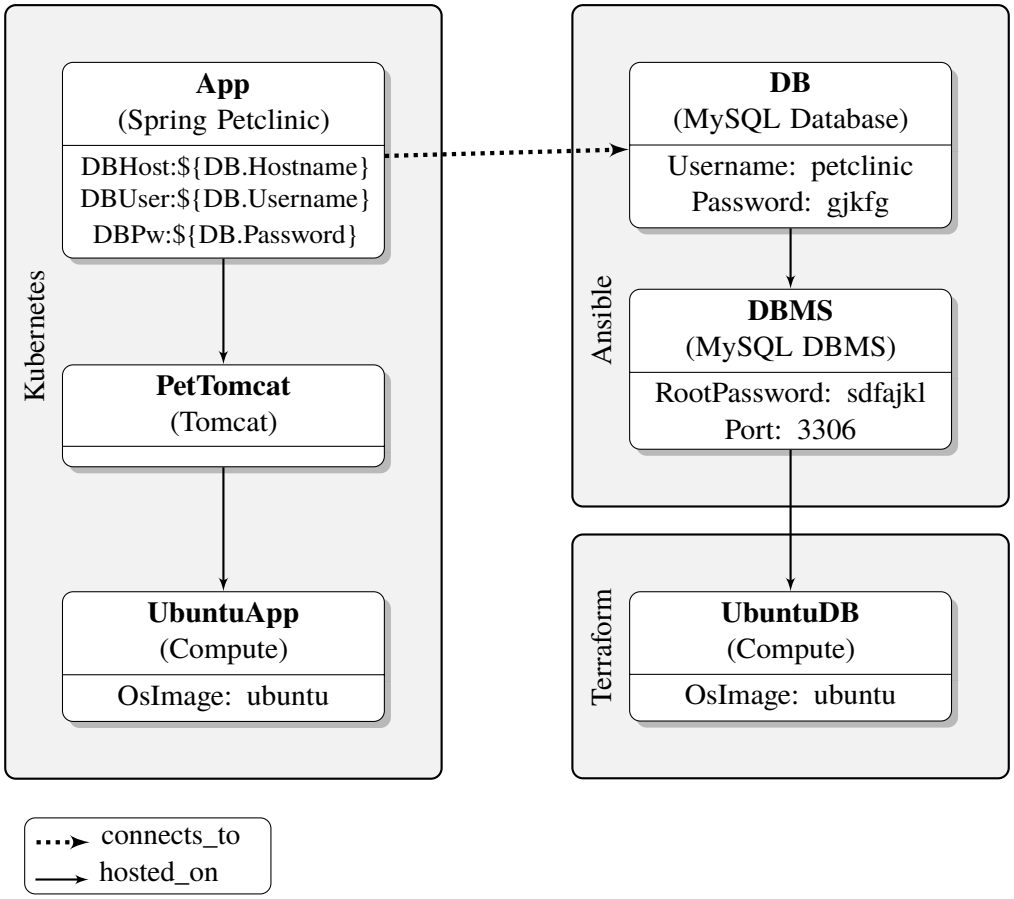

Figure 2.2: Motivating scenario with 3 different technologies

of the deployment because the infrastructure code is written with different tools that have to interact in some kind. Furthermore, there is no easy way to manage the complete deployment. The thesis will introduce a framework to solve the encountered problems.

Figure 2.2 illustrates a model of the described scenario. To create understandable models some of the modeling concepts introduced in 2.2 are used. Therefore, all logical parts are modeled as components that have relations to each other. Regarding the stack for the *Petclinic*, as a base services an Ubuntu instance, called *UbuntuApp*. The type compute just indicates a basis, mostly with an installed operating, system, where other software can be installed on top. It does does not impose restrictions on the way it is provided. Some possible instantiations could be a real physical computer, a virtual machine, or just a pod on a platform, where the underlying machines are abstracted away. Because Kubernetes was chosen, the latter is the case. The next level on this side is a tomcat webserver (*PetTomcat*) which is necessary to serve the application. The arrow depicts the relation between these components. In this case the tomcat web-server needs to be installed respectively hosted on the Ubuntu image, so it's a HostedOn relation. Hosted on relations represent a concept of tightly coupled components, that are part of the same machine or image. The real business logic of this stack, the *App* component, is defined at the top. This is again connected with a *hosted_on*

relation. For all three components Kubernetes is the target technology. Because Kubernetes is used all three components will be baked into one image because of the connection between them. This image is then deployed to the Kubernetes platform.

However, to show and store useful information about pets, the application needs a database. The database uses Ubuntu as operating system as well. But in this case the target technology is Terraform, which implicates possible different platforms as well. E.g. a virtual machine from a cloud provider. Hosted on this, *DBMS* represents the database management system, secured with a root password and a chosen port. They are modeled as properties from the EDMM. The last part is loading the schema and data for the Petclinic. Furthermore, setting the username and password. That these three components belong to each other, is again shown by the *hosted_on* relation.

Now both tech stacks exist but without a connection between them, so the functionality is not given. Therefore, the components require a connection, represented with the *connects_to* relation and the correct configuration has to be set for the App. For a working deployment the hostname, username and password are needed, again depicted as properties. Because they are depending on properties from another component stack they are marked explicitly with a special syntax ${<component>.<property>}. As an additional of this scenario, the hostname is only known after the virtual machine is started, but is needed for the Ansible and Kubernetes group, to install software or establish a connection.

The deployment of this scenario illustrates main challenges that are faced and will serve as an example to clarify concepts in the following chapters. A way to interact and share properties between Kubernetes, Ansible and Terraform is necessary. As a first step, some related work is looked at and examined why they don't solve all the problems from this scenario already, respectively a multi tool deployment.

3 Related Work

The automation of deployments in a multi cloud setup or the prevention of a vendor lock-in are recognized as problems that attract a lot of attention. Most of the time the suggested solution is roughly separated into a modeling step where a tool and provider agnostic model is defined. This is then transformed to a technology specific model. Afterwards, an execution phase or runtime is responsible for the actual deployment.

In the work from Bergmayr et al.[5] a systematic review of cloud modeling languages was conducted. One of the mentioned approaches that contain a runtime as well is CloudML [10]. The goal is explicitly to create a way to manage multi cloud systems. For this goal a Cloud Provider Independent Model (CPIM) needs to be defined by the developer. This is then semi-automatically, some input is needed, converted to a Cloud Provider Specific Model (CPSM). At the end of the transformation this is converted to input for selected frameworks and libraries that provide services to deploy on multiple clouds with reduced complexity which abstract already from the cloud provider to some degree, for example Cloud Foundry. The focus is on the modelling and the goal of supporting different deployment technologies is not a specified goal.

Another relevant work is the OASIS TOSCA standard [19]. It provides a standardized format that describes the components, their relationships and management functionality. This standard format allows the portability and reuse of components. In combination with a runtime as in [6] the same model can be reused for different cloud providers. However, this standardized exchange and deployment only works with a runtime that understands the format. So, the widely used deployment tools are not available. As a special note, Wettinger et al. [26] investigated how to transform artifacts from different tools, for example a chef cookbook, to a standardized Topology and Orchestration Specification for Cloud Applications (TOSCA) artifact, which can then be deployed with the OpenTOSCA runtime [6]. This covers the other way around. Instead of allowing the deployment with multiple tools and runtimes, implementations or models from varying tools can be imported to this specific runtime.

A technology from a more practical background is Bosh [11]. Bosh is a tool collection that unifies the interfaces to different cloud providers and abstracts over virtual machines and containers. However, for the installation several components are needed that cannot be replaced with different deployment tools.

A work that is closer to the concepts presented in this thesis is the approach from Di Cosmo et al. [8]. As a first step the user must define some constraints and traits the deployment needs to have, in their example a Wordpress backend that in turn needs a MySQL database. Afterwards, this representation is converted to an abstract representation of the target system with a tool called Zephyrus. Followed by a second tool called Armonic, which manages the concrete deployment on a platform. For this purpose it needs to provision VMs configure them and execute these steps in

the correct order. These are problems that need to be solved in this thesis as well. However, they use their own tool Armonic to execute these deployments. This is in contrast to the goal of using standard tools that are already widespread.

In the paper from Alipour et al. [1] the first step is to define a cloud provider independent model as well. Subsequently, it is transformed to a Cloud Provider Specific Model(CPSM). These models are then converted to deployment tool specific scripts, in their case Packer and Ansible. So, they use tools that are already established instead of writing their own runtime, which is the goal of this thesis as well. However, they only explore Packer and Ansible, no further options are explored.

Guillén et al. [12] explore the problem of writing components that are portable between different cloud providers. To achieve this, they present a cloud agnostic framework for Java that handles all the configuration and integration tasks. But this is only intended for applications where the source code is available and at the writing of the paper restricted to Java. The goal of this thesis is to not be opinionated about the type of software or scripts.

Some of the works cover parts of the things this thesis wants to achieve, but often the focus is quite different. Additionally, all the presented approaches don't intend to split a deployment with a mix of different tools.

4 Framework for Deployments with Multiple Technologies

In the following, a concept for a framework is introduced to automate and simplify the deployment with multiple technologies. The end goal is the ability to design a model as input that is as technology and cloud provider agnostic as possible. Components in the model can be assigned different technologies. Afterwards, this model is validated, processed and then transformed to technology specific models. The last step is an automatic deployment under utilization of the respectively chosen technologies.

This approach allows to take advantage of the features from these technologies that are proven to work. No own complex logic needs to be written for this, as was chosen by other works examined in Chapter 3. At the same the framework reduces the manual labor or the writing of error prone scripts to coordinate between the tool boundaries. The motivating scenario, introduced in Section 2.3, will serve a reoccurring example to clarify the decisions.

To get to the desired end state, the problem is divided into smaller sub-problems illustrated in 4.1. As an entry-point to the framework in 4.1 a meta-model is defined that allows the definition of technology agnostic models based on the EDMM by Wurster et al. [29]. This serves as the basis for the following steps. Next in 4.2 areas are determined in the model that use the same technologies and fulfill some additional restrictions. In 4.3 these partial models are transformed to deployment technology-specific models (DTSMs). Additionally, based on the areas a workflow is generated in 4.4. Lastly, in Section 4.5 a concept is described to deploy everything automatically.

4.1 Define a Technology Agnostic Model

To allow a split and change of technology, the deployment needs to be described in a way that is independent of the used tool. As already discussed and seen by the related work, a declarative model is often chosen. The focus lies on the description of components, their configuration and dependencies, without describing every step in detail on how to achieve this end state. With the Essential Deployment Metamodel(EDMM), described in Section 2.2, a suitable foundation is already defined. The EDMM, introduced in Section 2.2, extracts the essentials of 13 deployment

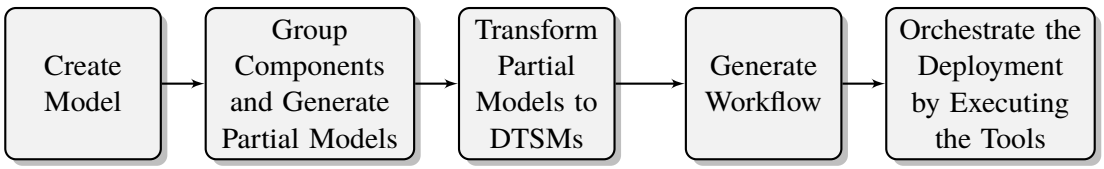

Figure 4.1: Illustrates the steps the framework

technologies which allows the mapping of the defined technologies to the more generic concepts. It enables the developers to express their deployment in a technology agnostic way. Additionally, it is easier to get a holistic overview and the model needs less changes when a technology is swapped because not every every step is listed in great detail.

