Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Decentralized Cross-Organizational Application Deployment Using Multiple Different Deployment Automation Technologies

Nakharin Donsuypae

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Karoline Wild, M.Sc., Michael Wurster, M.Sc.
Commenced:	September 9, 2020
Completed:	March 9, 2021

Abstract

With the rise of cloud computing, the automated deployment and management of applications has become increasingly important. Manual execution of deployment steps can become error-prone, time-consuming, and costly. Therefore, deployment technologies are necessary to achieve a high level of automation. However, the majority of these technologies have one common drawback, they all use a central orchestrator for the deployment execution. Most often, multiple departments or even companies participate in the deployment process. Moreover, most deployments are not based on workflow technologies, which restrict the orchestration capabilities as well as the customization of complex deployment logic that may be needed for the deployment of complex applications. Furthermore, often more than one deployment technology is used, e.g. Terraform for setting up the virtual machine and Kubernetes for running the applications on top of it. As a result, the orchestration of the deployment (i) between participants and (ii) different deployment technologies for the correct deployment execution and data exchange is necessary. To address these challenges, this work proposes an approach to enable decentralized cross-organizational application deployment based on multiple deployment technologies. This is done by (i) annotation of a deployment model with participant information and (ii) generation of participant-specific workflows orchestrating the deployment with different technologies and data exchange between participants. To prove the feasibility of this approach, a prototypical implementation and an accompanying case study is provided.

Kurzfassung

Mit der steigenden Nutzung von Cloud Computing ist die automatisierte Bereitstellung und Verwaltung von Anwendungen ein entscheidendes Thema geworden. Die manuelle Ausführung kann daher fehleranfällig, zeitaufwändig und sehr kostspielig werden. Daher sind Deployment-Technologien notwendig um einen hohen Automatisierungsgrad zu erreichen. Die meisten Technologien weisen jedoch einen gemeinsamen Nachteil auf: Sie verwenden einen zentralen Orchestrator für die Ausführung des Deployments. Allerdings sind in den meisten Fällen mehrere Abteilungen oder auch Unternehmen am Deployment beteiligt. Ebenfalls basieren die meisten Deployments nicht auf Workflow-Technologien, was somit die Orchestrierungsmöglichkeiten, sowie komplexe Deployment-Logik einschränkt. Darüber hinaus wird oft mehr als nur eine Deployment-Technologie verwendet, wie z.B. Terraform für das Aufsetzen der virtuellen Maschine und Kubernetes für die Ausführung der Applikation. Dementsprechend ist die Orchestrierung des Deployments zwischen (i) mehreren Teilnehmern und (ii) verschiedenen Deployment-Technologien für die korrekte Ausführung des Deployments notwendig. Daher wird in dieser Arbeit ein Ansatz vorgeschlagen, welches ein dezentrales Deployment auf Basis mehrerer Deployment-Technologien ermöglicht. Dies geschieht durch die (i) Annotation eines Deployment-Modells mit teilnehmerrelevanten Deployment-Informationen und (ii) die Generierung von teilnehmerspezifischen Workflows, die das Deployment mit verschiedenen Technologien und den Datenaustausch zwischen den Teilnehmern orchestrieren. Um den vorgestellten Ansatz zu validieren, wird eine prototypische Implementierung und eine begleitende Fallstudie bereitgestellt.

Contents

1 Introduction **15**

2 Fundamentals, Motivating Scenario, and Problem Statement **17**
2.1 Cloud Application Deployment 17
2.2 Essential Deployment Meta Model (EDMM) 20
2.3 Business Process Model and Notation (BPMN) 22
2.4 Motivating Scenario and Problem Statement 23

3 Related Work **25**
3.1 Automated and Distributed Cloud Application Deployment 25
3.2 Cloud Application Deployment Using Choreographies 26

4 Decentralized Orchestration and Deployment with Multiple Technologies **29**
4.1 Create Global EDMM model 30
4.2 Define Multi-Participant Regions 32
4.3 Define Technology-Specific Regions 33
4.4 Divide Model and Generate Partial Models 35
4.5 Transform Partial Models to DTSMs 36
4.6 Determine Deployment Order 36
4.7 Generate BPMN Workflow 37
4.8 Execute Automated Deployment 42
4.9 System Architecture 43
4.10 Discussion and Limitations 44

5 Prototypical Implementation and Case Study **47**
5.1 EDMM Model Specification 47
5.2 Overview Extended Transformation Framework 49
5.3 Case Study 54

6 Conclusion and Future Work **57**

Bibliography **59**

A Appendix **63**

List of Figures

2.1 Declarative Deployment Approach . . . 18
2.2 Imperative Deployment Approach . . . 19
2.3 The Essential Deployment Meta Model from Wurster et al. [WBF+20]. . . . 20
2.4 Architecture of the EDMM Transformation System [WBB+19]. . . . 21
2.5 An example BPMN process . . . 22
2.6 Deployment scenario with three different deployment technologies and two participants 23

4.1 Concept for a decentralized deployment using multiple different deployment technologies and participants . . . 29
4.2 Simple EDMM model including components, relations, and property values . . . 31
4.3 Deployment scenario with two participants annotated . . . 32
4.4 Shared EDMM model by multiple participants . . . 33
4.5 Deployment scenario with two deployment technologies used in the perspective of participant B . . . 34
4.6 Algorithm to divide and merge the EDMM model to generate partial models, based on [WBL+21] . . . 35
4.7 Generated deployment order including affected components, target technology, and necessary input property values in the perspective of participant B . . . 36
4.8 Generated workflow in BPMN for participant B . . . 38
4.9 Sample Multi Receive BPMN activity based on an EDMM model . . . 40
4.10 Generated Task Type using the $determineTaskType$ algorithm . . . 41
4.11 System architecture of the extended EDMM Transformation Framework with new components in white and modified components in gray. . . . 43

5.1 Simplified UML class diagram of the extended EDMM Transformation Framework 50
5.2 Simple Graphical User Interface for the automated management of BPMN workflow files . . . 52
5.3 Generated BPMN workflow of participant A . . . 54
5.4 Generated BPMN workflow of participant B . . . 54

List of Listings

4.1 BPMN Deploy 39
4.2 BPMN Send 39
5.1 Extended EDMM Model with three deployment technologies and two participants 48
5.2 Sample component of an EDMM model 48
5.3 Deploy REST JSON Body 49
5.4 Deploy BPMN Template 51
5.5 Structure of created transformation files in the perspective of participant B 53

List of Algorithms

4.1 $GenerateExecutableProvisioningPlan(p_i, provisioning\_order\_graph_i)$ . . . 40
4.2 $DetermineTaskType(p_i, group_{i-1}, group_i, group_{i+1})$ 40

Acronyms

API Application Programming Interface. 15

BPEL Business Process Execution Language. 18

BPMN Business Process Model and Notation. 16

CI Continuous Integration. 45

CLI Command Line Interface. 21

DTSM Deployment Technology-Specific Model Fragments. 23

EDMM Essential Deployment Meta Model. 15

GDM Global Deployment Model. 29

GUI Graphical User Interface. 42

IaaS Infrastructure as a Service. 19

LDG Local Deployment Group. 29

LDM Local Deployment Model. 29

REST Representational State Transfer. 21

SaaS Software as a Service. 19

VM Virtual Machine. 15

1 Introduction

With the rise of cloud computing, the success of companies is often determined by the automated deployment and management of their applications and cloud infrastructure. Today, applications are becoming more complex and cloud computing has changed the way how the provisioning of these applications are executed: For instance, instead of buying the hardware required for the deployment, IT resources, such as Virtual Machine (VM), can be consumed in a cost-effective manner with a pay-per-use model [MG+11]. Therefore, to fully exploit the potentials of cloud computing, the automated deployment of complex distributed applications has become increasingly important. Since manually executing the deployment of such applications is error-prone, time-consuming, and costly [BCS18], numerous different technologies have been developed for the automation of deployment, configuration, and management of applications. These applications typically consist of multiple components and can vary from the technologies that have to be used to deploy them. For example, Terraform [Ter21] is used for the creation of a VM, whereas Ansible [Ans21] is responsible for the configuration of the created VM. Some of the popular deployment automation technologies used in industry are, for instance, Chef [Che21], Terraform, or Ansible. The majority of these technologies are not limited to a specific infrastructure and able to manage multi-cloud applications but they have one common drawback, they all use a central orchestrator for the deployment execution.

However, in practice often multiple application components are managed in a distributed environment, with components belonging to different departments or even different companies. Security issues and potential attacks can arise from a centralized deployment and therefore participants want to keep control over the deployment and not disclose where and how each component is hosted internally. Since one company might not want to expose their internal Application Programming Interface (API) to the outside or leave their credentials to other participants due to legal and compliance rules that have to be followed. Consequently, centralized deployment technologies are not suitable for a cross-organizational deployment [WBK+20].

To tackle these challenges, Wild et al. [WBK+20] introduced an approach to enable decentralized cross-organizational application deployment. For this, for each participant a workflow is generated that orchestrates the local deployment tasks as well as handle the message exchange with the other participants. As a result, all workflows form implicitly the deployment choreography. However, this approach is limited to TOSCA [OAS21] as a modeling language and OpenTOSCA [BBH+13] as a deployment engine, which in practice is not widely used. Therefore, Wurster et al. [WBF+20] investigated 13 deployment technologies, extracted their essential parts and introduced the Essential Deployment Meta Model (EDMM). EDMM provides a common understanding of declarative deployment technologies and allows with the EDMM Transformation Framework [WBB+19] the creation of technology-agnostic deployment models that can be translated into one of the 13 investigated deployment technologies [WBB+19]. With an extension, also the combination of several deployment technologies is possible, e.g. the creation of VMs with Terraform and the

modification of the created VM through Ansible [WBL+21]. The components in the EDMM model are annotated with the required deployment technology and the different parts are then transformed and a central orchestrator executes the fragments in the correct order.

However, the usage of different production-ready deployment technologies is currently limited to a centralized approach and is not based on workflow technologies which restrict the orchestration capabilities as well as the customization of complex deployment logic that may be needed for the deployment of complex applications [BBK+14]. With the use of workflow technologies, complex applications and their respective deployment steps can be executed fully automatically and do not have to be deployed manually by the application developers. Additionally, widely adopted standards, such as Business Process Model and Notation (BPMN) [OMG11], allows the usage of different workflow engine providers or the existing workflow engine in a company.

To tackle the aforementioned issues, this work proposes an approach to enable decentralized cross-organizational application deployment using multiple different deployment automation technologies. For this, the two approaches for decentralized orchestration by Wild et al. [WBK+20] and the deployment with multiple deployment technologies by Wurster et al. [WBL+21] are combined. To achieve the objective of decentralized orchestration, the technology-agnostic EDMM model has been (i) extended for the annotation of different participants. With the annotated EDMM models, (ii) a participant-specific workflow is generated for each participant to orchestrate the deployment of the participant's components with different deployment technologies and message exchange. And thus, all workflows together form implicitly the deployment choreography. Conclusively, the feasibility of the presented approach is prototypically implemented with the EDMM Transformation Framework and validated in a case study.

The thesis is divided into the following chapters:

Chapter 2 – Fundamentals, Motivating Scenario, and Problem Statement: In this chapter, the fundamentals of this work are presented. Cloud Application Deployment, the Essential Deployment Meta Model, and the Business Process Model and Notation is introduced. At last, a problem statement introduces the challenges that have to be tackled in this work.

Chapter 3 – Related Work: This chapter introduces related work in the field of automated distributed cloud deployment, and cloud application deployment using choreographies.

Chapter 4 – Decentralized Orchestration and Deployment with Multiple Technologies: In this chapter, an approach to enable decentralized cross-organizational application deployment with multiple technologies is introduced. In the end, a discussion and the limitations of the approach are presented.

Chapter 5 – Prototypical Implementation and Case Study: The prototypical implementation is presented and validated based on a simplified case study.

Chapter 6 – Conclusion and Future Work: This chapter provides a conclusion of this work and an outlook to future work.

2 Fundamentals, Motivating Scenario, and Problem Statement

This chapter provides the fundamentals, that are required to understand used terms and the remainder of this work. First, an introduction to the term Cloud Application Deployment is presented and an overview of deployment technologies are provided. Moreover, the Essential Deployment Meta Model and the Business Process Model Notation, which are relevant for this work are covered. Finally, a problem statement is illustrated and the challenges that have to be tackled are introduced.

2.1 Cloud Application Deployment

Due to the heavy use of IT in every aspects of a company, the increasing management effort has become a challenge for companies, as new technologies can increase the degree of complexity [BBKL14]. To deal with the increasing complexity, the automation of IT management has become an important topic and automation technologies have paved the way on how companies use and think about IT [BBKL14]. These automation technologies are both enabled and supported by cloud computing and according to Hentschel et al. [HLB19], the efficient use of cloud computing can be considered one of the most important technological drivers of the digitalization of enterprises [HLB19].