One concrete implementation uses YAML Ain't Markup Language (YAML) files to express all the concepts from the EDMM which will be used by the prototype. How this can be used and what the entities of the model map to is defined by the EDMM in YAML Specification[1]. Some examples of commonly used types are defined there as well. Other implementations like graphical tools are imaginable as well, similar to Winery[2] for TOSCA. The motivating scenario uses some of the concepts as well in a more loose manner.

In general, all the ideas introduced by the EDMM can be use for the framework. A model consists of components, which have properties, operations and artifacts. Component types are used for declaring supported/default properties and to give semantic meaning to a component. For example, a component with type compute will have to be treated differently than a software component. Some examples for meaningful properties are given in the motivating scenario. The desired operating system, default usernames, ports and so on are possible properties. The interactions between components are, as defined, relations with their corresponding type. Especially, hostedOn and connectsTo will be used to represent the most common patterns, however the framework is not restricted to them and further ones can be added. Apart from these, some new concepts have to be added for a deployment with multiple technologies.

Before these concepts are introduced, a more formal definition for a model follows as the tuple:

$$m = \langle C_m, R_m, CT_m, RT_m, \mathsf{type}_m \rangle$$

The elements of the tuple are defined as follows:

- C_m: Set of components in m, where m is the model.
- $R_m \subseteq C_m \times C_m$ represents the set of relations, where each $(c_s, c_t) \in R_m$ represents a relation with the source c_s and target c_t.
- CT_m: Set of component types, where each $ct_i \in RT_m$ represents the semantic meaning for a component with this type
- RT_m: Set of relation types in m, where each $rt_i \in RT_m$ describes the semantic meaning for a relation with this type
- type_m: Mapping that assigns each *Model Element* in m to its *Model Element Type* Letting $ME_m := C_m \cup R_m$ be the union set of all *Model Elements* of m, and $MET_m := CT_m \cup RT_m$ be the union set of all *Model Element Types* of m, type_m is defined as: $\mathsf{type}_m : ME_m \rightarrow MET_m$

For the goal of describing a deployment with multiple technologies, a way to assign one tech to each component is necessary. This is straight forward to add. Graphically, it's enough to just draw a box around components with their desired technology, as shown in the motivating scenario. In the YAML format used in the prototype, this was achieved with an additional list that contains the

[1] https://github.com/UST-EDMM/spec-yaml
[2] https://projects.eclipse.org/projects/soa.winery

components and their corresponding technology. In general, a method to label a component has to be added and every component needs exactly one label, where the label represents a specific technology. This is defined as follows:

- T_m: Set of all supported technologies, where each $t_i \in T_m$ is a deployment technology.
- $tech_m$: Mapping, which assigns each component to a deployment technology $t_j \in T_m : tech_m : C_m \rightarrow T_m$

However, the notion of groups or components that belong together is lost through this notation. Most of the time, more components belong together and not every component uses a different technology. In the motivating scenario as well the expectation is that the stack on the left side results in one Kubernetes deployment and not a separate one for every component. They logically belong together. A solution for this is to find all groups of components with the same technology programmatically. This approach is presented in Section 4.2 and has the advantage that the user does not need to think about finding the valid optimum.

An additional challenge represent dependencies between components in the form of properties. Looking at the motivating scenario, the component *DB* needs the root-password from the database management system and the port number to access it. These properties could be provided as environment variables, which are then used to establish a connection to the database. As a design decision, properties from components that are connected with an hostedOn connection transfer all their properties implicitly and no special mention in the model is necessary. The framework takes care of it. So, in the database case all the attributes from the database management system are made available to the overlying database as well. Because they share a hostedOn relation, they are on the same machine anyway.

For other relation types an explicit reference is needed. For example, in the reference scenario the *Petclinic* needs access to the username, password and hostname from the database related by a "connectsTo" connection. The assumption here is that different services don't need to know everything about each other. There is no reason for the *Petclinic* to have access to the root database password. With ${} a special notation is used within this thesis to describe these references. The exact notation is only an implementation detail and could of course look different as well.

This approach tries to find a balance between explicitness and the ease of use. Components in the same machine or container, implied by hostedOn, are pretty tightly coupled already. A connectsTo relation implies a clearer separation, so a clearer separation and visibility in the model is sensible. Other relation types can be easily added to one of the two categories.

A special case of properties not discussed so far represent runtime variables. Some properties can't be known in advance and are only determined after the deployment of the component is completed. Examples of these variables are things like a hostname or ip address which is only known after the deployment process for this component is finished. Others could be dynamically generated secrets or certificates to encrypt the connection between applications. In a production environment these should not be hardcoded in the model.

Looking at the motivating scenario, *UbuntuDB* exports a property hostname with the ip address. This attribute becomes only known after Terraform has finished execution but it is needed by *DBMS* and *DB*. Without knowing the hostname Ansible can't establish a ssh-connection to install these components.

In the case that every component is deployed with the same technology, this can be handled inside the specific tool. As one example, Terraform allows to reference output variables from one component as input to another one, other tools have similar capabilities. When the component, that needs the runtime property is deployed with a different technology, this is not possible anymore and another solution needs to be found.

One solution to this problem could be to write all properties into the model with a reference syntax to resolve them explicitly at runtime. These references are then only resolved at when the component is deployed. But this would reduce the independence of the model from the underlying technology, as not all of these properties are technology independent. In the motivating scenario the components deployed with Ansible need access to the hostname of *UbuntuDB* to deploy components on top of it. But under the assumption that all three components are deployed with the same technology, this reference is not is not necessary or would not be sensible at all. For example, with Kubernetes one application consisting of multiple components is merged to one image anyway. So a hostname can't exist before all the components are deployed. So, in the scenario *UbuntuApp* represents only a base image for a pod and not an existing virtual machine.

To stay more technology agnostic these properties are not represented in the model when components are related by hostedOn. They are instead handled by the framework and technology specific implementations. They must infer that an attribute is not known yet and what is needed through the component and relation types. One necessity to make this viable is that after the instantiation of the group the necessary properties are exported which is discussed in more detail in Section 4.3 and Section 4.5. For connectsTo and in general less tightly coupled relations the reference syntax has to be taken again.

In summary, the modeling follows mostly the Essential Deployment Metamodel. One exception is that a labeling mechanism is introduced to assign a technology to each component. Furthermore, properties from underlying components are propagated automatically without the necessity to mention it in the model and will be automatically detected by the framework, for hostedOn. On the other hand, connectsTo and other relations don't propagate automatically. When an attribute is needed it has to be specified explicitly in some way. Properties which will only be known at runtime need no special treatment and will be handled by the framework.

4.2 Group Components to Technology Areas and Generate Partial Models

In the model Section 4.1 it was determined to assign every component a technology label. However, with this approach every component would need to be completely independent from the other ones. Intuitively and visually it you would think that the three Kubernetes components in the motivating scenario belong to one group and the two Ansible components build one group as well. Depending on the underlying technology, this separation might not even be viable at all. For example, with a platform specific deployment tool like Kubernetes or Docker Compose the optimal result is one image that has all the components, connected with a hostedOn relation, baked in. If now all components are regarded as an independent entity, this complicates the transformation and viewing them as one group is the more sensible approach. The grouping has advantages in a more general case as well. When more components are in one group they can be condensed to one technology specific model, described later in Section 4.3. Additionally, features from the well-tested technology can be used to transfer information or optimize the deployment.

To get these advantages without compromising the model process, an automated grouping needs to be found. Furthermore, it is necessary to generate partial models for the groups to allow the transformation. Many cases where a grouping makes sense are rather obvious but others are more subtle and it is not obvious what can be deployed together and what would make the deployment impossible. A similar problem was encountered in the paper"Method, formalization, and algorithms to split topology models for distributed cloud application deployments" by Saatkamp et al. [23]. Here, the relevant goal was to split the model into smaller models for each specific cloud provider.

In the following, first, the challenges are identified, based on examples, and then an algorithm is presented. The goal is to group all components optimally that use the same technology and still allow a valid deployment.

4.2.1 Create a dependency graph from the model

As a first step, the model is transformed to a different representation that is more suited for grouping and workflow purposes. The relevant part in this view is mostly components and relations between them. Properties, artifacts and so on have no influence on the the relations between components and in the case that a component needs the data from another one, then this must be explicitly declared with a relation. When no relation exists, the components can't have any direct dependencies.

Relations are especially important for the order of deployment. For example, when the component *DBMS* is hosted on a compute instance they share a dependency and the base component must exist first. In the graph this is represented by an edge from *UbuntuDB* to*DBMS*. To simplify the model a bit the same assumption is made for connectsTo relations. When *App* connects to the database, the database has to exist first. In a general view the components can be seen as nodes and the relations as edges that indicate the dependency between components. With this simplified model, there is no need to differentiate between the relation types.

More formal as already mentioned before the relations in R_m imply which component needs to be deployed first. To get the correct order all edges are reversed R_{mrev}. Together with the component set this builds an acyclic directed graph $G = (C_m, R_{mrev})$. This graph will serve as a basis to find all technology groups, which do not impose an impossible deployment. Figure 4.2 shows the dependency graph for the motivating scenario.

The graph must be acyclic, otherwise no viable order can be found. As the simplest example, component A depends on component B and component B depends on component A. Now, every component needs the other to exist first and there is no way to resolve this. The deployment is not valid anymore. In general, anything that would create a cycle is not possible, because then the order cannot be defined anymore. This is the main restriction when grouping components.

The basic idea is now to merge components step by step as long as they use the same technology and without introducing a cycle. When two components are merged to a group their inner relations are not relevant for the overall graph anymore and can be removed. They will only be needed for the transformation step explained later. Relations to the outside on the other hand are important, but it's just important that it originates or targets the group. The exact component is not important here.

Some examples are shown next that clarify what can be grouped and what is not valid.

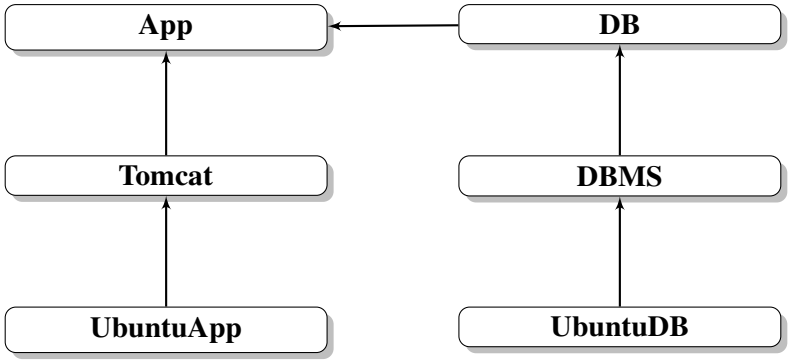

Figure 4.2: Transformation of the motivating scenario model to a dependency graph

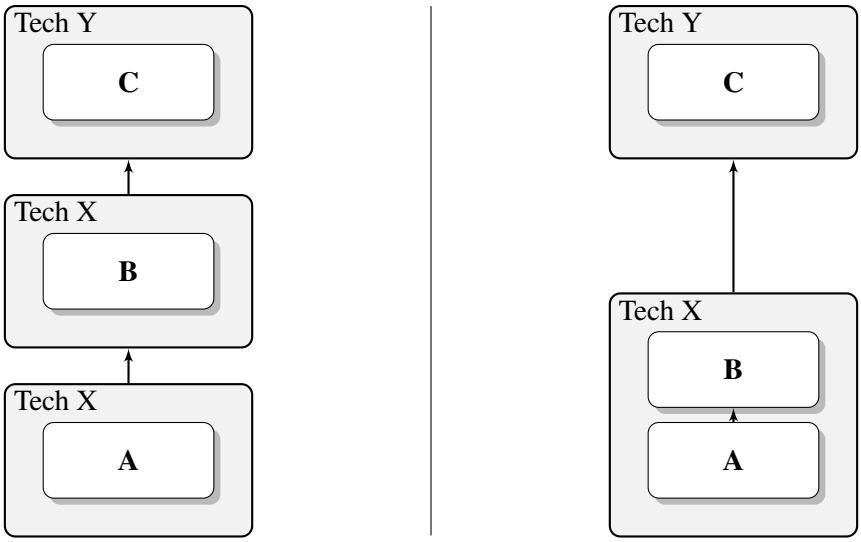

Figure 4.3: Example of two components with the same technology that are groupable.

4.2.2 Examples

As a first example 4.3 depicts a simple case where two components use the same technology and component *B* depends on *A*. Because no other indirect dependencies between the two and other components exist, they can be grouped. No cycle can be created this way and these components could now be treated as one in the next iteration. In the motivating scenario this case is represented on the left side with Kubernetes. There is only one incoming edge from the three components which cannot result in a cycle and it is a valid grouping. Similarly, the part deployed with Ansible can be regarded as this. Another example is the next case in 4.4 where no dependencies exist between *A* and *B* while they use the same technology. Both have only an incoming dependency from *C*, which cannot lead to a cycle as is depicted visually as well. So, it is safe to merge them together. A possible scenario for this is the deployment of multiple virtual machines that have no direct relation to each other, but it would be nice to combine them all to one group. This way the specific technology can optimize the deployment, e.g. make it parallel, and it is not necessary to invoke the tool again and again.

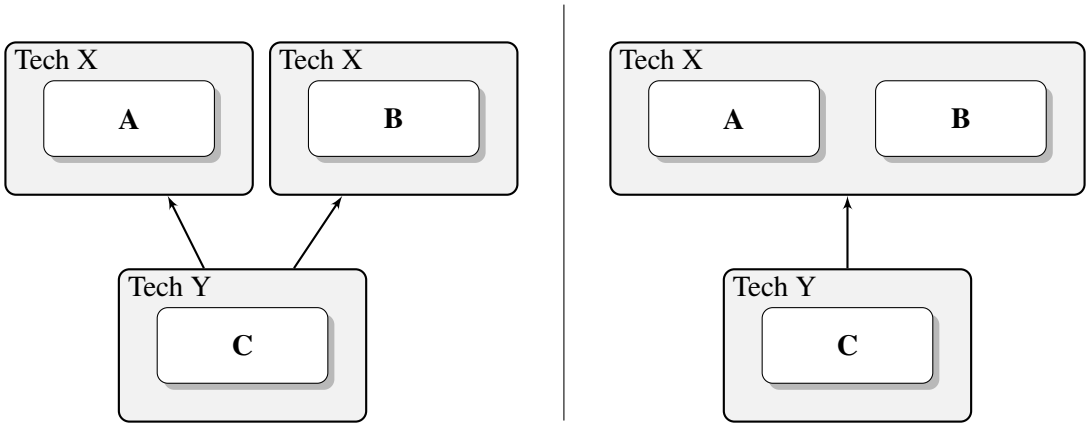

Figure 4.4: Group together when no relation exists between them

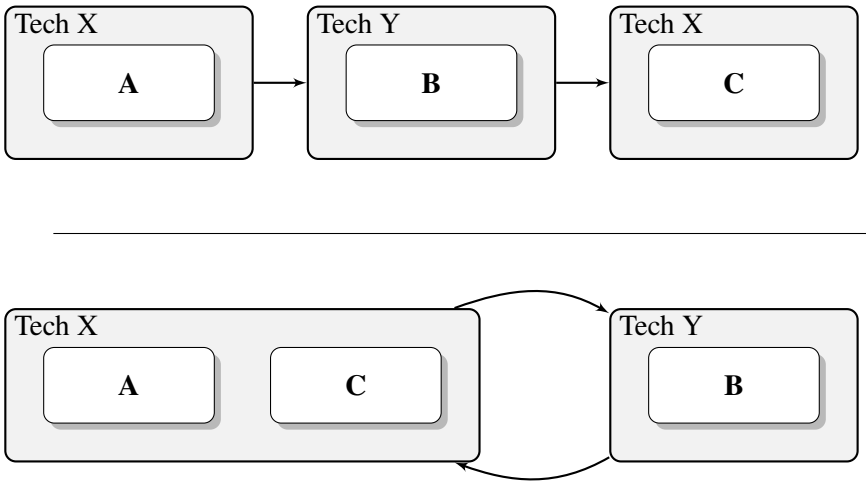

Figure 4.5: Example where a cycle would be created by merging A and C

The next example case in 4.5 contains two components from the same technology and one component that needs to use another technology. So, component *A* has a transitive dependency to *C* over a component with another technology. If the component in the middle would use the same technology as well, this all three could be grouped. However, because of the other technology this is not as straight forward. In the case of a merge of component *A* with *B* the result is shown to the right. A cycle is created that would prevent a deployment order, so the algorithm must prevent this case from happening.

The last example in Figure 4.6 contains all the cases from the previous ones and illustrates a more complex scenario. The basic idea is to deploy 2 virtual machines with cloud formation and install an application on one. On the other one a database is installed with Ansible, similar to the motivating scenario. Additionally, the App is accessed by another Service deployed with Cloudformation.

Regarding the grouping now, *VM1* and *VM2* have no direct or indirect relation. This is the same as the second example and mergeable without introducing a cycle. The components *DB* and *DBMS* represent a similar case to the first example. Only a direct dependency exists, but no other indirect ones. Consequently, they are merged as well. A component that cannot be added is the *Service*. An

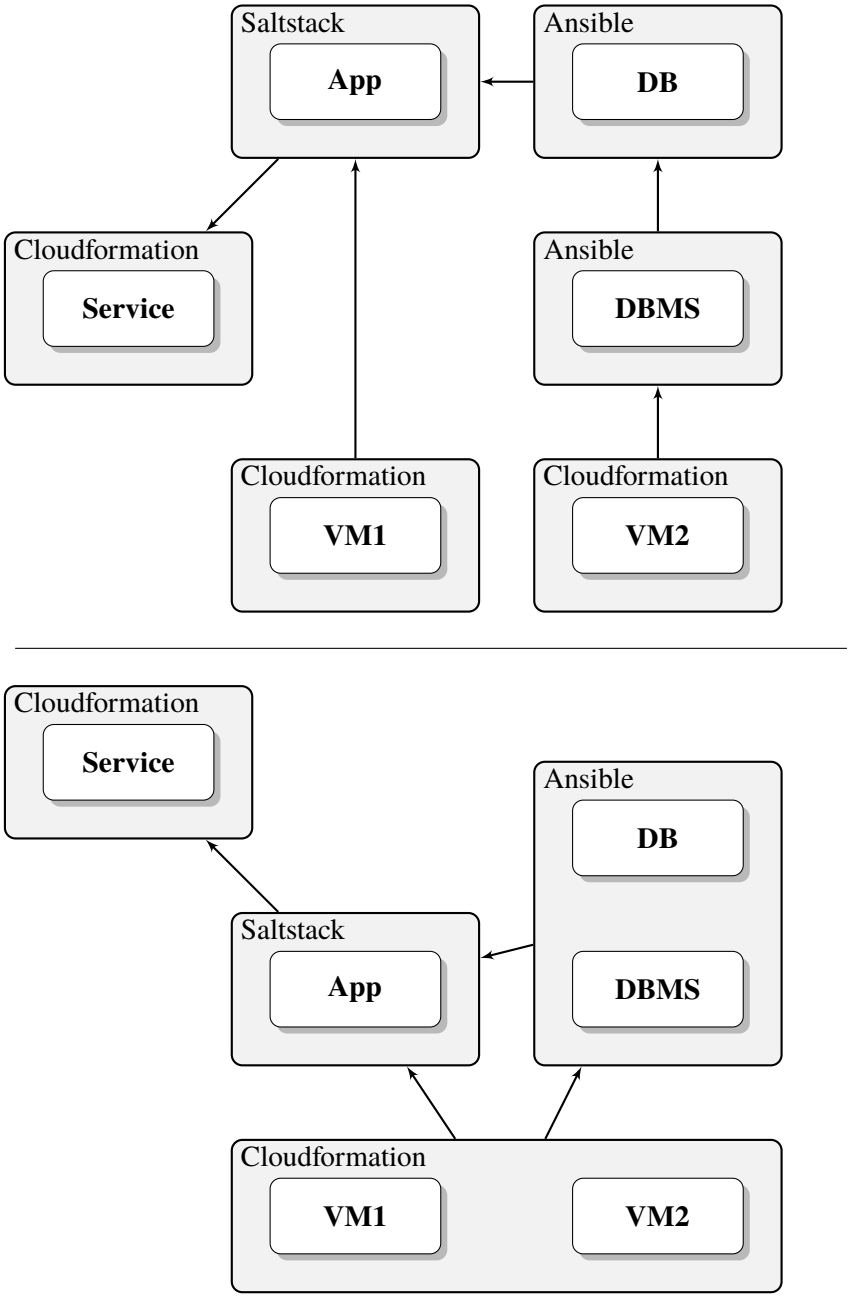

Figure 4.6: More complex example for the grouping algorithm, where all three cases from before exist.

indirect dependency exists between the *Service* and *VM1*. Assuming both were in the same group, they both are deployed at the same time. But *App* is installed on *VM1*, so *VM1* must exist first. *Service* on the other hand needs the application to exist first. There is clearly a cycle, as depicted in Figure 4.5, and no valid order can be found. The motivating scenario is rather straight forward. There is no way to introduce a cycle, so all components with the same technology can be merged.

In summary, a group should contain as many components as possible that can be deployed together and no cycle is created. This translates to all the cases, where only a direct or no dependency between two components of the same group exist.

4.2.3 Algorithm to determine technology areas

With the basic idea cleared up, next, an algorithm is defined to solve the problem in general.

The algorithm is derived from Saatkamp et al. [23]. The merging of groups from the same provider poses a similar challenge, as merging components with the same technology. The problem is similar, therefore the following algorithm shares many similarities. The notation is kept similar where it is sensible.

The following functions will be used in the algorithm:

- $transitiveClosure(G)$: computes the transitive closure of the graph G. This adds edges to all transitively reachable nodes.
- $transitiveReduction(G)$: computes the transitive reduction of graph G. This removes all edges that are only that can be reached transitively as well.
- $mergeGroups(G, g_i, g_j)$: concatenates the nodes represented by group g_i and g_j. This results in creating a new node, all incoming and outgoing edges of g_i and g_j use this node now and all edges between these two nodes are removed. The function does nothing if the resulting graph contains a cycle.

The input for the algorithm is the model m as defined in Section 4.1. Especially, relevant for this algorithm are the set of components C_m, the set of relations R_m and a mapping to a label for every component $tech_m$. As a first step, the algorithm initializes a group for each component and assigns them a label(lines 3-7). Next, all relations are added between the groups(lines 9-12). These steps establish the basic graph that will be used for the merging of groups. Afterwards, all transitive dependencies are removed, because they could result in a cycle as shown by Figure 4.5(line 13). This reduced graph is then used as a basis to merge all directly connected groups, as long as no cycle is induced(lines 15-17). The simplest case is shown in Figure 4.3. After these are merged, the next step is to merge all unrelated groups, where no direct relations exist, similar to the second example depicted in Figure 4.4. For every group of a technology the transitive closure is built and other groups, where the closure contains no edge or only one between them are searched (lines 21-22). If such a group pair is found, they are merged(line 24). As a last step EDMM models are built for each group. This model contains all the group internal relations and the components(lines 29-35).