Since modern enterprise applications typically consist of complex composite applications with multiple individual components, the process to deploy an application can often be error-prone and complex, if done manually. As a result, the efficient use of deployment technologies has become increasingly important and many cloud service offerings support the automated deployment of applications [EBF+17]. Besides these technologies, standards, for example TOSCA, have been established to describe the topology of cloud-based web services and offer high automation, reusability, and easy usage in order to operate the business functionality [EBF+17]. Even though the features of aforementioned technologies, APIs, and standards vary, they have in common that they support the same deployment automation principles and can be categorized in: (i) declarative and (ii) imperative deployment modeling approaches [EBF+17].

2.1.1 Declarative Deployment Approach

In a declarative deployment modeling approach, structural models that describe the desired application structure and state are used. These models are then interpreted by a deployment system to enforce the desired state [EBF+17][WBF+20]. In Figure 2.1, on the left side, an exemplary declarative deployment model is shown, describing all components that have to be deployed as well as their relationships. In this model, a *Java Web Application* that needs to be deployed on a *Tomcat*

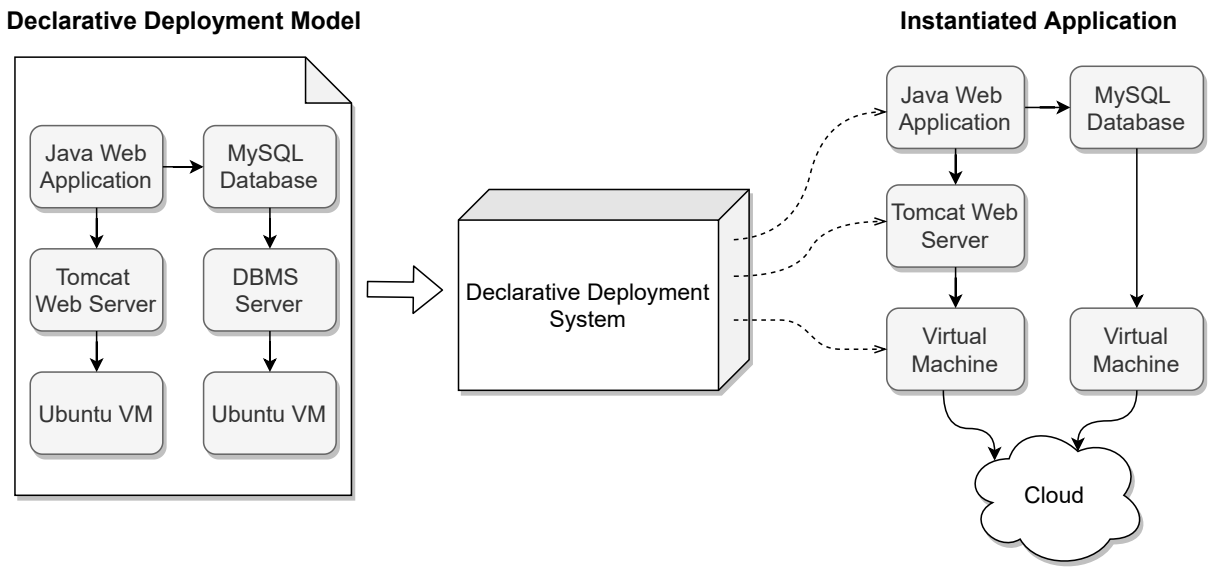

Figure 2.1: Declarative Deployment Approach

Web Server running on a *Ubuntu VM* and provisioned on an arbitrary cloud provider is shown. In addition, the Java Web Application is connected to a *MySQL database* hosted on a *DBMS server* and running on a Ubuntu VM. The model is then interpreted by a deployment system that derives and executes the model to get the desired state. In the presented example, the Ubuntu VM needs to be deployed on a cloud first before the Tomcat Web Server and the DBMS Server are installed on the Ubuntu VM. After the MySQL Database is set up and the DBMS Server is installed, the Java Web Application is deployed on the Tomcat Web Server and connects to the MySQL Database. If the desired state of the model is reached, the deployment is finished. As a result, declarative deployment models are suitable for deployment of standard applications that consist of well-known, common components but lack the arbitrarily customization of complex deployment steps [EBF+17].

2.1.2 Imperative Deployment Approach

In contrast, the imperative deployment modeling approach describes all deployment tasks that have to be executed in the form of a process [EBF+17][WBF+20]. An exemplary abstract imperative deployment model shown in Figure 2.2, is interpreted by the imperative deployment system and executes the model. For such models, standard workflow languages, such as Business Process Execution Language (BPEL) or BPMN can be used. Each task in the workflow is defined explicitly and depending on the task type, for example, an API of a service that executes the required operation or a script can be executed. Thus, arbitrary logic gets executed as specified and custom deployments is possible. However, immense technical deployment expertise is needed and the creation of such a model can become complex, error-prone, and time-consuming [EBF+17].

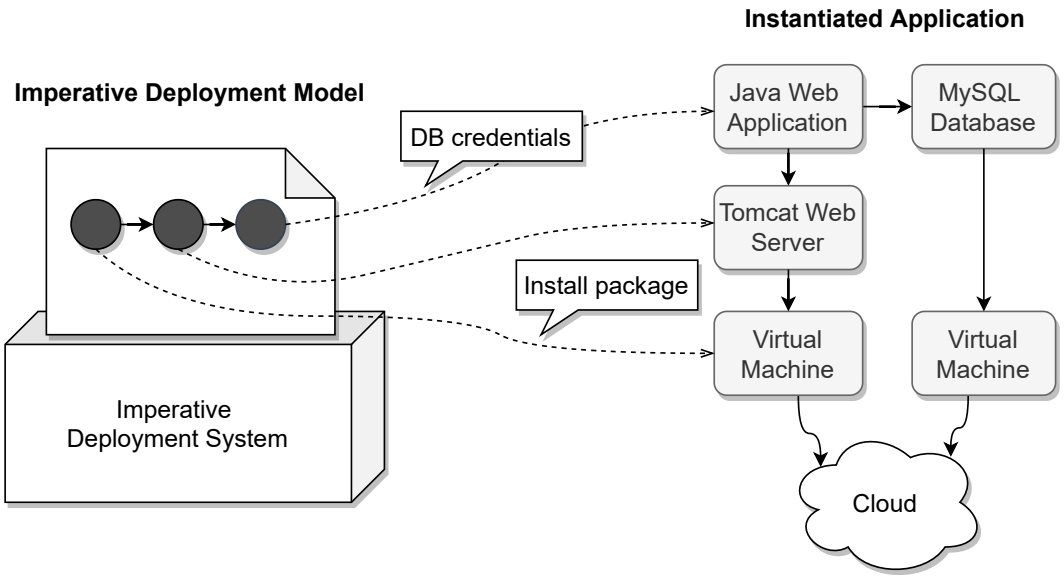

Figure 2.2: Imperative Deployment Approach

2.1.3 Deployment Technologies

Deployment technologies have been released to accomplish the need for deployment of applications. According to Wurster et al. [WBF+20], declarative deployment models are widely accepted in industry and research as the most appropriate approach for application deployment and configuration management. As a result, various different technologies, such as Chef [Che21], Puppet [Pup], AWS CloudFormation [Clo], Azure Resource Manager [Mic21], Terraform [Ter21], and Kubernetes [Kub21] follow this approach. Although they share the same approach, they differ in their modeling language, supported features, and mechanisms [WBF+20]. To further break down deployment technologies, in a review conducted by Wurster et al. [WBF+20], they can be divided into three categories: (i) general-purpose deployment technologies, (ii) provider-specific deployment technologies, and (iii) platform-specific deployment technologies.

General-purpose deployment technologies are characterized by the support of all deployment features and mechanisms as well as their support of single-, hybrid-, multi-cloud deployments, and different kinds of cloud services, such as Infrastructure as a Service (IaaS), and Software as a Service (SaaS). Furthermore, they can be extended by reusable and customized components for various providers or services [WBF+20]. Tools in this category are for example Puppet, Chef, Ansible, and Terraform. On the contrary, *provider-specific* deployment technologies only support single-cloud deployments as they are offered by specific cloud providers and therefore only support services of the respective provider [WBF+20]. Examples for this group are AWS CloudFormation and Azure Resource Manager. *Platform-specific* deployment technologies, are in contrast, restricted to the cloud delivery model and need to conform to a specific platform [WBF+20]. For instance, a container runtime and format is needed to deploy applications with Kubernetes. Technologies in this group are e.g. Kubernetes and Docker Compose.

These three categories have shown that often multiple tools are needed to solve specific challenges and it is difficult to compare technologies by their capabilities, as they all have their own modeling language and mechanisms. To tackle this issue, a more technology-agnostic way is needed and as a result, Wurster et al. [WBF+20] introduced the Essential Deployment Meta Model.

2.2 Essential Deployment Meta Model (EDMM)

Due to the large amount of deployment technologies offered, it has become quite a challenge to choose the most appropriate technology and also determine the right one for each case. Furthermore, in cases where applications have to be migrated to the chosen deployment technology and target environment, technology-specific knowledge of these features and mechanisms are needed [WBF+20]. Therefore, Wurster et al. [WBF+20] conducted a systematic review and investigated 13 of the most popular deployment technologies, applied in industry and research, to extract their essential parts and provide a common understanding of these technologies. As a result, the Essential Deployment Meta Model (EDMM) has been derived. The EDMM encompasses the essential parts of declarative deployment models and showed that all analyzed technologies complied to the EDMM semantically. Consequently, deployment structures, such as components and their relations can be described through the EDMM [WBF+20].

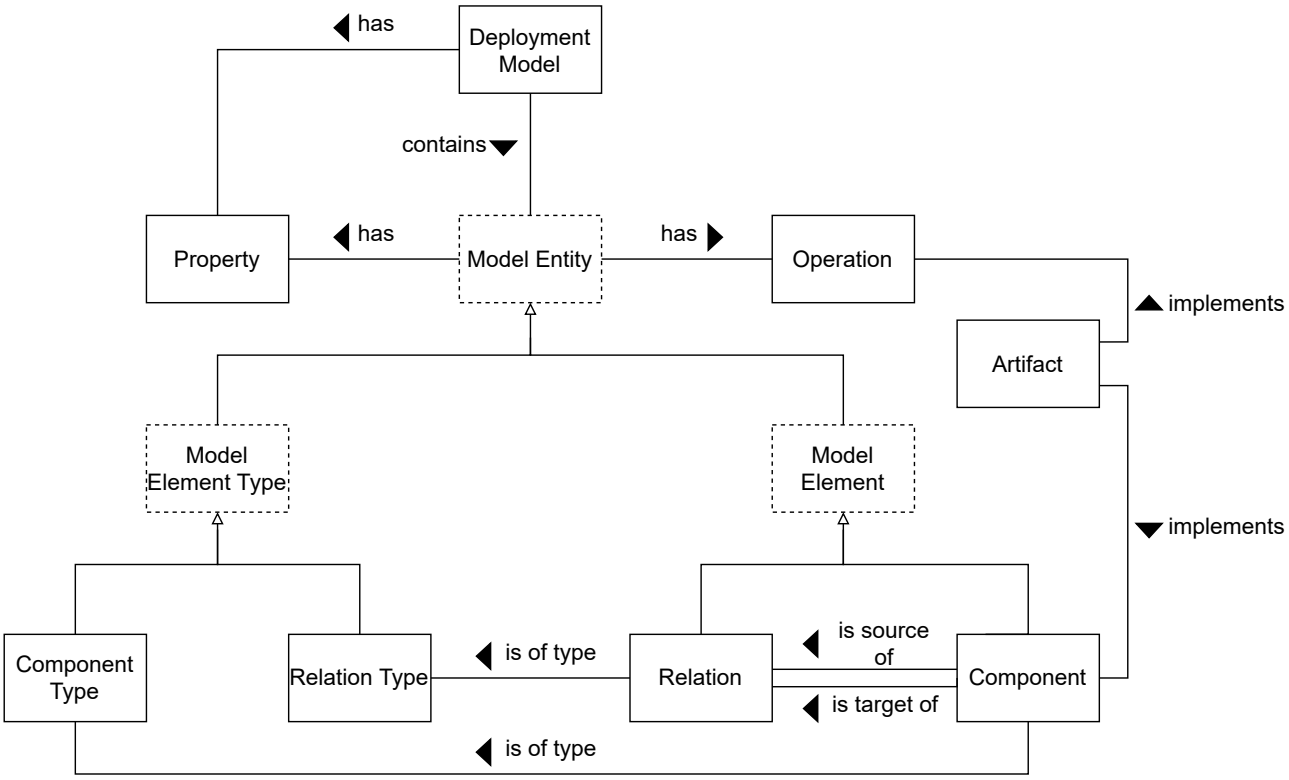

Figure 2.3: The Essential Deployment Meta Model from Wurster et al. [WBF+20].

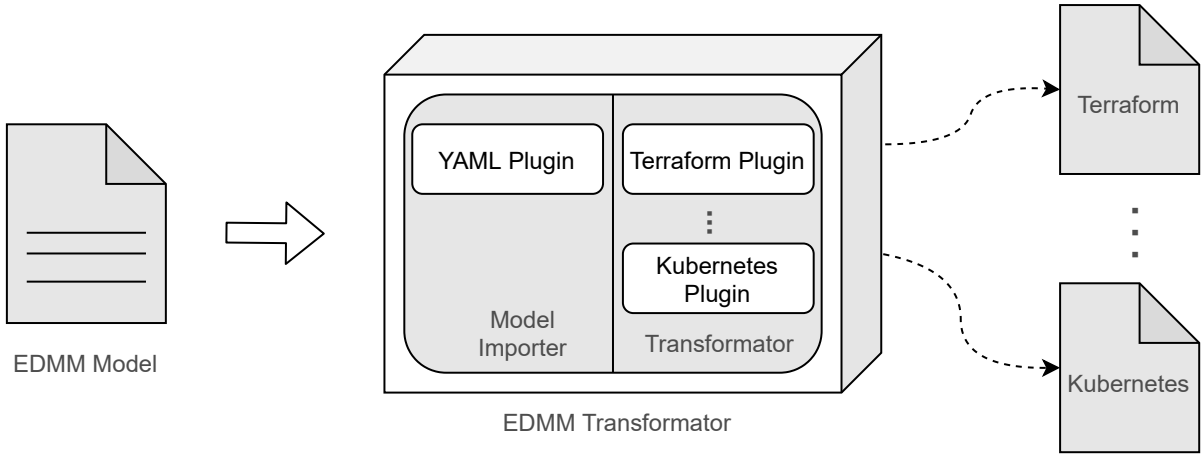

Figure 2.4: Architecture of the EDMM Transformation System [WBB+19].

In Figure 2.3, the EDMM is depicted and illustrates the structure and relation of each entity. The entity *Component* describes a physical, functional, or logical part of the application [WBF+20]. This can be, for instance, a database that stores the data of a user or a newly created virtual machine, that is running on a platform. The *Component Type* defines a reusable entity and gives semantics to a component. For example, components can be applied a specific semantic meaning, such as the component *Order App* has the component type *Java-based Web Application* or the component *Tomcat* has the component type *Tomcat Server*. Directed, physical, or logical dependencies between two components are called *Relation*. Similar to components and component types, *Relation Type* is a reusable entity which gives semantic meaning to a relation. As an example, the relation of the *Order App* to *Tomcat* of the type *hosted on* describes that the *Order App* has to be installed on top of the *Tomcat Server*. Whereas the relation of the type *connects to* describes a component that needs to establish a connection, such as a database. An *Operation* describes an executable procedure that is performed to manage a component or relation. To give an example, bash scripts that are executed on a VM to modify the database are operations. With *Property*, the current state or desired target state or configuration of a component or relation are described. Properties are for instance the IP address or port of a VM that needs to be accessed.

Moreover, with the introduction of the *EDMM Transformation Framework*, created EDMM model files can be transformed into the desired target deployment technology [WBB+19]. This is done by the Command Line Interface (CLI) or a provided Representational State Transfer (REST) endpoint. As an input, YAML files, which depict the EDMM model are supported and all components and component types must be provided in the model. After the transformation, the output is an executable, technology-specific deployment model, that can be executed using the selected technology [WBB+19].

The architecture of the EDMM Transformation Framework, illustrated in Figure 2.4, employs a plugin architecture that deploys various deployment technologies through integrated plugin modules. Each plugin defines a deployment technology and provides an implementation to transform EDMM-based models into the implemented target technology. As an example, the Kubernetes plugin allows the transformation of the EDMM model file to Kubernetes resource files. With this, technology-specific commands can then be applied to the resource files by the user. With an extension, the combination of several deployment technologies is possible as well. The components in the EDMM

model have to be annotated with the required technology and different parts are then transformed and executed in the correct order. However, this approach is currently not based on workflow technologies and therefore restrict the orchestration capabilities as well as compensation, scalability, and the customization of complex applications. To tackle this issue, workflow technologies to execute business process models are needed.

2.3 Business Process Model and Notation (BPMN)

Business Process Model and Notation (BPMN) [OMG11] is a specification developed by the Business Process Management Initiative. It is considered a standard for representing business processes in a graphical way and finds its usages in every kind of organization and processes, such as travel booking procedures or text document creation [CT12]. With the addition of workflow engines, such as Camunda [Eng21], the execution can be automated. In addition, due to its simple semantics, BPMN provides a notation that is readily understandable by everyone, starting from business analysts who create drafts of the process to the software developers that implement the technical activities [Whi04].

BPMN consists of a large variety of elements and allows the development of simple diagrams that resemble flowchart diagrams [Whi04]. Basic categories of element are, for instance, *Flow Objects* and *Connecting Objects* [Whi04]. Flow objects are a small set of core elements, including *Events*, *Activites*, and *Gateways*. Events are represented by a circle and describes something that happens during the course of a process. An activity is represented by a rounded rectangle and specifies work that is performed within the process. An example activity could be for example a task to automatically send out data to the customer. Gateways are represented by the diamond shape and is used to control the divergence and convergence of the sequence flows and thus determine what path is taken in a process [Whi04]. Connecting objects are used to connect flow objects and is represented by *Sequence Flows*. The sequence flow is represented by a solid line with a solid arrowhead and is used to show the order that flow objects will be performed in a process.

In Figure 2.5, a BPMN process is illustrated and describes a simple scenario in which an order is received and the order data processed. After this step, the membership of the user and their order is evaluated through the gateway and determines what path is taken next. If the user is a registered member, the order is processed and the process is finished. On the other hand, if the user is not a registered member, a rejection mail is sent and the process is finished.

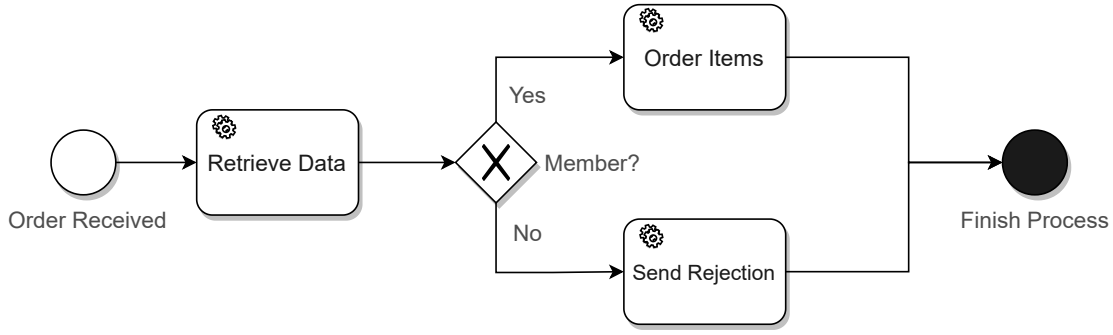

Figure 2.5: An example BPMN process

2.4 Motivating Scenario and Problem Statement

With deployment technologies, the deployment, configuration, and management of applications can be automated. The introduced EDMM enables the transformation to 13 deployment technologies and with the extension of Wurster et al. [WBL+21] different technologies can be combined. This is done by (i) extending the EDMM model with technology-specific regions indicating which deployment technology is used for the deployment and then (ii) dividing the EDMM model into Deployment Technology-Specific Model Fragments (DTSM). A DTSM is a group of components that can be individually deployed with the same technology and executed at once to deploy the deployment group. Moreover, (iii) a deployment order is created and determines how the overall application has to be deployed. Since different technologies provide different outputs, a central orchestrator coordinates the deployment of technologies by triggering the respective CLIs and APIs.

However, the approach is limited to a central orchestrator and therefore does not provide the benefits of a decentralized deployment, such as cross-organizational collaborations [WBK+20]. Additionally, the created deployment order is not based on workflow technologies and therefore lack robustness and scalability. To generate deployment choreographies based on deployment models, Wild et al. [WBK+20] introduced an approach by (i) annotating multiple participants in the deployment

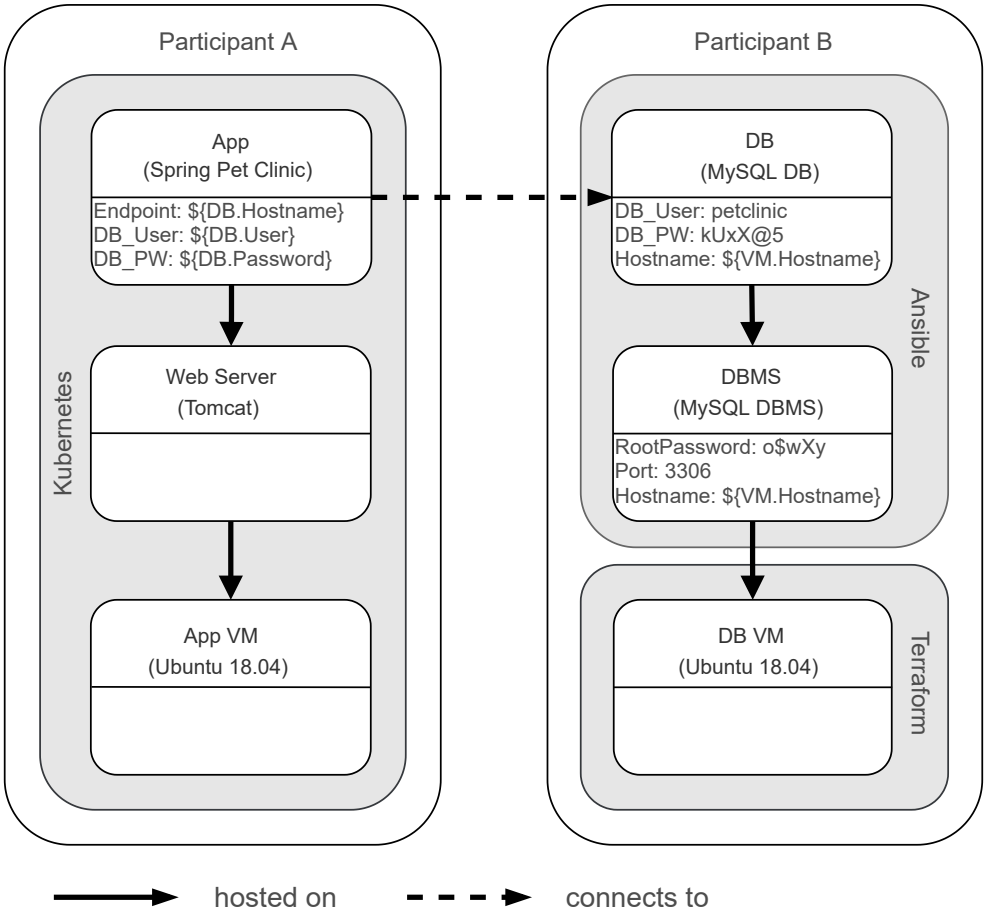

Figure 2.6: Deployment scenario with three different deployment technologies and two participants

model and (ii) based on that, for each participant a participant-specific workflow is generated. All created workflows form implicitly the deployment choreography and communicate for the data exchange [WBK+20]. This approach, however, is only limited for TOSCA and OpenTOSCA, which in practice is not widely used.

To emphasis the vision of this work, a simple deployment scenario has been chosen as a motivating scenario and is illustrated in Figure 2.6. The presented scenario describes an *App*, Spring Pet Clinic [App21], which is hosted on a Tomcat *Web Server* and the Tomcat Web Server is hosted on a Ubuntu *App VM*. On the right side, a database *DB* is hosted on a *DBMS*, in which the DBMS is hosted on a Ubuntu *DB VM*. In addition, the App establishes a connection to the DB. To further demonstrate multi-participant data exchange, components are annotated with a *target technology* and the respective *participant*. Participants describe the owner of the component and also define the participant's deployment duty. In this scenario, participant A is responsible for the deployment of the components App, Web Server and App VM with the target technology Kubernetes. On the right side, participant B is responsible for the deployment of DB, the DBMS, and the DB VM. A special annotation ${<component>.<property>} used in some components, references properties that are dependent on another component. For instance, the App can only connect to the DB, if the properties *hostname*, *user*, *port*, and *password* are provided by the DB. Since *hostname* is an output from the DB VM, the property is only provided after the DB VM is started and therefore has to be passed through, before Ansible and Kubernetes can be deployed or a connection established. Each participant is an independent organizational entity, e.g., different companies, and controls the deployment of the components, he/she is responsible for. Thus, the resulting deployment has to be executed by independent deployment engines and coordinated across the different participants.

Currently, the generation of the deployment order based on the EDMM model does not differentiate between different participants. The participant's region and the components and deployment technologies he/she is responsible for can not be annotated as of now. In addition, the deployment order is not based on workflow technologies and therefore does not provide the benefits that come with these technologies for the deployment of complex distributed applications. Additionally, since both participants are responsible for the deployment of the overall application, the generated workflows needs specific tasks including required properties for the data exchange between other participants. For example, participant A and its component App is dependent on the output properties of participant B's DB. Therefore, output properties have to be collected and sent to the other participants when required.

By extending the EDMM Transformation Framework and combining the presented approach of Wurster et al. [WBL+21] and Wild et al. [WBK+20], the aforementioned drawbacks are tackled in this work. Additionally, to demonstrate the feasibility of this work and the resulting implementation, the simple deployment scenario is used to highlight the main challenges of this work. Further, the upcoming chapters present concepts on how these challenges are tackled. To achieve the overall objective, the EDMM model has to be extended for the annotation of multiple participants and a concept created for the generation of workflows based on the participant's annotated EDMM model. Furthermore, the generated workflows have to manage the data exchange and communication for the deployment of different technologies with multiple participants.