Algorithm 4.1 DetermineTechnologyAreas(m)

let $gpo := (G_{gpo}, O_{gpo}, tech_{gpo})$
//initialize all groups and assign them a technology
for all $c_i \in C_m$ **do**
    let g_{new}:={c_i}
    g_{new}:=tech(c_i)
    $G_{gpo} := G_{gpo} \cup g_{new}$
end for
//add all reversed edges from the relations
for all $r_i \in R_m | \exists g_i, g_j \in N_{gpo} : \pi_2(r_i) \in g_i \wedge \pi_1(r_i) \in g_j$ **do**
    let $order_{new} := (g_i, g_j)$
    $O_{gpo} := O_{gpo} \cup \{order_{new}\}$
end for
gpo:=TRANSITIVEREDUCTION(gpo_t)
//merge all directly connected groups with the same labels if no cycle emerges
for all $(o_i = (g_s, g_t) \in O_{gpo} | tech_{gpo}(g_s) = tech_{gpo}(g_t))$ **do**
    MERGEGROUPS($gpo_t, \pi_1(order_i), \pi_2(order_i)$)
end for
//merge all independent groups with the same technology if no cycle emerges
for all $t_i \in T$ **do**
    for all $g_k \in G_{gpo} | tech(g_k) = li$ **do**
        let $b_1 := (\nexists r_k \in$ TRANSITIVECLOSURE(gpo): $(\pi_1(r_k)) = g_k \wedge (\pi_1(r_k) = g_z)$
        let $b_2 := (\nexists r_k \in$ TRANSITIVECLOSURE(gpo): $(\pi_1(r_k)) = g_z \wedge (\pi_1(r_k) = g_k)$
        if $\exists g_z \in G_{gpo} : tech(g_z) = t_i \wedge (b_1 \wedge b_2))$ **then**
            $gpo :=$ MERGEGROUPS(gpo_t, gz, gk)
        end if
    end for
end for
// create an EDMM model for each group
let M_g:={}
for all $(g_k \in G_{gpo})$ **do**
    let $m_g := (g_k, \{\}, CT_m, RT_m, type_m)$
    $R_{m_g} := R_m \setminus \{r_i = (c_s, c_t) | c_s \notin C_{m_g} \vee c_t \notin C_{m_g}\}$
    $M_g := M_g \cup m_g$
end for
return (gpo, M_g)

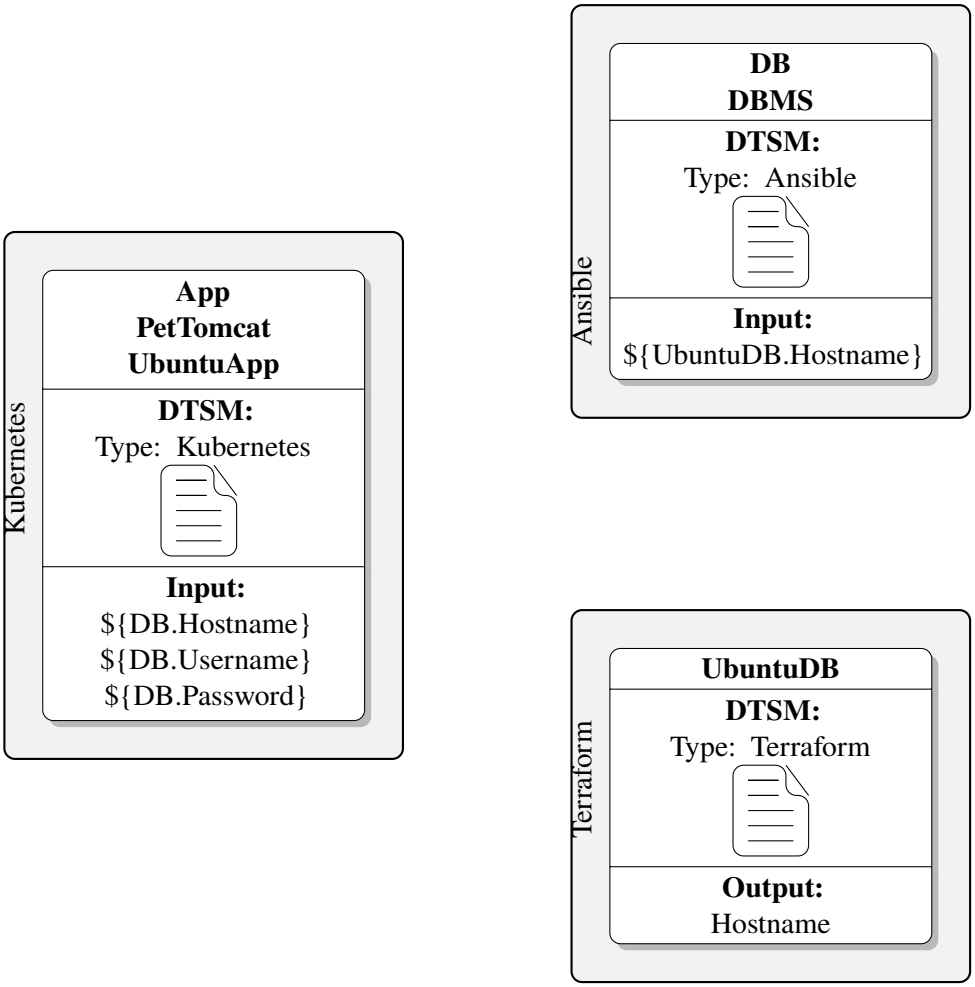

Figure 4.7: Schematic result of the transformation

The result of the algorithm (gpo_m, M_g) is a graph that contains all areas that can be deployed together with the inter-group relations and additionally a sub-model for each group, which contains the group internal relations. This serves as a basis for the workflow generation in Section 4.4 and the creation of technology specific models in Section 4.3.

4.3 Transform Groups to Technology Specific Models

After the grouping and their partial EDMM models are determined, the next step is to transform the agnostic models to deployment technology specific models(DTSMs). The grouping guarantees that components in a group use the same tool and can be deployed together. Therefore, not every component needs to be transformed individually, but on a group basis. Each group gets its own deployment technology specific model(DTSM) based on the partial models generated models. These resulting DTSMs are in in a declarative representation and a domain specific language used by the specific tool. Additionally, the interactions between groups need special attention.

In Section 2.1 declarative deployment technologies were split in different categories. But no matter which category, the basic principles for the deployment are the same. The desired state is written down in a declarative model, which can be represented in a file. However, there is not one language for all of them, but every tool uses its own language. Some examples are, Terraform with its own configuration language[3], Ansible with playbooks in a YAML format[4], Kubernetes with its objects [5], or Cloudformation with its Templates in a JSON format[6]. When these models exist, the actual deployment of these models is fully automatable. For most of the tools it is enough to just invoke a simple command with the files as input. No further manual or hidden steps are necessary.

For the thesis multiple technologies are combined, so the model as defined in Section 4.1 represents the "source of truth". This model is then transformed to these technology specific models. Because the EDMM was extracted from 13 deployment models, every part can be mapped to the assigned technology. However, many degrees of freedom exist and it is the job of the transformation to find a suitable mapping for the defined semantics from the EDMM.

To achieve a successful deployment, the goal of this step is to define these models completely during this phase. Meaning, no further changes and manual steps are necessary for these files in later steps. Additionally, this approach guarantees a clearer separation between the transformation and a later execution. What will be deployed is clearly visible and reproducible. Additionally, only some meta info about them is necessary, apart from that they can be treated like a black box after their creation.

The actual transformation can now be achieved by iterating over all groups with their sub-model as input and transforming them one by one. Additionally, they need to be aware of the complete model for inter-group relations. Depending on the technology a different implementation, later called transformer, is necessary, because of the mentioned technology specific languages.

For a general transformation, assuming only one technology and group, the transformer has to take into account everything from the sub-model m_g. This includes the relevant components and their types. When the type is a compute instance, the treatment will be completely different to a software component. Furthermore, all the information components contain, including artifacts, properties and operations is relevant. For instance, in the motivating scenario the bottom components have a property for the operating system(OS), set to Ubuntu. This must be taken into account by the transformer and mapped to the technology specific description. In the DTSM this could then be just called "id123". As soon as the model consists of more than one component, relations have to be taken into account as well. In contrast to the later defined workflow generation, the relation type and their semantic meaning is relevant as well. A hostedOn relation must be treated completely differently to a connectsTo relation.

For example, in the motivating scenario, the Ansible components rely on the knowledge, that at the bottom of the component stack, some kind of machine exists, it can connect to. For this, the transformer needs to take into account the relation hostedOn and the type of the underlying instance. It mustn't install the software on another instance, which only shares a connectsTo relation. Relations are an important information for other transformation decisions as well. For instance,

[3] https://www.terraform.io/docs/configuration/index.html

[4] https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

[5] https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[6] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/gettingstarted.templatebasics.html

for the platform specific tool Docker Compose, a container image needs to be created with all the software installed. The complete prebaked image is then deployed on the platform. This can only work if the relation between the components is clearly defined with a hostedOn. Otherwise it would make no sense to include them in the image. With all this information combined a technology specific model can be built. The exact transformation and mapping to EDMM entities per technology is not the focus of this work and is better explained more thoroughly in other works.

However, in a deployment with multiple technologies the above mentioned approach alone is not viable anymore. Components in a group can have dependencies to other groups as well. As an example, attributes from am underlying component, like the password for a database needs to be made available to other components. Such attributes are mostly encoded in the form of properties.

The discovery of these properties can only work when the technology specific implementations is aware of the whole EDMM model in addition to the own sub-model. With this information combined it can transform its own group and find all the necessary properties from other groups. Possible sources for properties are:

- explicitly through other relations with the defined notation
- the own properties
- inherited through the type system
- transitively through hosted_on relations

The resolution of properties can be solved similar for all components and is depicted in Algorithm 4.2. As described in the modeling section, only properties from the own stack are collected without explicit references. No reference is resolved in this step. On the contrary, the technologies can use their model knowledge to add additional references. With all this information a transformation to a deployment technology specific model is possible.

One topic that needs special consideration though, is that not all properties are resolvable at transformation time, because they will only get a valid value when the group was deployed. This is again only the case because of the usage of multiple tools. Under utilization of only one tool and group internal this is no problem. It can be handled with all tools in some way. So, it is not possible to build a completely deployment ready model. But this is a specified goal of this step, that the models don't need change anymore. A viable solution is to recognize all references and refer to a file, environment variables or something similar instead. These are only populated in before the model is executed. This has the advantage that in the models no changes are necessary anymore and it is clearly visible by looking at the model what variables are not yet defined. Because the models are technology specific, this needs to be solved in different ways depending on the concrete implementations. In the case of Terraform or Ansible, one solution is to reference a JSON file or environment variables that will exist and contain the necessary attributes when the tool is executed. More detailed information for the specific implementations is given in Chapter 5. The transformation to a deployment technology specific model is explained more there as well.