3 Related Work

This chapter presents related work and discusses various work in the field of cloud deployment, workflow generation and approaches for a decentralized deployment.

3.1 Automated and Distributed Cloud Application Deployment

Guillén et al. [GMMC13] introduced a framework for developing cloud agnostic applications that may be deployed indifferently across multiple cloud platforms. This is enabled by separating all cloud related information from the source code, so the development process is therefore no longer conditioned by external requirements and constraints. So far, the deployment across multiple cloud platforms is done by a central orchestrator and the creation of certain workflows to enable the imperative approach is also not possible. Furthermore, for this approach to work, the source code of the application has to be provided and the programming language needs to be set to Java only.

Sebrechts et al. [SVW+18] present an approach in which a hierarchical collection of independent software agents, collectively managing the cloud application, is used and was done to introduce more abstraction to the topology-based cloud modeling languages. Due to the nature of the decentralized conversations, the communication and collaboration of these agents enable the management of cloud applications concurrently. As a result, the scalability of the solution is increased and alleviates the complex and error-prone process of manual configuration. However, it is not possible to define or generate a workflow for each agent that can be executed in a specific execution order.

Sandobalin et al. [SIA19] present a framework called ARGON, which is a model-driven infrastructure provisioning tool. ARGON includes a domain-specific language for the documentation and modeling of the characteristics of the cloud infrastructure and also provides transformation capabilities to automate the deployment for different cloud providers. As a result, the framework allows the automated deployment of an application in a distributed cloud environment. However, ARGON is limited to a central orchestrator for the coordination of the deployment and also lack the support of workflow technologies.

Pierantoni et al. [PKT+20] proposed a cloud technology-agnostic approach for the application description based on existing standards. This was done by extending the TOSCA policy hierarchy with several scalability and security policies. The resulting solution is a generic and pluggable framework which supports a secure automated deployment and orchestration of applications in the cloud. The automated deployment however does not take multiple participants into account.

3.2 Cloud Application Deployment Using Choreographies

Breitenbücher et al. [BBK+14] proposed a standards-based approach to generate provisioning plans based on declarative TOSCA topology models. The approach is divided in three parts and consists of (i) the generation of a *Provisioning Order Graph*, which is a graph version of the topology template, and then (ii) the translation of the provisioning order graph to the *Provisioning Plan Skeleton*. This skeleton defines an extended structure of the provisioning order graph and include empty provisioning activities. In the last step (iii) the empty provisioning activities are filled out by elements of a workflow language. For example, in BPMN, *Abstract Tasks* and in BPEL *Opaque Activities* are used. The result is an *Executable Provisioning Plan*, which is a fully automatically executable workflow that can be used for workflow engines and later be customized by the application developer after generation. For the execution of certain actions, implementation artifacts are used. Actions are for example scripts that are attached for the connection of a PHP application to the database. This approach however has only been implemented for TOSCA topology models so far.

Dukaric et al. [DJ18] introduced an approach using BPMN workflows for the automated cloud orchestration, which includes the automated arrangement, coordination, and management of complex cloud systems, middleware, and services [DJ18]. This was done by (i) defining a meta model for the modeling of cloud-specific workflows for the use in BPMN business process engines and (ii) extending the BPMN 2.0.2 specification[1] to orchestrate cloud-specific workflows. Furthermore, (iii) the meta model is implemented with the BPMN extensions, to show how cloud orchestration workflow elements, such as activities, map onto the extended BPMN elements. As a result, the overall complexity of the cloud orchestration can be reduced by using the proposed BPMN extension. However, this approach is limited to a central orchestrator which handles the execution of diverse technologies and therefore is not suitable for decentralized deployment. In addition, this approach requires to use the extended version of the BPMN 2.0.2 specification and thus requires further modification to run the specific workflows.

Further, Eilam et al. [EEKS11] introduced a model-based approach to bridge the gap between deployment models and workflows. The approach supports a separation of concerns where operation logic, such as scripts or workflows, are developed independently of the deployment model. As a result, developers are able to continue leverage useful libraries for the creation of scrips or workflows, while still benefitting from the deployment model, used for validation and constraint satisfaction. However, the current approach does not support multiple participants to take part in the deployment process.

Calcaterra et al. [CCDT17] developed an orchestrator capable of automating all the required tasks to deploy an application. The orchestrator takes a TOSCA service template as an input and transforms it into a BPMN workflow. The resulting workflow is deployable and can be used in an arbitrary workflow engine for the deployment execution. Similar to the presented approach by Breitenbücher et al. [BBK+14], only TOSCA is supported, which in practice is not widely used.

[1] https://www.omg.org/spec/BPMN/2.0.2

The presented approaches have shown that research in the fields of multi cloud technology deployments are ongoing and the orchestration is mostly focused on a central orchestrator. Further, most deployments are not workflow-based and therefore does not provide the benefits of workflow technologies. In the upcoming chapters, a concept for decentralized deployment is introduced which combines the declarative and imperative approach of deploying multiple different deployment technologies using workflow technologies.

4 Decentralized Orchestration and Deployment with Multiple Technologies

This chapter introduces the concept for a decentralized deployment using multiple different deployment automation technologies. The objective of this work is to fully automate the deployment of (i) different participants and their respective components with each participant (ii) using multiple deployment technologies.

As depicted in Figure 4.1, the concept is structured in eight steps. Since this work is an extension of the work of Wurster et al. [WBF+20], the steps which have been added or modified for the concept are shown as a dotted marked area. In the first step, a Global Deployment Model (GDM) is defined

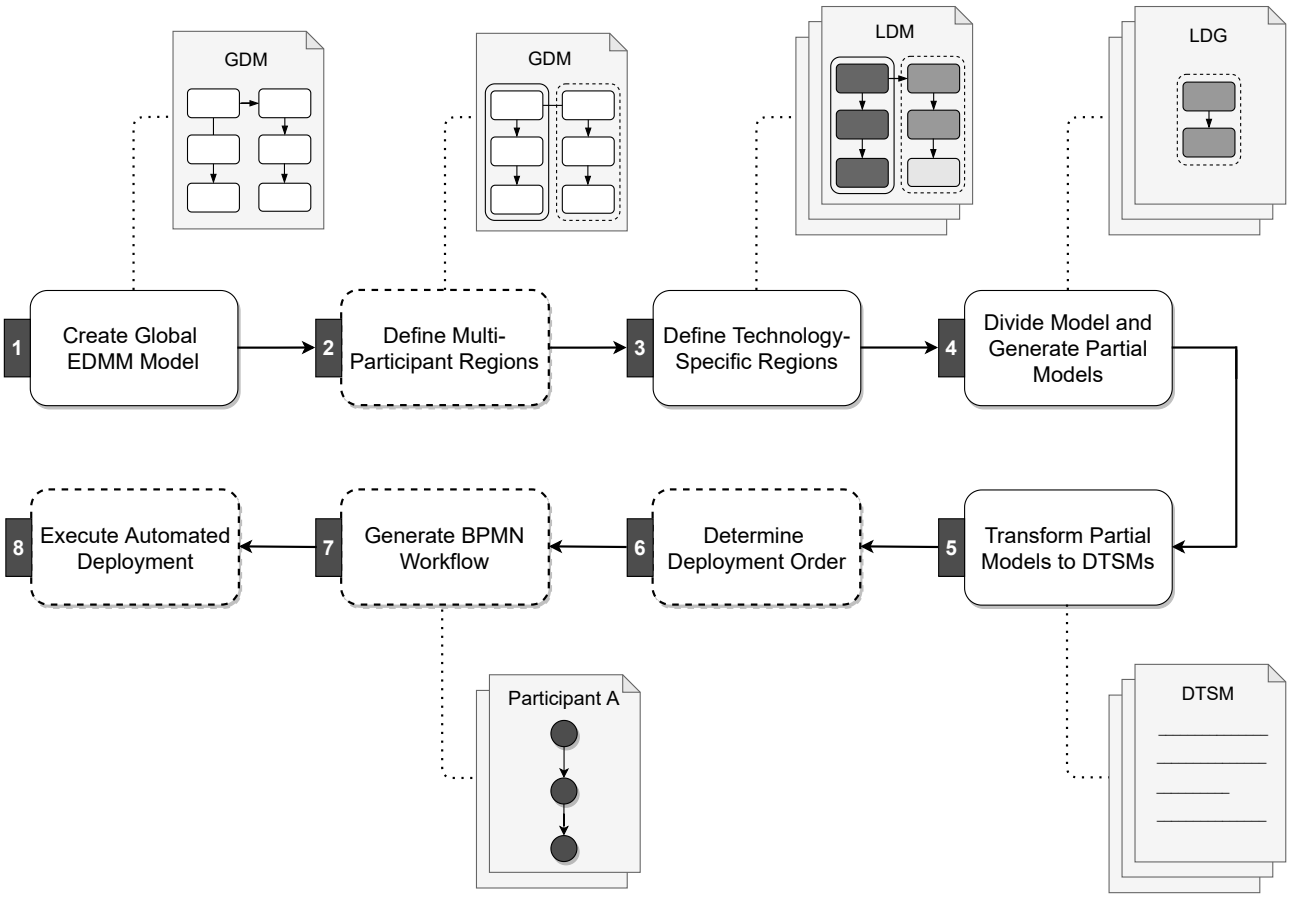

Figure 4.1: Concept for a decentralized deployment using multiple different deployment technologies and participants

that describes the structure of the whole application. This is done by creating an EDMM model, based on Wurster et al. [WBB+19]. Next, the created EDMM model is further annotated with multiple components and their respective participants that have to be deployed on the participant's side. After this step, the EDMM model is shared between all participants and their respective components are annotated with the technology-specific regions. Since each participant has its own model, these models are called Local Deployment Model (LDM). Furthermore, the resulting LDM is then divided into multiple partial deployment groups, called Local Deployment Group (LDG), and transformed in executable DTSMs. Further, the deployment order of the generated DTSMs is determined and based on this order, a BPMN workflow is generated. At last, the BPMN workflows are used for the workflow engine and execute the automated deployment. As a result, all workflows together form the deployment choreography. Each step is described in detail in the following sections.

4.1 Create Global EDMM model

As we have seen in Section 2.2, the most common deployment technologies in research and industry can be mapped into the EDMM, introduced by Wurster et al. [WBB+19]. As it is also possible to generate deployment-ready technology-specific files through the EDMM Transformation Framework, the EDMM paves a way to express deployments in a technology-agnostic way. In case of a technology change in the overall application deployment, the EDMM can be used and transformed into the desired target technology rather than manually adjusting the deployment files of the new deployment technology. Furthermore, a concept and prototype has already been implemented by Wurster et al. [WBL+21] for the deployment of different deployment technologies. For this reason, EDMM was chosen for the concept for a decentralized deployment using multiple different automation technologies.

With EDMM, we are able to define our overall deployment through the presented syntax. A model consists of components, which are defined by several properties, operations, and artifacts. The component types define the semantics of the components and relations define the interactions between components. Thus, we can define if something is *hosted on* by a component or a connection established *(connects to)*. In the following, a formal definition of an EDMM model defined by Wurster et al. [WBL+21] is presented:

$m = (C_m, R_m, CT_m, RT_m, type_m)$

where the elements are defined as follows:

- C_m : Set of Components in m, where each $c_i \in C_m$ represents a component
- R_m : Set. of Relations in m, where each $r_i = (c_s, c_t) \in R_m$ represents a relationship, in which c_s is the source and c_t the target component
- CT_m : Set of Component Types in m, where each $ct_i \in CT_m$ describes the semantics of the components assigned with this type
- RT_m : Set of Relation Types in m, where each $rt_i \in RT_m$ describes the semantics of the relations assigned with this type

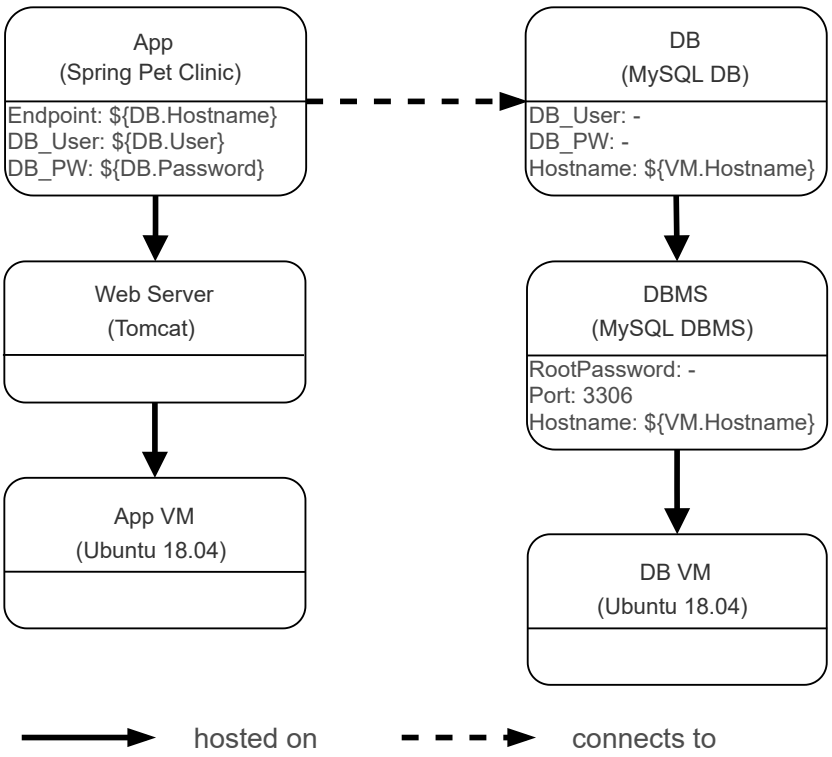

Figure 4.2: Simple EDMM model including components, relations, and property values

- $type_m$: Mapping which 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, and $type_m : ME_m \rightarrow MET_m$

This model is modified in the upcoming sections for the annotation of multiple participants and the annotation of different technology regions. But using the presented definition, an EDMM model shown in Figure 4.2, with multiple components and relations can be created. Since dependencies are important for the data exchange between other participants, the EDMM allows the definition of a global EDMM model, which is called Global Deployment Model (GDM), where all referenced property values are provided as globally visible information. For instance, the *DB* component is instantiated with the *MySQL DB* component type and properties are defined to the component type's specification to further configure the deployment. Furthermore, a special notation like ${<component>.<property>} is used to reference property values from related components. Since some of the property values like ${<DB>.<Hostname>} are only available during runtime, the presented notation has to be used to define references to property values that have yet to be set. At last, the global EDMM model with its components, relations, and types can be expressed as a YAML file defined by the EDMM YAML Specification[1].

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

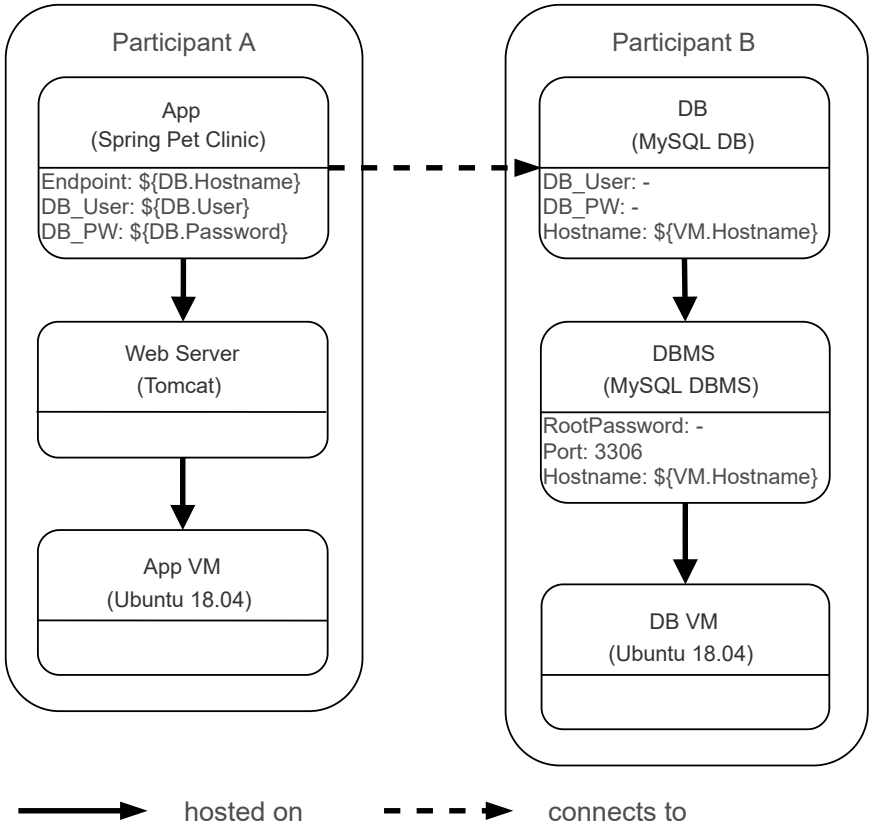

Figure 4.3: Deployment scenario with two participants annotated

4.2 Define Multi-Participant Regions

In this step, the created GDM will be annotated with participant-specific regions and indicates which component belongs to which participant and has to be deployed as such. Furthermore, components including property values which have a reference to another component that have yet to be set, e.g. DB_User shown in Figure 4.3, are annotated on the components of the respective participant. This allows further flexibility for the participants, since he/she does not have to provide a value upfront and can adjust these property values on their local EDMM model, called Local Deployment Model (LDM). With this approach, internal information, such as credentials, can be added later on and does not have to be shared upfront. As seen in Figure 4.3, a sample deployment scenario with two participants is shown and each participant responsible for the deployment of their participant region. Formally, the EDMM model is extended with the following definition of participants:

- ID_m : Model $multi\_id$ which defines a unique value
- P_m : Set of all available participants, where each $p_i \in P_m$ is a participant
- O_m : Owner o which this EDMM model belongs to, where $o \in P_m$
- $participants_m$: Mapping which assigns each component to a participant and endpoint with $p_i \in P : participants_m : C_m \rightarrow P_m$

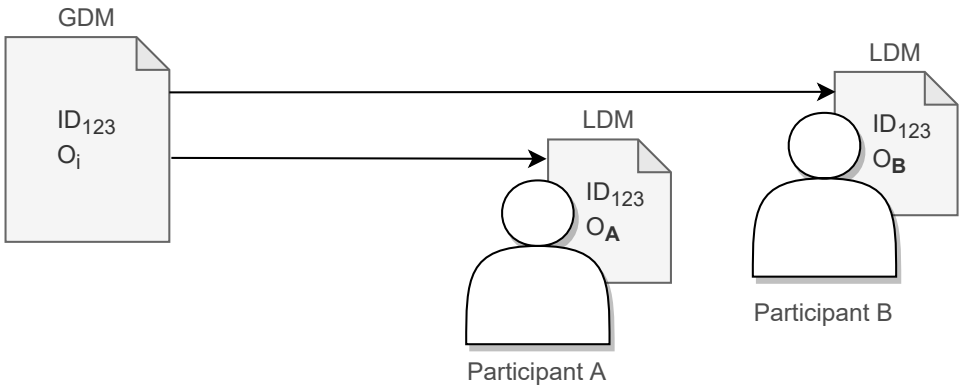

Figure 4.4: Shared EDMM model by multiple participants

Since the internal information has to be set by each participant on their own, a unique identifier is needed for the identification of the GDM. For this, the EDMM model has been extended with ID_m, called *multi_id* in the EDMM model. The ID_m is set, so that the GDM can be sent across multiple participants with each participant having their own version but the same unique identifier. As a result, participants define their internal information on their LDM without having to share their information with other participants.

For instance, as shown in Figure 4.4, participant A gets the GDM with the ID_{123} and sets property values for their LDM, whereas participant B also received a separate GDM with the same ID_{123} and sets their private information. Thus, each participant has its own version of the EDMM model. In addition, O_m in the LDM is defined to distinguish the different participants and set the owner of the LDM. Furthermore, with the mapping $participants_m$ all components are assigned a participant including their respective endpoints. As a result, the extended EDMM is now defined as follows: $m = (\mathbf{ID_m}, \mathbf{P_m}, \mathbf{O_m}, \mathbf{participants_m}, C_m, R_m, CT_m, RT_m, type_m)$ and includes all relevant information needed for the orchestration of different participants.

4.3 Define Technology-Specific Regions

In this step, all participants receive a copy of the annotated GDM from the previous step Section 4.2 and each participant can further modify the received model. Internal information that could not be made public due to privacy or compliance reasons can now be added to the LDM. As shown in Figure 4.5 in a sample deployment scenario, Participant B adds further information, such as DB_User, DB_PW, and $RootPassword$ to the components DB and $DBMS$. Additionally, Participant B annotates *technology_regions* introduced by Wurster et al. [WBL+21]. And thus, each component is assigned a technology region with the following formal definition:

- 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 with $t_i \in T$: $tech_m : C_m \rightarrow T_m$

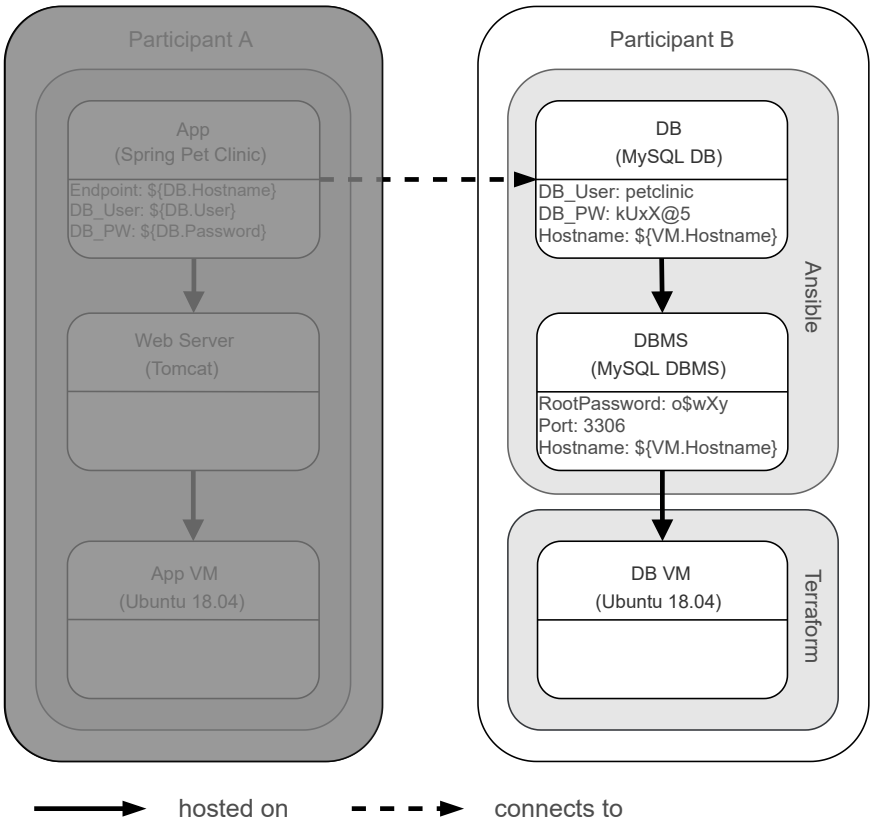

Figure 4.5: Deployment scenario with two deployment technologies used in the perspective of participant B

The resulting extended EDMM is now defined as follows: $m = (\mathbf{ID_m}, \mathbf{P_m}, \mathbf{O_m}, \mathbf{participants_m}, \mathbf{T_m}, \mathbf{tech_m}, C_m, R_m, CT_m, RT_m, type_m)$ and further adds the annotation of different technologies that have to be used for the overall deployment.

In Figure 4.5, the assignment of the technology regions are shown as such: Component DB and $DBMS$ are to be deployed with Ansible and hosted on the $DB\,VM$, which is deployed with Terraform. As a result of this approach, the participant B is only responsible for their own components and can focus on adding further information to their LDM. Moreover, participant B has no knowledge of the internal information or the technologies that are used for participant A. Furthermore, in a bigger deployment scenario, the need to add all property values is diminished and split between multiple participants. As an example, a GDM consisting of public information could be created and then later on modified as a LDM on the participant's part. Therefore, the overhead of preliminary work is reduced drastically.

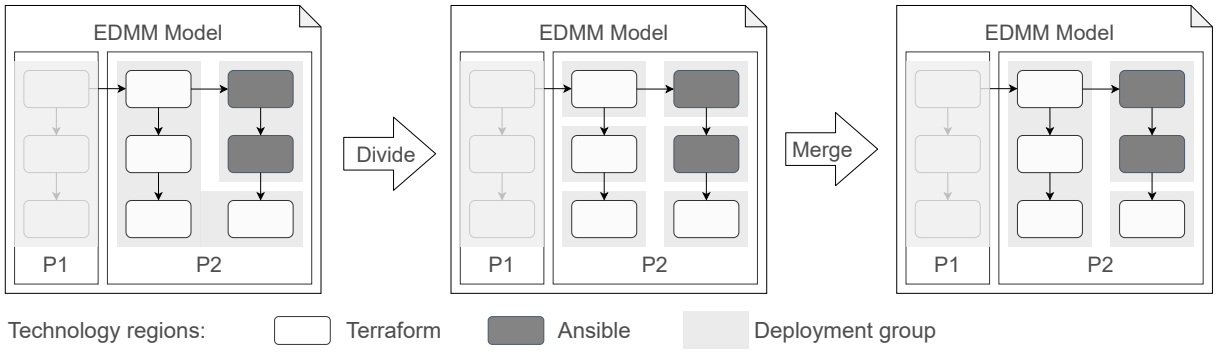

Figure 4.6: Algorithm to divide and merge the EDMM model to generate partial models, based on [WBL+21]

4.4 Divide Model and Generate Partial Models

For this step, the algorithm introduced by Saatkamp et al. [SBKL20] and extended by Wurster et al. [WBL+21] is used to divide the model further and generate partial deployment technology groups, called Local Deployment Group (LDG). The motivation here is to group technologies that can be deployed by a single run of the underlying deployment technology. This step has already been implemented by Wurster et al. [WBL+21] and has not been further modified in this work.