An abstract representation of the result for this step for the motivating scenario, is given in Figure 4.7. The result consists mostly of a deployment technology specific model per group, which have input parameters and output parameters built in where necessary. One set of file for each, the Terraform group, Ansible group and one for Kubernetes. The actual DTSM for Terraform is then a resource definition for a virtual machine in the Terraform configuration language. Because Terraform

is a general purpose tool and it is not defined in the model explicitly, many different concrete implementations are possible. The model could provide a machine from many different providers. It depends on the implementation's default and given properties. That it is a compute instance and not only installed software can the transformer derive from the component type. All the properties expressed like the operating system must be taken into account as well. In this case this means, that Ubuntu will be chosen as the operating system. The transformer will read all information and then output a Terraform file with the fitting resource definition filled in. Some examples of how a resource definition looks can be found online, e.g. for Openstack[7] or Azure[8]. Additionally, as a specialty in the multi tool environment, information to access the machine has to be added to the DTSM as well. In the case of this group,this manifests in the form of a hostname that is the exported output of the Terraform resource. The correct attribute for the ip address needs to be found. This can then be referenced in another section, e.g. to define a file output with this attribute as hostname, which in turn can be read during the execution.

The transformer for the Ansible components(DB and DBMS) can derive them as some kind of software component, which needs an underlying machine. The DTSM, here called cookbook and in a YAML format, needs to define a reference to a hostname that will be entered later. This will be used to establish a ssh-connection and then the execution of scripts is added to the cookbook. All non runtime information, for example necessary username, ports, other properties, operations and so on, is extracted from the model and encoded into it.

The last remaining DTSM is the Kubernetes stack. Here, the transformer must recognize that all three components connected with a hostedOn connection belong together. Because Kubernetes is containmer based, these result in one container image, described in a Dockerfile. As with the other technologies all information from the model needs to be taken into account. For instance, that the basis should have the Ubuntu operating system. Additionally, a deployment in Kubernetes is described as a Deployment object[9]. Here is the best place to define some attributes that will only be known at runtime as well and then provided as environment variables or files. Again similar to other technologies, only runtime properties must be represented as references. A so called "ConfigMap" would be an idiomatic choice to store configuration. This could be used to reference the needed hostname from the database instance. Through this separation the actual deployment is defined and no changes are necessary anymore to the DTSMs anymore.

All in all, through the essential deployment model a connection between these tools can be established like this. Additionally, as much as possible from the deployment technology specific models is defined. In summary, the implementation of this step consists of iterating over all groups and collecting all information necessary, with the subsequent transformation to a DTSM. Runtime properties are only marked with a reference to a later initialized file or object.

[7] https://www.terraform.io/docs/providers/openstack/r/compute_instance_v2.html
[8] https://www.terraform.io/docs/providers/azurerm/r/linux_virtual_machine.html
[9] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Algorithm 4.2 $CollectPropertiesForComponent(model, c_i)$

```
1: let props:=
2: props:=props ∪GetProperties(c_i)
3: let parent:=GetParentType(c_i)
4: while parent! = null do
5:     for all prop ∈ parent do
6:         props := props ∪ prop
7:     end for
8:     parent:=GetParentType(parent)
9: end while
10: for all comps ∈ GetHostedOnComponents(C_i) do
11:     props:=props ∪CollectPropertiesForComponent(c_i,model)
12: end for
```

4.4 Generate Workflow

After components are grouped, the next problem is the generation of a workflow which establishes the order of deployment and writes down imperatively what will happen step by step. A workflow is necessary, because there exist dependencies between components implied by the relation types and not all use the same technology. As an example in the motivating scenario, naturally, the *UbuntuDB* needs to be deployed first, before a software or components can be installed on top. The same is true for connects_to relations. The connection between *App* and *DB* can only be established after the database is up. Because they use different technologies, this can't be resolved by the tool. Consequently, a valid order has to be found and the steps for a successful deployment must be written down imperatively.

As a simple rule, already used in the previous section as well, every kind of outgoing relation implies a dependency that needs to be provisioned first. Other entities like artifacts and properties have no influence on the overall order of components, because they are mostly relevant for the transformation of a component to a specific technology. Relevant are the relations between components. This can be modeled with the already introduced reversed edge graph, illustrated for the motivating scenario in Figure 4.2. A similar approach is chosen by Breitenbücher et al. [7] with a Provisioning Order Graph. To simplify the steps, it is assumed that every relation implies a dependency represented by an edge in the reversed order. So, if a software is hosted on a virtual machine, the software depends on the virtual machine represented by the incoming edge. Additionally, the dependency graph may not have any cycles. If a cycle exists, it is no longer a valid deployment. An order could be determined now by looking at all components and their relations.

However, in Section 4.2 a graph gpo_m with a grouping of components and dependencies between them was defined. This grouping can be used to simplify the workflow generation. Instead of looking at every component it is enough to look at the areas. When three components are to be deployed with platform specific technologies like Kubernetes or Docker Compose, they result in the same container anyway. In the general case, when components are merged, there is no other component that needs to be deployed in between. Otherwise, a cycle would be created in the graph. Through this approach no relations are lost that have influence on the overall order. The ones that are not considered exist only from one group member to another one. For example, in the motivating

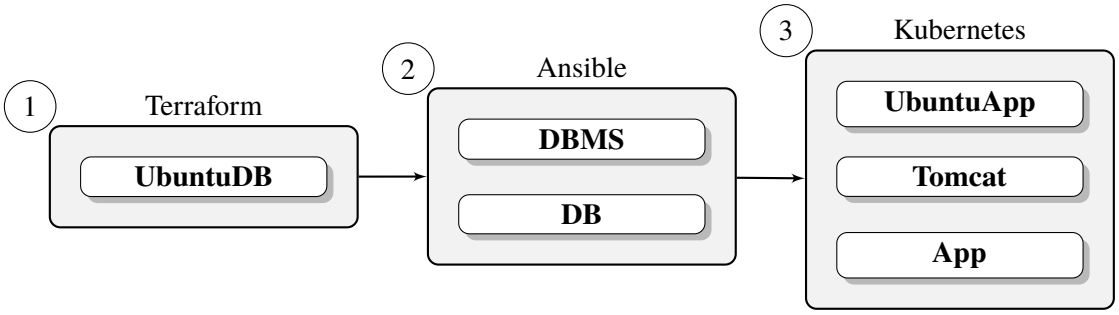

Figure 4.8: Order of deployment for the reference scenario

scenario the database management system needs to be installed before a db schema can be loaded. Dependencies like this can be solved better within the transformation step described in Section 4.3, which allows amongst others things better utilization of the tools. How the ordering is solved inside groups depends on the technology. Some have built in solvers and it's enough to mark the relations with a dependsOn. For others a sorting is necessary as well. Some example are given in 5 and this is solved in Section 4.3. For the workflow generation groups can be regarded as a black box. The internals are not of relevance, only inter group dependencies matter. The workflow now simply consists of assigning a order to groups depending on their relations.

So, for the deployment workflow as a whole it is enough to specify the order in which the groups are deployed. Groups that have no incoming edge anymore can be deployed safely. With this approach an order is established that respects all relations. In a graphical sense groups that have no incoming edges are free to be deployed. When no relations exist between them they can be even deployed in parallel. This sorting is a well known problem. One algorithm that solves it is described in Topological sorting of large networks by Kahn [16]. How the resulting order of steps, the workflow, is represented is not fixed. Different workflow languages or a custom representations are imaginable. This depends on the concrete implementation. Some examples are BPMN [20] and BPEL [3].

Looking at the motivating scenario, the resulting order is displayed in Figure 4.8. The first step is to deploy *UbuntuDB*, then the Ansible group and the last step builds the Kubernetes stack with the Petclinic application. How the internals are handled is not of relevance for this step. The groups are essentially a black box.

Another aspect, that is now possible with the grouping available and the workflow generated, is validation. The easy part is to just check if the graph contains a cycle, then the deployment is clearly not valid. Another reason, that makes validation necessary is, because not every possible combination of technologies is feasible. Additionally, some tools are restricted to a specific platform as categorized in Section 2.1. As an example Docker Compose needs a specific runtime environment. So, it's not possible to start a virtual machine, install Tomcat on it and then select Docker Compose to deploy the application. This combination is not compatible because for Docker Compose the underlying components need to be deployed in a Docker environment. This shows as well why the validation is only possible after the grouping. This can't be decided by looking at every component alone. Instead the group as a whole has to be valid.

In summary, an order is defined in which groups are deployed represented in some way. This is achieved by a topological sorting of the graph generated by Algorithm 4.1.

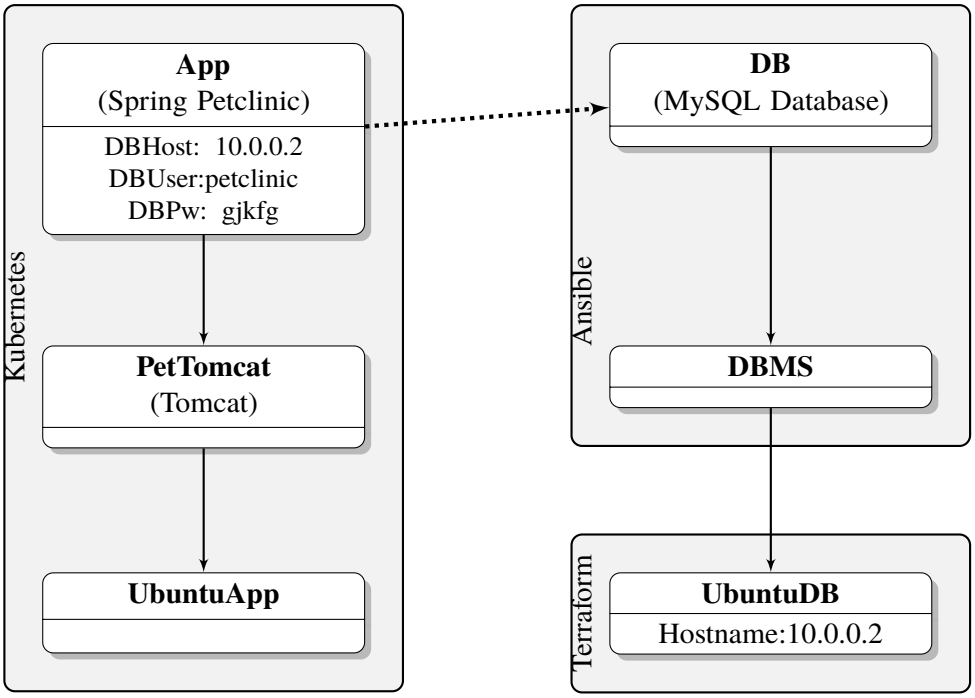

Figure 4.9: Model with all added properties after the deployment

4.5 Deploy and Execute the Technologies

The last phase is the automatic execution of the created deployment technology specific models(DTSMs) and managing the interactions between tools. As mentioned, to deploy new versions as often as possible manual tasks should be prevented where it's feasible. Because all the necessary information is available, it makes sense to automate the process. In Section 4.4 an ordered workflow with groups was created and in Section 4.3 groups were transformed to technology specific models. This serves as the basis for an automatic deployment. The idea behind this step is to have a main coordinator, which deploys groups one after another similar to Section 4.3. But, now the order defined by Section 4.4 is followed. For every technology a specific implementation is necessary again. This implementation is then responsible for calling the tool with the deployment technology specific model(Deployment Technology Specific Model (DTSM)) as input. Every concrete deployment action of a group is then managed by the tool. The workflow gives a clear order of groups that can be deployed after each other. In theory it should now be enough to just call the fitting tool for the groups one after another.

4.5.1 Procedure

Before the deployment technologies can be invoked, some preparations are necessary though. This arises from the interaction between multiple deployment tools as discussed in the transformation step. A DTSM can have input parameters and output variables, arising from runtime properties

needed or provided to other groups. Therefore, a path to share properties between groups has to be established. A simple solution for this problem is the augmentation of the EDMM model. Runtime properties are written into it with the value after a component was deployed.