An abstracted EDMM model example shown in Figure 4.6, demonstrates the grouping of all deployment technologies on the far left side. The EDMM model consists of two participants and shows the annotated model of P2. Therefore, we have no knowledge of the components of P1 in terms of what deployment technology is used and thus is marked as *Undefined*. With the annotated deployment technologies of P2, two deployment groups are defined with Terraform and Ansible. If we consider the deployment groups as self-contained entities, we can determine a cyclic-based graph and due to the cyclic dependencies, the deployment cannot be executed. The reason for this, is because the deployment group Terraform can only be deployed after the deployment group Ansible is deployed, but Ansible requires a Terraform component to be deployed first. To tackle this issue, all components are divided into individual deployment groups and then merged until no more deployment groups can be merged without resulting in cyclic dependencies. The algorithm presented by Saatkamp et al. [SBKL20] has been extended by Wurster et al. [WBL+21] and defines the following rules: Two components can be deployed if (i) the same technology is assigned and (ii) merging the components does not result in cyclic dependencies. The result is a graph that contains all deployment technologies in a technology region that can be deployed together.

As an example, illustrated in Figure 4.6 on the far right side, an EDMM model is shown that has been first divided and then merged to remove cyclic dependencies. For the generation of workflows and technology-specific models, the presented approach by Wurster et al. [WBL+21] is served as a basis for the upcoming sections.

4.5 Transform Partial Models to DTSMs

With the resulting LDGs in Section 4.4, multiple Deployment Technology-Specific Models (DTSM)s are created. Each LDG that has been identified by the algorithm and is part of the respective participant is turned into a DTSM. This process is done by the EDMM Transformation Framework by Wurster et al. [WBF+20] and each LDG is used for the EDMM Transformation Framework for the automated generation of artifacts, files, etc. for the desired target technology. However, the transformation has to be aware of the dependencies of the deployment groups. For instance, shown in Figure 4.3, because of different participants, the component *App* does not have any knowledge of the deployment of component *DB*, but still needs the property values *Endpoint*, *DB_User*, and *DB_PW* sent before the component *App* can be deployed. Therefore, a special annotation ${<component>.<property>} is used to define property values that are dependent on another component and have yet to be created during runtime. Further, the generation of DTSMs by the EDMM Transformation Framework has not been modified for this work and used as a basis for the deployment execution.

4.6 Determine Deployment Order

In this step, an approach to determine the deployment order with multiple participants and different technologies is presented. The deployment order is an essential part of the overall deployment, since it has to keep track of the dependencies between several deployment groups as well as the different participants. One participant might need property values of another participant to deploy a specific deployment group. As a result, a data exchange mechanism through several deployment steps and a means for participant communication has to be introduced. From the example application presented in Figure 4.5, the algorithm introduced by Saatkamp et al. [SBKL20] and adapted by Wurster et al. [WBL+21] would determine the following order shown in Figure 4.7: *DB VM* is deployed first with Terraform, then *DB* and *DBMS* are deployed with Ansible and at last *App*, *Web Server*, and *App VM* is an external deployment group and deployed by participant A. The result is a provisioning

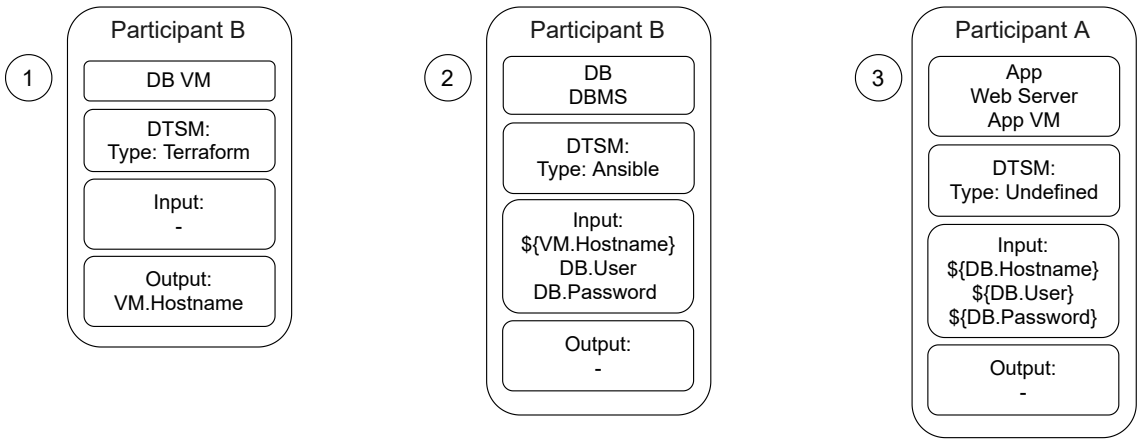

Figure 4.7: Generated deployment order including affected components, target technology, and necessary input property values in the perspective of participant B

order graph, as defined by Wurster et al. [WBL+21], which contains the deployment groups as well as their deployment order. Furthermore, since the last three components are to be deployed by another participant, participant B has no knowledge of the underlying deployment order and technology that has to be used. The only relation between participant A and participant B are the components App and DB. With this relation, we can derive that DB has to deployed first before App can be deployed. In addition, App requires an $Endpoint$, DB_User, and DB_PW credentials by the other participant to start the deployment. Therefore, only LDGs and the deployment order of the components within the participant's responsibility are relevant and only dependencies between other participants have to be considered in the workflow.

For this step, the deployment order generation has been modified and takes different participants into account. External deployment groups which are not in the participant's responsibility are filtered out in the algorithm and the deployment order generation is used only on components which belong to the respective participant. As a result, for each participant a local deployment order is generated, which only he/she has knowledge of the deployment order. In addition, for the cross participant data exchange, dependencies, such as input, output, and relation, stay the same for each participant.

4.7 Generate BPMN Workflow

This step introduces the generation of a participant-specific BPMN workflow based on the generated deployment order of Section 4.6. For the cross-participant application deployment, data exchange between participants is required before the components are deployed and the connection can be established. Additionally, independent on which participant initiates the deployment, all other participants have to deploy their respective parts as well. A similar issue was tackled by Wild et al. [WBK+20] and include activities, such as *Send*, *Receive*, and *Initiate* to handle the orchestration of multiple participants. For this work, the approach presented by Wild et al. [WBK+20] is adapted and the following definition of participant-specific workflow models is presented:

For each participant $p_i \in P$ a participant-specific workflow model w_i is defined as:

$w_i = (A_{w_i}, E_{w_i}, V_{w_i}, i_{w_i}, o_{w_i}, T_{w_i}, tech_{w_i}, type_{w_i}, D_{w_i}, group_{w_i})$

where the elements are defined as follows:

- A_{w_i} : Set of activities in w_i with $a_y \in A_{w_i}$
- $E_{w_i} \subseteq A_{w_i} \times A_{w_i}$: A set of control connectors between activities, where each $e_y = (a_s, a_t) \in E_{w_i}$ defines that activity a_s has to be deployed before a_t can start
- $V_{w_i} \subseteq \wp(\sum^+) \times (\sum^+)$: Set of data elements, where $\sum^+$ is the set of characters in the ASCII table
- i_{w_i} : Mapping, which assigns to each activity $a_y \in A_{w_i}$ with its input parameters $i_{w_i} \rightarrow \wp(V_{w_i})$
- o_{w_i} : Mapping, which assigns to each activity $a_y \in A_{w_i}$ with its output parameters $o_{w_i} \rightarrow \wp(V_{w_i})$
- T_{w_i}: Set of all supported technologies, where $t_y \in T_{w_i}$ is a deployment technology

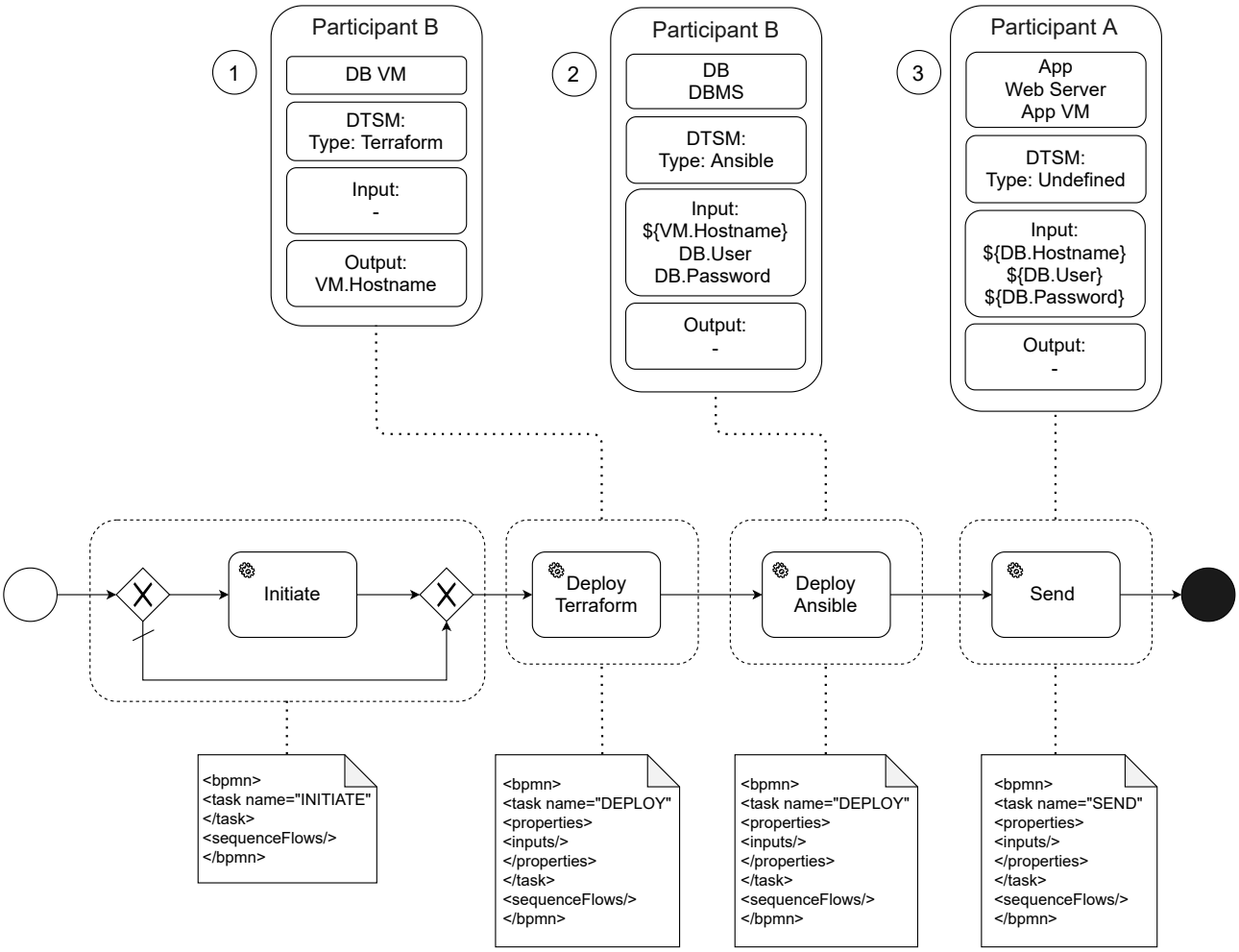

Figure 4.8: Generated workflow in BPMN for participant B

- $tech_{w_i}$: Mapping, which assigns to each activity $a_y \in A_{w_i}$ with its target deployment technology $t_y : A_{w_i} \rightarrow T_{w_i}$
- $type_{w_i}$: Mapping, which assigns each activity $a_y \in A_{w_i}$ to an activity type $A_{w_i} \rightarrow \{receive, send, initiate, deploy\}$
- D_{w_i} : Set of all deployment groups, where $d_y \in D_{w_i}$ is a deployment group
- $group_{w_i}$: Mapping, which assigns to each activity $a_y \in A_{w_i}$ its deployment group $d_y : A_{w_i} \rightarrow D_{w_i}$

Since w_i consists of all relevant information to deploy technology groups and information exchange between participants, we can retrieve the data and map them into a BPMN skeleton. Each task type is assigned a specific template to map the respective task to a BPMN activity. For instance, Figure 4.8 shows the mapping of deployment groups to specific BPMN task templates. After all deployment groups are mapped, a *Start Event*, an *Initiate* BPMN task, and an *End Event* is added to the workflow. The Initiate BPMN task determines whether an Initiate message has to be sent or skipped. An Initiate is skipped by the alternative sequence flow, when another participant has

Listing 4.1 BPMN Deploy

```
<bpmn:serviceTask id="Deploy_1" name="Deploy Terraform">
  <properties>
    <property name="participant" value="http://localhost:5000" />
    <property name="component" value="ubuntu_db" />
  </properties>
  <bpmn:incoming>Flow_1</bpmn:incoming>
  <bpmn:outgoing>Flow_2</bpmn:outgoing>
</bpmn:serviceTask>
```

Listing 4.2 BPMN Send

```
<bpmn:serviceTask id="Send_1" name="Send">
  <properties>
    <property name="participant" value="http://localhost:7000" />
    <property name="input" value="db_hostname" />
    <property name="input" value="db_user" />
    <property name="input" value="db_password" />
  </properties>
  <bpmn:incoming>Flow_3</bpmn:incoming>
  <bpmn:outgoing>Flow_End</bpmn:outgoing>
</bpmn:serviceTask>
```

already started the deployment. In addition, the sequence flows are automatically mapped to the incoming and outgoing activities, so that additional activities can be simply connected by another task template.