Then, the steps necessary to deploy a group are as follows:

- resolve and import all necessary attributes
- Execute the specific tool
- Export attributes, that are only known after execution

The first step is resolving and importing runtime attributes that are needed as input parameters for the DTSM to deploy. With an implementation that utilizes the EDMM model to store all runtime information as well it is easily solvable. To resolve them correctly the Algorithm 4.2 can be reused. This delivers all properties a component has access to and then just the ones that are specially marked as computed have to be chosen. Additionally, explicit references, as defined in Section 4.1 have to be resolved now. Afterwards, depending on the technology, they are provided to the DTSM such that they can be parsed. This must be coordinated with the corresponding transformer implementation of the technology. Common ways are environment variables or for better visibility, an exchange format like JSON, YAML or tool specific formats. As explained, the actual implementation could look different depending on the used technology. When all necessary info provided, the tool can be invoked.

Production-ready tools all have some way of controlling them with a command line interface (CLI) or an application programming interface (API). These can either be called manually or more optimally in an automated way. For example, the process could be always triggered when a new change was made. They are often designed with this workload in mind. The declarative nature helps with this as well. The technology will reach the described state completely autonomous. This means, that as mentioned it is mostly enough to invoke one or two commands. Some examples are `terraform apply`, `kubectl apply -f <filename>`, `ansible-playbook <filename>`. In some cases like Salt, some software has to be installed and configured on the target machines before they can be used. But this represents only one additional command and can be handled by the executor. What is written exactly in the DTSM is not of relevance and doesn't need to be parsed by the executor. It can be treated like a blackbox, the tools will take care of it completely.

After the tool is finished, the missing step is to extract outputted information. This is necessary for the next groups, which need information like the *hostname* or a generated secret. Again, as with the import this happens technology specific and must be coordinated with the transformer. Some possible implementations are writing to a file or parsing the console output. As an easy way to distribute the properties, they are just added to the corresponding component. For instance, for a virtual machine component a property *hostname* is added, which can be read by the next step. When this is finished, the next group can be deployed with their specific technology. Starting with the import of needed attributes again.

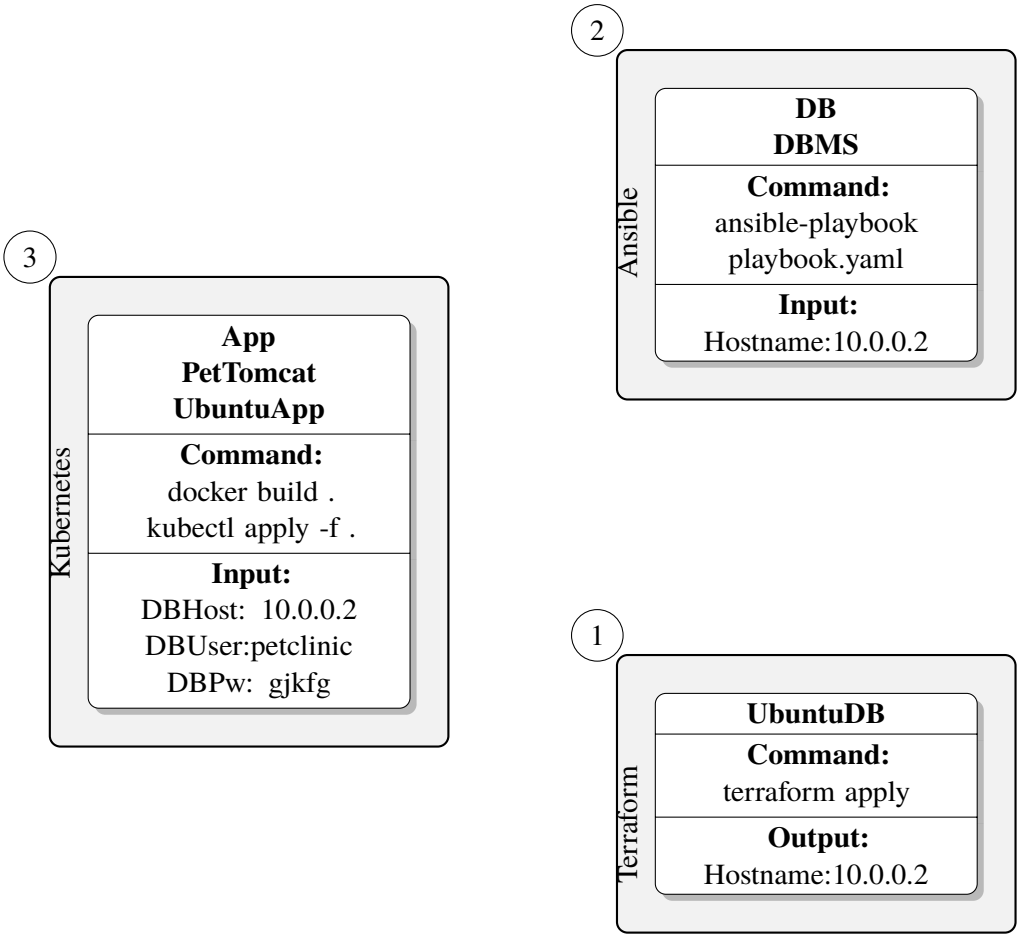

Figure 4.10: Schematic representation of the execution steps

4.5.2 Practical Example

To clarify the concept with a more concrete example Figure 4.10 shows in an abstracted way the relevant parts that will happen. First, as determined by the workflow, the machine for the technology specific implementation for Terraform is invoked to start the Virtual Machine (VM) for the database. The process takes place as described above. It starts with the collecting of unresolved variables. In this case none is needed, all information is available. Next up is executing the DTSM with Terraform. In the case of Terraform this could be simply executing the command `terraform apply` in the folder the DTSM is located. The actual deployment and everything around is handled by Terraform, the executor does not need any specific information. After it has finished executing, the *hostname* is exported in some way e.g. a JSON file depending on how it was defined in the DTSM. Here, the executor is relevant again and needs to take this information and write it back to the model. Thereby, the next groups can have access to it. The added properties are illustrated in Figure 4.9.

After the virtual machine is deployed the software installation can be started. This is executed by an Ansible specific executor. The first step is again to collect all relevant properties from the model. In this case, obviously the *hostname* from the virtual machine is necessary to establish a connection. Because it was added in the previous step, the info is available and can be shaped in a form the DTSM

expects. One possibility is the creation of a JSON file as is Implemented in the prototype. With all input parameters available Ansible can be invoked. As with Terraform and many other possible tools this may translate to simply calling a command. In this case `ansible-playbook playbook.yaml`. The *playbook.yaml* is the chosen name from the transformer of the DTSM. Ansible will independently establish a connection to the machine and install the defined database management system and then load the schema. All the normal defined properties are already encoded and so the database password and so on is set correctly. No runtime info is created here, so an export is not necessary. The last group is the Petclinic application. As always, the first step is resolving all references again. In this case, the hostname, username and password of the database are relevant. This needs to be made available to the deployment group. An idiomatic way in Kubernetes to store the configuration of a container is in a ConfigMap which the DTSM can reference. Because this group is deployed on the Kubernetes platform the treatment is a bit different. As a first step an image needs to be built from the component in addition to the deployment. Here again, the transformer and the executor needs to be coordinated. Commonly Docker to build container images. The necessary Dockerfile is already built by the transformer. The executor just has to invoke the docker build process, which means invoking a command again. Possibly, a further command needs to be invoked to push it to a registry. Eventually, with all the preparation done, the actual deployment can start. This entails the invocation of a CLI tool kubectl with the command `kubectl apply` or something similar. The goal is to send the ConfigMap and the DTSM files to the Kubernetes cluster. There the files are interpreted and the container, respectively the deployment, is transferred to Kubernetes. Because of the ConfigMap, a connection to the database can be established. Again, as with the other steps all the actual work is done by Kubernetes this is only called by the framework. This concludes the execution step.

In summary, this step consist mostly of invoking technology specific implementations for each group. The task of an implementation is largely to manage the export and import of runtime specific properties. The import means to resolve all the references that are contained in the model. Then the invocation of the technology starts, mostly just one or two commands. Afterwards, the last step is the export of generated properties. They are again, depending on the technology, exported or read out in some way and then must be written back to the model.

Algorithm 4.3 $MultiToolDeployAlgorithm(m)$

1: (gpo, M_g):=DetermineTechnologyAreas(m)
2: wf := CreateWorkflow(gpo_m)
3: *//transform all groups to their DTSM representation*
4: **for all** $g_i \in gpo$ **do**
5: **let** $transformer$:= InitTransformer($label(g_i)$)
6: **let** $dtsm_i$:= Transform($transformer, m_{gi}, m$)
7: **end for**
8: *//execute all DTSMs*
9: $execM := m$
10: **for all** $g_i \in wf$ **do**
11: $executor$:= InitExecutor($label(g_i)$)
12: $execM$:= Execute($executor, g_i, execM$)
13: **end for**

Finally, after all the steps are finished, the deployment is done and the targets are in the desired goal state. The complete process is summarized with Algorithm 4.3 which shows the complete framework, where the functions execute the steps described in their respective section. The framework starts with the model as input. With the help of it, technological areas are determined and sub-models are built for each group. Afterwards, the workflow is generated on the basis of this group provisioning order graph. Next, it is iterated over all groups, for each the correct technology is initialized and then used to transform the EDMM model to a DTSM. After the DTSMs are created, the execution of them is next. The iteration uses the order defined by the workflow and initializes the right executor, again technology specific. After every step executed properties are written back to the execution model. This clarifies as well, that only the last two steps depend on a specific technology and need a special implementation. A prototype of this approach is implemented in the next chapter.

5 Prototypical Implementation

To validate the aforementioned concepts a prototype was implemented. The prototype enables the creation of technology agnostic models and allows the automatic deployment with multiple deployment technologies. Some basic functionality was already implemented by the EDMM transformation framework presented in [28]. The basic ideas stay the same. First, the user needs to define a declarative model as defined by Section 4.1 in a YAML format. This file serves then as input to the program together with a desired target technology. After the execution, technology specific models were created that can be deployed with a specified technology. For this thesis the system was extended to support the assigning of differing tools to different components. Additionally, an automatic deployment was added for some technologies.

To validate the framework a model was created that mirrors the motivating scenario introduced in Section 2.3. Appendix A.1 shows the file in the YAML format without the definitions for the relation and component types. The model contains a pet-clinic application with Kubernetes as assigned technology. For the other stack a compute instance with Terraform exists which serves as a base to the Ansible assigned database components. Furthermore, properties are chosen similarly as well.

5.1 Overview

The prototype is written in Java and exposes a command line tool to control it. To get a better understanding, Figure 5.1 shows a simplified class diagram of the application. As explained in the concept, as much work as possible is done technology agnostic and only the transformation to DTSMs and the execution an implementation is unique for the technologies. This architecture allows the easy extension with other technologies, they just need to implement the Transform and Execute interface.

The application is executed with the following command if the desired goal is a deployment with multiple technologies:

```
edmm transform multi <input>
```

Whereby, <*input*> is the file location of the yaml-model that is to be transformed. For a deployment with multiple tools the second word is always multi. Alternatively, the desired target technology can be used, then the assigned technologies in the model are ignored.

After the program is executed a folder, called *multi* exists. The structure of it for the example scenario is shown in Listing 1. It contains a *execution.plan.json* file, where the workflow is written down as described in 4.4. Furthermore, a folder is created for every group containing the transformed models. They contain the DTSMs and the artifacts that will need to be copied. The execution halts after that and asks if an automated execution and deployment is desired as well. Which can be accepted or not

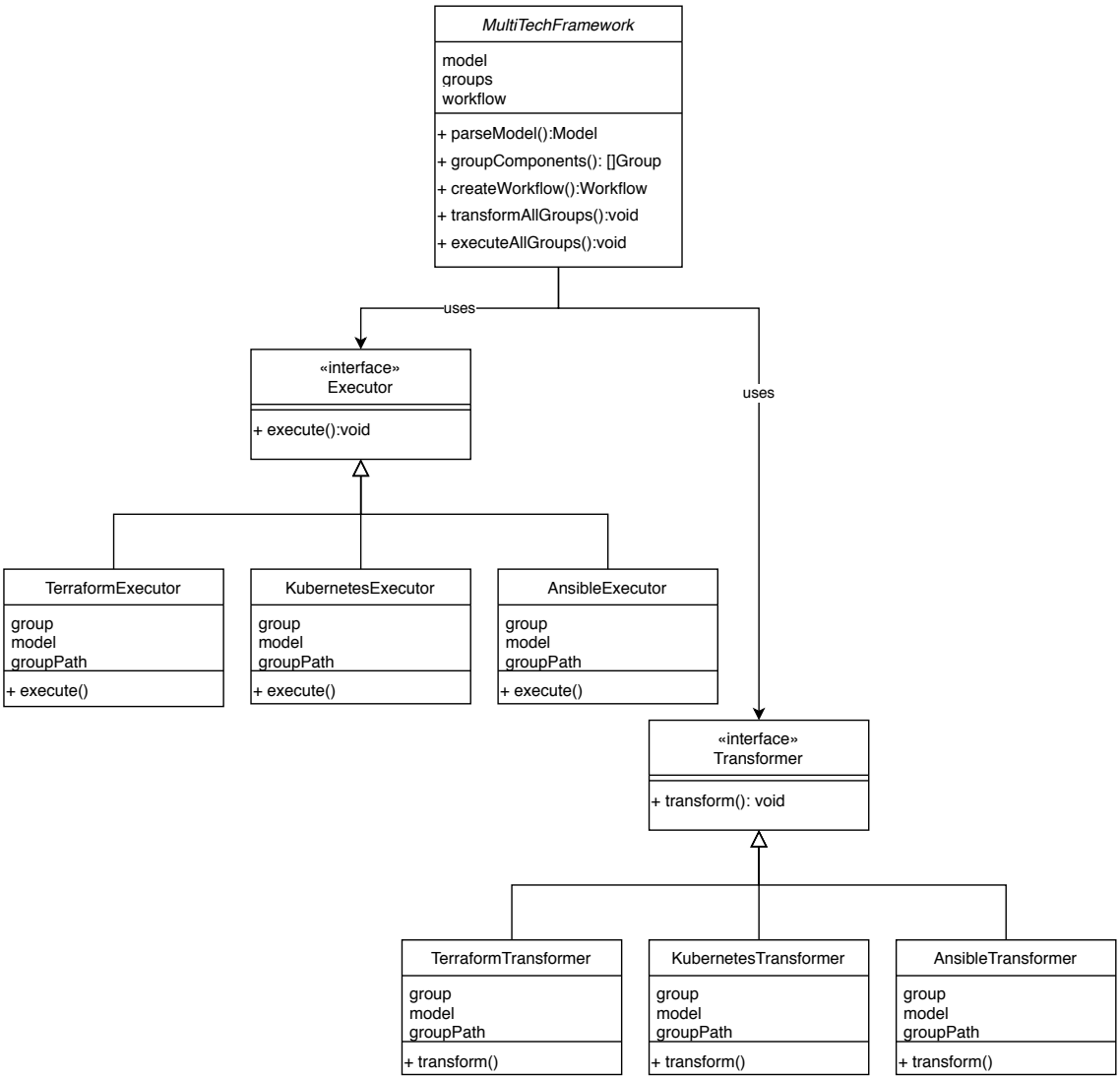

Figure 5.1: Simplified class diagram of the prototype

after the resulting models were inspected. If the command is given, the whole model is deployed with the specified target technologies. For the prototype this contains Kubernetes,Terraform and Ansible.

Listing 1 Basic structure of the transformation result for motivating scenario

```
multi
├── execution.plan.json
├── step0_TERRAFORM
│   ├── files
│   │   └── ...
│   └── compute.tf
├── step1_ANSIBLE
│   ├── files
│   │   └── ...
│   └── deployment.yml
└── step2_KUBERNETES
    ├── files
    │   └── ...
    ├── pet-clinic-deployment.yaml
    └── ...
```

Listing 2 Example for a component

```
pet_clinic:
  type: web_application
  artifacts:
    - war: ./files/petclinic/petclinic.war
  operations:
    configure: ./files/petclinic/configure.sh
  relations:
    - hosted_on: pet_clinic_tomcat
    - connects_to: db
  properties:
    db_hostname: ${db.hostname}
    db_user: ${db.user}
    db_password: ${db.password}
    db_port: ${db.port}
```

5.2 Implementation of the Modeling

Some implementation details are explained in more detail in the following.

5.2.1

For the prototype a representation in yaml was chosen for the essential deployment model which serves as input. It follows the same conventions as in the transformation framework and the ideas and rules explained in Section 4.1. Listing 2 shows how a component is defined in the YAML format. It is a snippet from Appendix A.1 which shows a complete model. Every component is identified by their unique name. As defined in Section 4.1, components can contain different properties, relations, artifacts and operations with their respective syntax. With artifacts and operations a file path is required that will be copied over or executed on the target host. Properties have a name and a value

Listing 3 Example Technology Split

```
orchestration_technology:
  ansible:
    - db
    - dbms
  terraform:
    - ubuntu_db
  kubernetes:
    - ubuntu_app
    - pet_clinic_tomcat
    - pet_clinic
```

that is either a static value or references a value from another component, then the shown syntax with `${}` is used. Only other components that share a connection are allowed. These variables are only resolved during runtime and only a reference in the technology specific model is created. The multi tool approach is realized by adding the capability to assign a technology to components and by the introduction of a syntax to reference properties from other components. In Listing 3 shows a snippet of how every component can be assigned a label with a technology. For every technology there is a list of components that uses it. Every component needs to be listed here and of course the specified tools have to be supported. When the application is started the YAML file is parsed and converted to a Java representation. It utilizes inheritance for the types, properties and so on are attributes on the instantiated object.

5.3 Implementation of Technology Grouping and Workflow Creation

After the initialization of the model, a dependency graph is created that is needed for the grouping and worfklow creation. It follows the definition in Section 4.2. Components are added as nodes and relations are added as edges in reverse. For ease of use, the graphs are presented by the library JGraphT[1]. It already implements fitting data structures and algorithms for graphs. Because it is a generic graph structure, nodes can be represent with component objects and edges with relations. Therefore, all the information stays available. With the needed graph build, the steps from Algorithm 4.1 are followed to find the technology areas. Where the transitive reduction and the transitive closure is supported by the library and just a function call. The last step, the creation of an EDMM for each group can be simplified with JGgraphT as well. With the concept of a subgraph and all components of a group as input, the needed inner-group relations are chosen automatically.

In the next step, the workflow is generated which follows the concept‘s description as well. The graph library contains a built in iterator that will return the nodes in a topologically sorted manner. These steps don’t need a technology specific implementation.

Tool	Import	Export
Terraform	JSON, Environment Variables	JSON, CLI
Kubernetes	Configmap	Cli, Api
Ansible	JSON, Environment Variables	JSON
CloudFormation	Environment Variables, JSON	CLI
Azure Resource Manage	Parameter file	CLI
SaltStack	SLS file	JSON

Table 5.1: Some possibilities shown, how to import parameters and get an output

5.4 Implementation of the Transformation to DTSMs

Afterwards, the transformation to technology specific models follows. The implementation is similar to the concept given in Section 4.3. In the center is the transformation of groups.

The step consists now only of iterating over the group, using the correct implementation for each tech. All the implementations, called transformers, follow a similar pattern and have to implement the same predefined interface to simplify the usage, as shown in a simplified way in Figure 5.1. Additionally, it is easily extendable for further technologies. A transformer has access to a context object which contains the model, the group and the target path for the DTSM. With access to the model and the group all components and relations with their info can be read and transformed. To simplify this step Algorithm 4.2 was implemented as a static helper method that can be used by all transformers. Then this collected knowledge is used to generate the DTSM which is written to a file. Again, special consideration needs to be given to runtime properties. The implementations have to follow the principles defined in Section 4.3. Some suggestions how these can be handled is given in Table 5.1. All the results are stored in a corresponding folder for this step as can be seen in Listing 1. Through the context this folder is accessible. As a convention, every group gets its own folder where it can store artifacts and the DTSM. The name of the resulting folders for a were chosen as <step+technology>. This takes into account the order of deployment and is only done for clarity. It is not necessary to know the order of deployment in this step. The layout of the folder needs to be coordinated with the implementation of the executor of the same technology so that the DTSMs can be found. In this folder the DTSMs are viewable to verify what will be deployed in the execution step. These models will be deployed exactly as they are. The only missing thing are the dynamic properties which are clearly marked references as well.

Some technology specific details are given in the following. In **Terraform** the transformation starts with the creation of a context object for a group. Then for every component the necessary information is added to this context. Afterwards, this serves as input for a generic Terraform template file. This is dynamically substituted with the given context values. The order between the components of one group are guaranteed by Terraform when they are specified with the “depends_on” keyword. Special consideration was given to the fact that some values are needed in other groups and thus have to be exported. In the motivating scenario the hostname is such a case. Here, exported properties are written to a file with Terraform. Listing 4 shows a snippet for a template that exports a variable

[1] https://jgrapht.org/

Listing 4 Snippet of a template for Terraform DTSM

```
resource "local_file" "compute_${instance.name}" {
  content = jsonencode( {
      "hostname" = openstack_compute_instance_v2.${instance.name}.access_ip_v4
  })
  filename = "${instance.name}_computed_properties.json"
}
output "compute_${instance.name}_address" {
  value = openstack_compute_instance_v2.${instance.name}.access_ip_v4
}
```

Listing 5 Export Properties from Terraform model

```
resource "local_file" "compute_ubuntu_db" {
  content = jsonencode( {
      "hostname" = openstack_compute_instance_v2.ubuntu_db.access_ip_v4
  })
  filename = "ubuntu_db_computed_properties.json"
}
output "compute_ubuntu_db_address" {
  value = openstack_compute_instance_v2.ubuntu_db.access_ip_v4
}
```

and Listing 5 is the concrete instantiation for the reference scenario. For the templating Apache FreeMarker[2] is used. The plugin is now only responsible for providing the JSON files before the Terraform process is started and to read and write back exported variables at the end.

Ansible is implemented with a similar approach, where a Apache Freemarker templates are used to describe the wanted Ansible file. Needed variables are imported and exported with a JSON file. This is integrated in the templates.

A special case is represented by **Kubernetes**. It utilizes containers and abstracts away the notion of virtual machines. So, for a software to run it has to be provided as a docker image. The applications of a group are determined and transformed to Dockerfiles. To use it with Kubernetes, additionally, for every application a service and deployment file is created. In the deployment file, environment

Listing 6 Kubernetes model referenced properties

```
- name: "DB_HOSTNAME"
  valueFrom:
    configMapKeyRef:
      key: "DB_HOSTNAME"
      name: "pet-clinic-config"
- name: "DB_USER"
  valueFrom:
    configMapKeyRef:
      key: "DB_USER"
      name: "pet-clinic-config"
```

[2] https://freemarker.apache.org/

variables can be set. This is used to announce properties that will only be resolved during runtime as well. Listing 6 shows an example of that. These properties reference a ConfigMap for this component, which allows to look at the file and see all dependencies transparently. The full deployment file is shown in Appendix A.2. These files are generated with a Kubernetes library. Here, the templating is implemented by the library and it is enough the specific the Java objects which can then be exported in the correct format.