Listing 4.1 illustrates the generated task definition of a *Deploy* task based on Terraform and its component *DB VM*. A service task, which is linked to a custom implementation is generated and maps the input property values to the respective BPMN property elements. Line 3 defines a BPMN property with the endpoint of the participant and Line 4 defines the component *ubuntu_db* which has to be deployed. Furthermore, Lines 6-7 define sequence flows that connect the neighboring activities.

In another example, shown in Listing 4.2, a Send task including the components *App*, *Web Server*, and *App VM* is mapped into a BPMN activity. Line 3 defines the endpoint for the data exchange with the other participant and Line 4-6 define the property values that have to be sent. The extended syntax *db_hostname* specify that the property value *hostname* of the *db* has to be used as an input and sent to the participant shown in Line 3. At last, Line 8-9 define sequence flows that connect the neighboring BPMN activities.

Furthermore, in cases where components require property values from more than one component, as shown in Figure 4.9, a parallel gateway is created and linked to the other *Receive* tasks. For example, the component *App* can only be deployed when property values from the components *DB*, *Web Server*, and *Queue* are received. This approach is also used in cases when a parallel deployment is necessary and the parallel gateway would be linked to the other *Deploy* tasks.

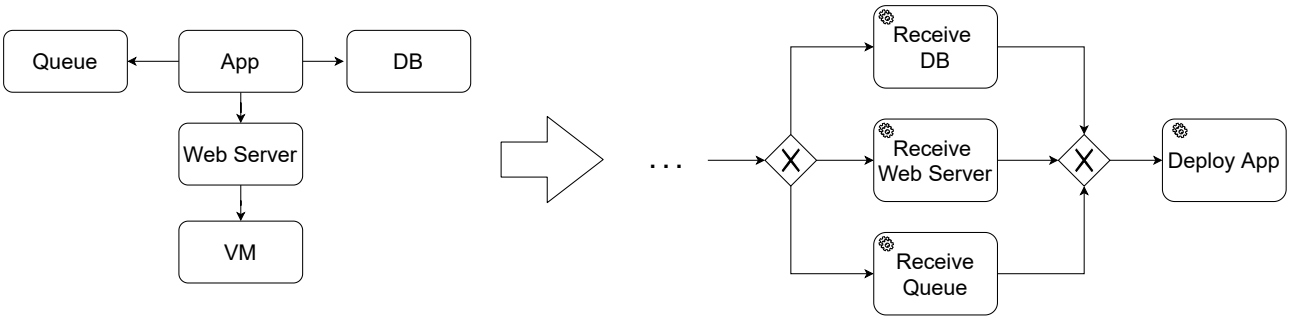

Figure 4.9: Sample Multi Receive BPMN activity based on an EDMM model

Algorithm 4.1 *GenerateExecutableProvisioningPlan*(p_i, *provisioning_order_graph*$_i$)

```
1:  let w_i ← null
2:  w_i add type_initiate
3:  for group_i in provisioning_order_graph_i do
4:      if group_i belongsTo p_i then
5:          w_i add (group_i, type_deploy)
6:      else
7:          let task_type_list ← []
8:          task_type_list ← determineTaskType(p_i, group_{i−1}, group_i, group_{i+1})
9:          for task_type in task_type_list do
10:             w_i add (group_i, task_type)
11:         end for
12:     end if
13: end for
14: w_i add type_end
```

Algorithm 4.2 *DetermineTaskType*(p_i, $group_{i-1}$, $group_i$, $group_{i+1}$)

```
1:  let task_type_list ← []
2:  for c1 in group_i do
3:      if group_{i−1} belongsTo p_i then
4:          for c2 in group_{i−1} do
5:              if (c1 isRelatedTo c2) then
6:                  task_type_list add type_send
7:              end if
8:          end for
9:      end if
10:     if group_{i+1} belongsTo p_i then
11:         for c3 in group_{i+1} do
12:             if (c1 isRelatedTo c3) then
13:                 task_type_list add type_receive
14:             end if
15:         end for
16:     end if
17: end for
18: return task_type_list
```

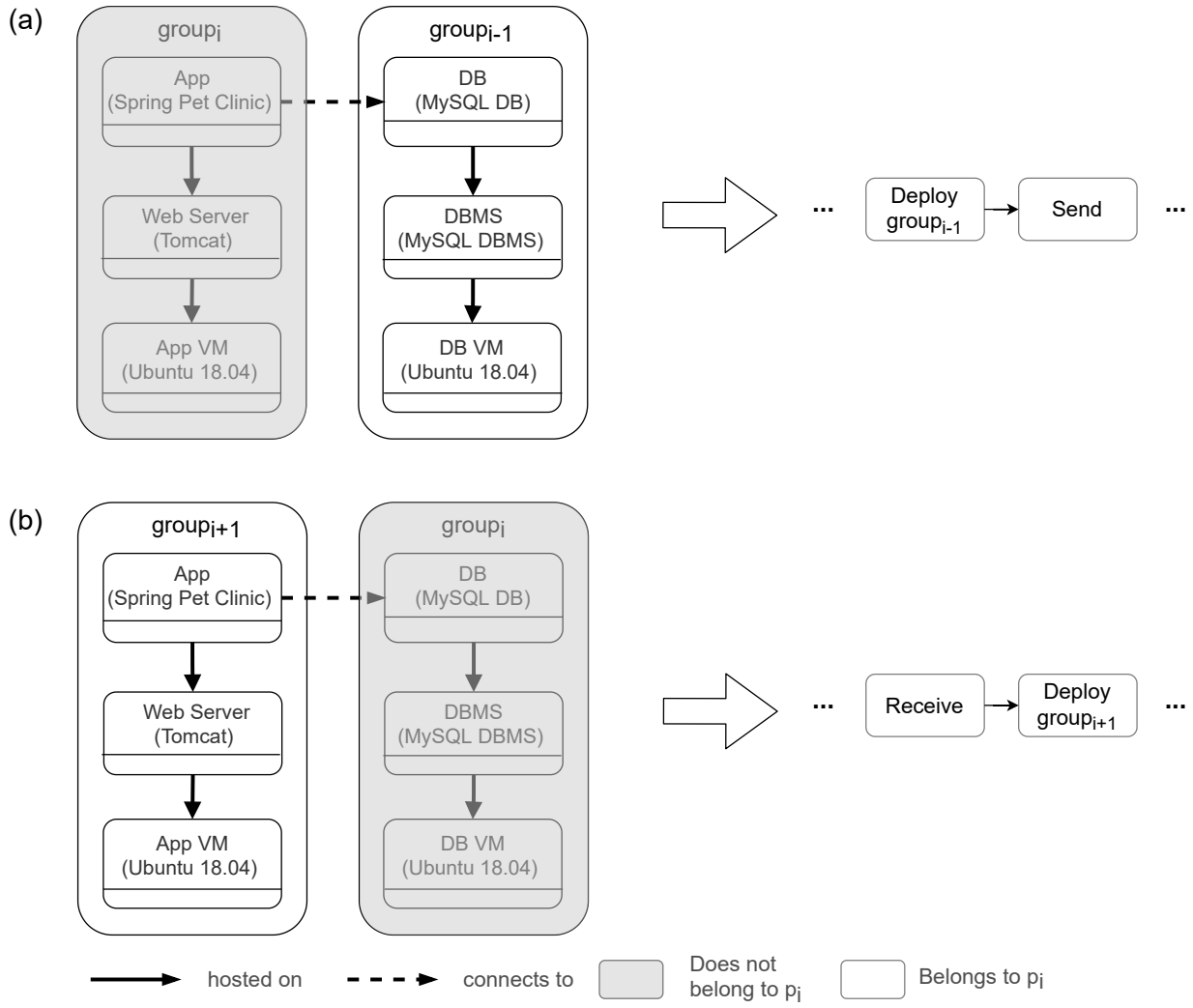

Figure 4.10: Generated Task Type using the $determineTaskType$ algorithm

To determine when a *Send*, *Receive*, or *Deploy* task must be applied, the algorithm shown in Algorithm 4.1 is defined to generate a participant-specific workflow w_i. It takes the participant p_i and $provisioning\_order\_graph_i$ as an input. Line 1 defines an empty workflow w_i, which is initially added with an *Initiate* task in Line 2. The Initiate task includes all participants that have to be notified, when the deployment is started. In Line 3, all deployment groups $group_i$ are iteratively extracted from the $provisioning\_order\_graph_i$ and compared in Line 4 if the $group_i$ belongs to the respective participant p_i. If that is the case, $group_i$ including the $type_{deploy}$ is added to the workflow w_i. Since $group_i$ belongs to the participant p_i, the specific deployment technology and relevant property values are known, therefore it can automatically be classified as a Deploy task. However, if $group_i$ does not belong to p_i, then the *else* block in Line 6 is used. Line 7 defines an empty $task\_type\_list$, in which all determined task types (receive, send) are added to the list (Line 8). For each $task\_type$ in the $task\_type\_list$, a new task is added to the workflow w_i. This is due to some cases, as shown in Figure 4.9, where multiple tasks are necessary. At the end, Line 14 adds a $type_{end}$ end event to the workflow w_i.

For determining the correct task type, the method $determineTaskType$ takes the participant p_i, the previous deployment group $group_{i-1}$, the current deployment group $group_i$ (which does not belong to the owner), and the next deployment group $group_{i+1}$ as an input. In Line 1, an empty list $task\_type\_list$ is defined. To determine whether a Send task can be added to the list, Line 2 extracts all components $c1$ iteratively of the deployment group $group_i$ and compares them with components of the previous deployment group $group_{i-1}$ (Line 3-5). If $group_{i-1}$ belongs to p_i, and a related component between $c1$ and $c2$ is found, e.g. as shown in Figure 4.10 in (a) the component $c1$ App of $group_i$ has an outgoing edge to the component $c2$ DB of $group_{i-1}$, then a Send task is created from DB to the App. In Line 6, the $type_{send}$ task is then added to the $task\_type\_list$.

In contrast, Line 10-12 compares the components $c1$ of $group_i$ with the components $c3$ of the next deployment group $group_{i+1}$. Here, the relation between $c1$ and $c3$ is checked (Line 12), for instance, as illustrated in Figure 4.10 in (b) the component App of $group_{i+1}$ has an outgoing edge to the component DB $group_i$, therefore a Receive task is created for the component App. In Line 13, the $type_{receive}$ is added to the list (Line 13). At the end, the $task\_type\_list$ is returned with all added task types for the deployment group $group_i$ (Line 18).

Finally, after all tasks are mapped, the resulting participant-specific BPMN workflow contains all information defined by w_i. Figure 4.8 illustrates the overall process of the BPMN generation. To summarize, the deployment groups shown on top are successively transformed to the respective BPMN templates at the bottom and the property values are mapped to the BPMN syntax. The resulting BPMN file is then used for a workflow engine for the automated deployment execution.

4.8 Execute Automated Deployment

For the execution of generated DTSMs, technology-specific implementation is needed to automatically execute the deployment of the overall application. Since nearly all deployment technologies provide different options to interact with, e.g. CLIs, APIs or even SDKs, the technology-specific implementation has to be extended so that the deployment can be executed with the provided input of the generated BPMN tasks. The deployment execution can be started by any participant. Since all generated participant-specific workflows include an *Initiate* task, other participants are automatically notified that the deployment has started. As seen in Figure 4.8, the Initiate task includes a parallel gateway, which determines whether the deployment has been started by the owner or another participant. If the deployment has been started by the owner, the Initiate task implementation notifies the other participants and immediately continues with their next task. On the other side, if the deployment has been initiated by another participant, the Initiate task is not triggered but skipped. This reduces the amount of calls that are necessary to communicate with other participants. To deploy the remaining components, multiple Deploy, Receive, and Send tasks are consecutively executed. As a result, we use the BPMN workflow to orchestrate the deployment of technologies and also for the communication exchange between different participants.

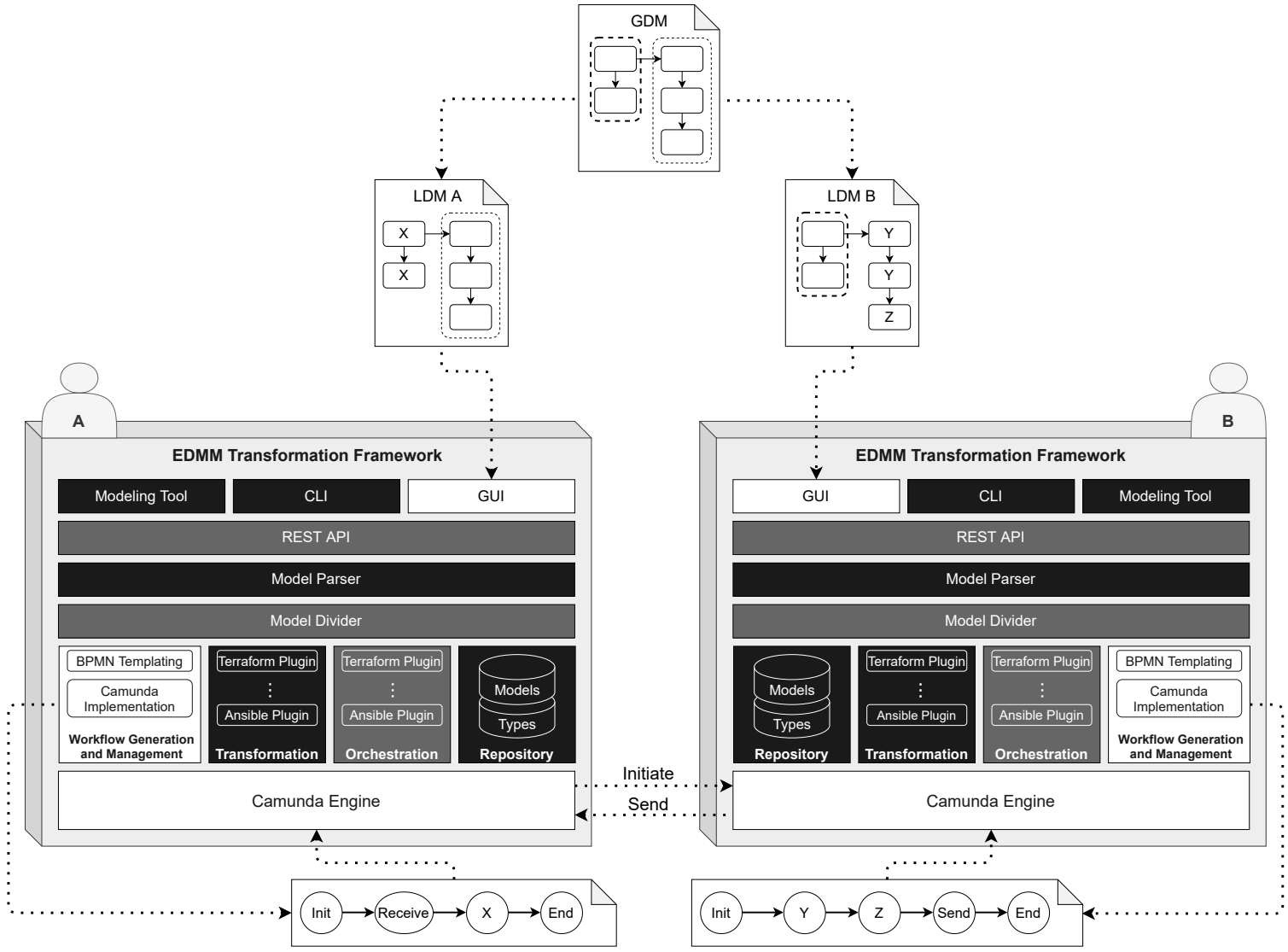

Figure 4.11: System architecture of the extended EDMM Transformation Framework with new components in white and modified components in gray.

4.9 System Architecture

This section introduces the system architecture of the modified and new components of the EDMM Modeling and Transformation Framework [WBF+20]. In Figure 4.11, a modular system architecture for two participants is illustrated with new components in white and modified components in gray. The *Modeling Tool* is a web-based modeling environment that allows to graphically compose the structure of the overall EDMM model. With the *CLI*, it is possible to *divide* and *transform* the EDMM model into multiple DTSMs and execute the automated deployment using multiple deployment technologies. To coordinate the deployment of different participants, a *Graphical User Interface (GUI)* has been added. This allows the automated generation of BPMN workflows through a simple user interface and manages the communication of the generated files with the integrated

Camunda Engine. Further, the deployment can be started and monitored until the deployment of the application is finished. The *REST API* is used to retrieve and update data in the *Repository* and has been further modified for the automated deployment. Components can be deployed by calling an endpoint with the respective input property values. For the translation of the EDMM model into an internal graph-based data structure, the *Model Parser* is used. With the graph-based data structure, the *Model Divider* performs an algorithm to generate different deployment groups. This component has been modified and only generates participant-specific deployment groups, which only take the owner's components into account and marks the components of other participants as external dependencies. The *Transformation* component is responsible for the transformation of EDMM model fragments into DTSMs. It uses a plugin approach and allows the integration of different deployment technologies in an extensible way. Each plugin contains logic and transformation for the creation of technology-specific directory structures, files, and artifacts. The *Orchestration* component is used to determine the deployment order and invoke the respective deployment plugins for an automated deployment. The execution is done by calling different CLIs, APIs, or even SDKs. For the deployment of multiple participants, it is necessary to return output property values and runtime information for the other participants. Therefore, the *Orchestration* component is coupled with the *REST API* and returns necessary data exchange information after a successful plugin run. The *Worfklow Generation and Management* component manages the generation of BPMN files by extracting the information generated by the deployment order and maps them to the respective BPMN template files. Here, Initiate, Receive, Send, or Deploy tasks are automatically generated for the participant. In addition, Camunda-specific implementation for the tasks are handled here and prepares them for the execution by the *Camunda Engine*. The data exchange is coordinated by the *Camunda Engine* and serves as an endpoint between the other participants. Furthermore, the *Camunda Engine* is also responsible for the execution of the generated BPMN workflows and handles the population and storage of property values and runtime information through the current workflow execution.

For the overall application deployment with two participants: In Figure 4.11, the GDM is annotated with different participants and annotated with necessary dependencies for the data exchange. Then, participant A and participant B annotate their participant-specific technology regions in the LDM and fill out property values that were not provided upfront. The EDMM Transformation Framework is then provided with the annotated LDM and transforms them into LDGs and their respective DTSMs. Moreover, based on the LDM, a participant-specific BPMN workflow is generated. Through the GUI, the BPMN workflows are automatically uploaded to the engine and either participant A or participant B can initiate the deployment. Upon initiation, the *Camunda Engine* starts the deployment and coordinates the data exchange with other participants.

4.10 Discussion and Limitations

The presented approach enables the creation of participant-specific BPMN workflows that can handle the orchestration of multiple participants as well as exchange necessary information for the deployment of the overall application. However, the orchestration was done by applying custom *Send*, *Receive*, *Deploy*, and *Initiate* implementation rather than the choreography notation provided by BPMN 2.0. The choreography notation allows the creation of choreography diagrams and focuses on between-processes interactions and message flows between multiple participants. This is done by creating choreography tasks with a specific business use case and an extension on how

individual processes interact with other participants. For instance, a choreography task is extended with individual processes on how to interact with other participants, such as the *Manufacturer* or *Customer*. The implementation of the choreography task then determines on how the message is to be sent, e.g. REST API, E-Mail, etc. The approach of this work is similar to the choreography diagram provided by BPMN 2.0 but due to the limited support of this feature of most BPMN workflow engines, the implementation of choreographies was done by service tasks which include special implementation on how the other participant can be reached. In addition, for the choreography diagram to work, a choreography model can be created upfront and the workflows for each participant modified later on. However, this would lead to an overhead, since all relevant information for the data exchange can be retrieved from the declarative deployment model.

Furthermore, this process still includes a few steps that have to be done manually and could be improved for the future. Since the distribution of the participant's EDMM model has to be done manually, another process could be integrated to automatically send out the respective model. Moreover, the workflow engine has to be sent the BPMN workflow manually and the user therefore has to interact with the workflow engine. To tackle this issue, this process could be integrated in a Continuous Integration (CI) environment or even integrated in a Dashboard, where all calls are orchestrated from there.

Lastly, the presented approach allows the modification of the LDM of the participant by annotating the components with the respective deployment technologies and modify property values of referenced components. It is possible to add further components or replace them with other components for the deployment, e.g. the component *App* is additionally hosted on a *Tomcat Web Server*, which is hosted on a *VM*. The annotation of these components does not have to be done upfront and can later be added on in the EDMM model. However, this is only possible when external dependencies do not rely on this component. If, for instance, a new component is added with additional property values and is reliant on dependencies by another participant, then the deployment of this specific component can not be done. This is due to the external dependencies that all stay the same through all participants in the GDM. Therefore, for components that are added and are reliant on another component by another participant, the GDM has to be updated first. For the modification of pre-defined components by the GDM in the LDM, the GDM does not have to be updated.

5 Prototypical Implementation and Case Study

In this chapter, to prove the feasibility of the presented approach, a prototypical implementation based on the EDMM Transformation Framework is presented. With the work of this thesis, the EDMM Transformation Framework is extended to enable the deployment of different technologies through BPMN workflows as well as the orchestration of multiple participants. Functionalities, such as the transformation of an EDMM model to DTSMs or the execution of the desired deployment technologies have already been implemented by the work of Wurster et al. [WBB+19][WBL+21]. However, the deployment is not workflow-based and therefore restricts orchestration capabilities. In addition, due to the central orchestrator used for the execution, the EDMM Transformation Framework has to be extended to enable a decentralized deployment approach. In the following, the implementation and the extension of the EDMM Transformation Framework is explained in detail.

Furthermore, to validate the prototypical implementation, the scenario presented in Figure 2.6 is used to demonstrate the extended functionalities in the EDMM Transformation Framework. The results are presented in a case study shown in Section 5.3. The created EDMM model of participant B for the case study are attached and are shown without the definitions of relations and component types in Appendix A. In addition, the automatically generated BPMN workflow of participant B is attached as well.

5.1 EDMM Model Specification

For the prototypical implementation, the EDMM model has been extended with the concepts presented in Section 4.2. Components, relations, necessary property values, and their artifacts are represented by a YAML format and shown in Listing 5.1. Line 2 has been added to the model and defines a unique identifier for the EDMM model. The identifier is used so that participants are able to annotate their LDM and still have a reference to the GDM. Moreover, Line 3 defines a new property *owner* and references the current LDM to a specific participant e.g. *partnerB* is the owner of the LDM and defines the components that have been assigned to him. The Lines 5-10 have been added by Wurster et al. [WBL+21] and defines components and the desired deployment technology. This part has been modified in this work for the orchestration of multiple participants and only components that have been assigned to the *owner* have to be annotated. Line 12-24 has been added anew and defines all participants that participate in the deployment. Each participant is assigned an endpoint in which the participant and their workflow engine can be reached. At last, the components are assigned to the respective participants.

For the definition of components and their relationships, a sample component is shown in Listing 5.2 and defines a *Spring PetClinic Application*. Line 4-12 define the type of the application, the necessary artifacts for the deployments, and the scripts that have to be run. Additionally, the

Listing 5.1 Extended EDMM Model with three deployment technologies and two participants

```
version: edm_1_0
multi_id: 12345
owner: partnerB

orchestration_technology:
    terraform:
        - ubuntu_db
    ansible:
        - dbms
        - db

participants:
  partnerB:
    endpoint: http://localhost:5000
    components:
      - ubuntu_db
      - db
      - dbms
  partnerA:
    endpoint: http://localhost:6000
    components:
      - ubuntu_app
      - pet_clinic
      - pet_clinic_tomcat
```

Listing 5.2 Sample component of an EDMM model

```
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}
```

relations between the Spring PetClinic Application and other components are defined. For instance, the Spring PetClinic Application has to be hosted on the *pet_clinic_tomcat* and connected to the *db*. In Line 13-17, needed property values for the deployment of the component are specified. Here, property values that are reliant on another component and have to be referenced are marked with the introduced syntax ${component.value}.

With the mentioned additions and modifications, the extended EDMM model is able to define multiple participants, their respective components and input property values as well as the necessary identifiers for the distribution of the EDMM model to multiple participants.

5.2 Overview Extended Transformation Framework

In this section, the implementation and the extension of the EDMM Transformation Framework is presented step by step. In addition, components that are important for this work are explained in more technical detail.

UML Class Diagram

Listing 5.3 Deploy REST JSON Body

```
{
    "modelId": "12345",
    "correlationId": "12014092-6586-11eb-ae93-0242ac130002",
    "components": [
        "db", "dbms"
    ],
    "inputs": [
        {
            "component": "ubuntu_db",
            "properties": {
                "hostname": "193.196.54.56"
            }
        }
    ]
}
```

The implementation is based on Java[1] the programming language and the Spring Boot Framework[2] for the creation of standalone web applications. In Figure 5.1, a UML class diagram is illustrated and shows a simplified version of the extended classes in the EDMM Transformation Framework. Since some features have already been implemented by Wurster et al [WBL+21], gray fields mark the classes which have been extended or implemented anew. At the top level, *executeTechnology()* has been added as a new method and allows the execution of the implemented technologies through a REST call. Previously, the execution of a specific technology could only be done via the CLI

[1] https://www.java.com/de/
[2] https://spring.io/projects/spring-boot