5.5 Implementation of Deployment Orchestration

The execution follows the basic principles of Section 4.5. Again, for every group a technology specific implementation an executor is called. The executor needs to implement the interface as shown in Figure 5.1. The order in which the executors are called with a group depends on the defined workflow from then previous step. The workflow is loaded from the file in the multi folder. The executor implementations have access to a context which contains the model and the path, where the DTSMs of the group to deploy are lying. For every group the one for the specific technology is called. When it is finished the next one is invoked. The task of the plugin is then as described. Get all necessary input parameters for the DTSM from the model. Utility functions like the collecting of all properties can be used by all plugins. Then, execute the technology in some way, mostly through invoking CLI commands or making calls to an API. At the end read exported properties and write them back to the model. This pattern must be followed by all implementations.

In **Ansible** these variables are saved in a JSON file, which a reference was inserted in the transformation step. For the execution of Ansible Ansible needs to be installed. The program just calls the Ansible command line interface, with a command similar to `ansible-playbook <filename>`. Other commands are not necessary.

The **Kubernetes** executor starts with the building of Docker images, for every application, which just translates to a Docker command. Additionally, the imported properties are written to a ConfigMap. Together with all deployments and services in the folder for the specific group they are then deployed. The deployment is handled by a Java package that invokes the Kubernetes API. If there are any dynamic parameters needed they can be read through the Kubernetes API and then written back to the model.

The **Terraform** executor starts with extracting properties from the model and writing them to a JSON file with the expected name. The DTSM generated in the previous step references this. The execution is then just the invoking of the command `terraform apply`. As with Ansible at the end per convention a JSON file exists with the exported variables. This is read and the properties are written back to the model for the corresponding component.

6 Conclusion and Outlook

The thesis introduced an approach for the orchestration of multiple deployment technologies. First, a tool agnostic modeling was defined on basis of the EDMM [29], incorporating the additional requirements with the usage of multiple tools. Such as labeling the components. This raised a problem that all components were completely isolated and not a group. A solution inspired by Saatkamp et al. [23] was provided, which groups all components with the same technology, where no cycle is induced. Next, a workflow was created by sorting the dependency graph topologically. Afterwards, a more difficult problem, the transformation of groups, was looked at. Special attention was given to the interactions between groups and different technologies. Here, some conventions and a interface was defined that the technology specific plugins have to follow and implement. Similarly, the automatic execution was handled. To validate the concept, a prototype was implemented.

In conclusion, the agnostic model simplifies the switch between technologies and the possibility to get a clear overview is improved. Additionally, multiple deployment tools can interact through the framework. Everything is automated and no manual tasks are necessary anymore after the model definition. Lastly, the framework can easily extended for new technologies, because it contains clearly defined interfaces and everything else is handled by the framework already. A case study in the industry could identify further points that need improvements or are needed for a productive usage.

The thesis serves as a basis for an upcoming paper with the title“Application Orchestration Using Multiple Production-Ready Deployment Technologies” [30].

In the future the framework could be extended to enable a more decentralized deployment, where different departments or companies can keep their internal information and APIs private, as described by Wild et al. [27]. So far, it uses a centralized approach. Furthermore, the framework could be extended to manage the complete lifecycle of a deployment, e.g. the possibility to allow changes or shut it down again.

The prototype can be improved as well. In the future the modelling process could use a graphical user interface to show and design the model. In contrast to a YAML file this ensures a better usability and allows to get a better overview of the deployment.

Bibliography

[1] H. Alipour, Y. Liu. “Model driven deployment of auto-scaling services on multiple clouds”. In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2018, pp. 93–96 (cit. on p. 22).

[2] Amazon Web Services. *AWS CloudFormation*. URL: https://aws.amazon.com/cloudformation/ (cit. on p. 13).

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. *Business process execution language for web services*. 2003 (cit. on p. 38).

[4] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, D. A. Tamburri. “DevOps: introducing infrastructure-as-code”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 497–498 (cit. on p. 13).

[5] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. “A Systematic Review of Cloud Modeling Languages”. In: *ACM Computing Surveys* 51.1 (2018). DOI: 10.1145/3150227 (cit. on p. 21).

[6] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “TOSCA: portable automated deployment and management of cloud applications”. In: *Advanced Web Services*. Springer, 2014, pp. 527–549 (cit. on p. 21).

[7] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. “Combining declarative and imperative cloud application provisioning based on TOSCA”. In: *2014 IEEE international conference on cloud engineering*. IEEE. 2014, pp. 87–96 (cit. on p. 37).

[8] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, A. Agahi. “Automated synthesis and deployment of cloud applications”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 211–222 (cit. on p. 21).

[9] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. “Declarative vs. imperative: two modeling patterns for the automated deployment of applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. 2017, pp. 22–27 (cit. on p. 15).

[10] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg. “Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems”. In: *2013 IEEE Sixth International Conference on cloud computing*. IEEE. 2013, pp. 887–894 (cit. on p. 21).

[11] C. Foundry. *Bosh*. URL: https://bosh.io/ (cit. on p. 21).

[12] J. Guillén, J. Miranda, J. M. Murillo, C. Canal. “A service-oriented framework for developing cross cloud migratable software”. In: *Journal of Systems and Software* 86.9 (2013), pp. 2294–2308 (cit. on p. 22).

[13] HashiCorp. *Terraform*. URL: https://www.terraform.io/ (cit. on p. 13).

[14] H. Herry, P. Anderson, G. Wickler. "Automated Planning for Configuration Changes". In: *LISA*. 2011 (cit. on p. 15).

[15] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010 (cit. on p. 13).

[16] A. B. Kahn. "Topological sorting of large networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562 (cit. on p. 38).

[17] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, T. Männistö. "The highways and country roads to continuous deployment". In: *Ieee software* 32.2 (2015), pp. 64–72 (cit. on p. 13).

[18] K. Morris. *Infrastructure as code: managing servers in the cloud*. Ö'Reilly Media, Inc.", 2016 (cit. on p. 13).

[19] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS), 2019 (cit. on p. 21).

[20] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG). 2011 (cit. on p. 38).

[21] D. Oppenheimer, A. Ganapathi, D. A. Patterson. "Why do Internet services fail, and what can be done about it?" In: *USENIX symposium on internet technologies and systems*. Vol. 67. Seattle, WA. 2003 (cit. on p. 13).

[22] Opscode Inc. *Chef*. URL: https://chef.io/ (cit. on p. 13).

[23] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Method, formalization, and algorithms to split topology models for distributed cloud application deployments". In: *Computing* (2019), pp. 1–21 (cit. on pp. 27, 31, 53).

[24] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, G. Jung. "Approaches for service deployment". In: *IEEE Internet Computing* 9.2 (2005), pp. 70–80 (cit. on p. 15).

[25] J. Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116 (cit. on p. 13).

[26] J. Wettinger, U. Breitenbücher, O. Kopp, F. Leymann. "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel". In: *Future Generation Computer Systems* 56 (2016), pp. 317–332 (cit. on p. 21).

[27] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. "Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2020, pp. 20–35 (cit. on p. 53).

[28] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. "The EDMM modeling and transformation system". In: *International Conference on Service-Oriented Computing*. Springer. 2019, pp. 294–298 (cit. on pp. 13, 45).

[29] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies". In: *arXiv preprint arXiv:1905.07314* (2019) (cit. on pp. 13, 15, 16, 23, 53).

[30] M. Wurster, U. Breitenbücher, F. Leymann, K. Saatkamp, F. Diez. “Application Orchestration Using Multiple Production-Ready Deployment Technologies”. in publication. 2020 (cit. on p. 53).

[31] L. Zhu, L. Bass, G. Champlin-Scharff. “DevOps and its practices”. In: *IEEE Software* 33.3 (2016), pp. 32–34 (cit. on p. 13).

All links were last followed on July 25, 2020.

A Appendix

A.1 Model of the Motivating Scenario for the Prototype

```
orchestration_technology:
  ansible:
    - db
    - dbms
  terraform:
    - ubuntu_db
  kubernetes:
    - ubuntu_app
    - pet_clinic_tomcat
    - pet_clinic
components:
  ## Spring PetClinic Application
  pet_clinic:
    type: web_application
    artifacts:
      - war: ./files/petclinic/petclinic.war
    operations:
      configure: ./files/petclinic/configure.sh
    relations:
      - hosted_on: pet_clinic_tomcat
      - connects_to: db
    properties:
      db_hostname: ${db.hostname}
      db_user: ${db.user}
      db_password: ${db.password}
      db_port: ${db.port}

  pet_clinic_tomcat:
    type: tomcat
    operations:
      create: ./files/tomcat/create.sh
      start: ./files/tomcat/start.sh
    relations:
      - hosted_on: ubuntu_app

  ## Database
  db:
    type: mysql_database
    properties:
      schema_name: petclinic
      user: pc
      password: petclinic
    artifacts:
      - sql: ./files/petclinic/schema.sql
    operations:
```

```
    configure: ./files/mysql_database/configure.sh
  relations:
    - hosted_on: dbms

dbms:
  type: mysql_dbms
  properties:
    root_password: petclinic
  operations:
    create: ./files/mysql_dbms/create.sh
    start: ./files/mysql_dbms/start.sh
  relations:
    - hosted_on: ubuntu_db

## Compute
ubuntu_app:
  type: compute
  properties:
    machine_image: ubuntu
    instance_type: large
    key_name: wursteml
    priv_key_path: ./files/ubuntu/key.pem

ubuntu_db:
  type: compute
  properties:
    machine_image: ubuntu
    instance_type: large
    key_name: wursteml
    priv_key_path: ./files/ubuntu/key.pem
```

A.2 Kubernetes Deployment

```
apiVersion: "apps/v1"
kind: "Deployment"
metadata:
  annotations: {}
  labels:
    app: "pet-clinic"
  name: "pet-clinic"
spec:
  replicas: 1
  selector:
    matchLabels:
      app: "pet-clinic"
  template:
    metadata:
      annotations: {}
      labels:
        app: "pet-clinic"
      name: "pet-clinic"
    spec:
      containers:
      - env:
        - name: "OS_FAMILY"
          value: "linux"
        - name: "MACHINE_IMAGE"
          value: "ubuntu"
        - name: "PRIV_KEY_PATH"
          value: "./files/ubuntu/key.pem"
        - name: "PORT"
          value: "8080"
        - name: "INSTANCE_TYPE"
          value: "large"
        - name: "DB_PORT"
          valueFrom:
            configMapKeyRef:
              key: "DB_PORT"
              name: "pet-clinic-config"
        - name: "DB_PASSWORD"
          valueFrom:
            configMapKeyRef:
              key: "DB_PASSWORD"
              name: "pet-clinic-config"
        - name: "DB_HOSTNAME"
          valueFrom:
            configMapKeyRef:
              key: "DB_HOSTNAME"
              name: "pet-clinic-config"
        - name: "DB_USER"
          valueFrom:
            configMapKeyRef:
              key: "DB_USER"
              name: "pet-clinic-config"
        image: "pet-clinic:latest"
        imagePullPolicy: "Never"
        name: "pet-clinic"
        ports:
```

```
  - containerPort: 8080
    name: "pet-clinic-tom"
nodeSelector: {}
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature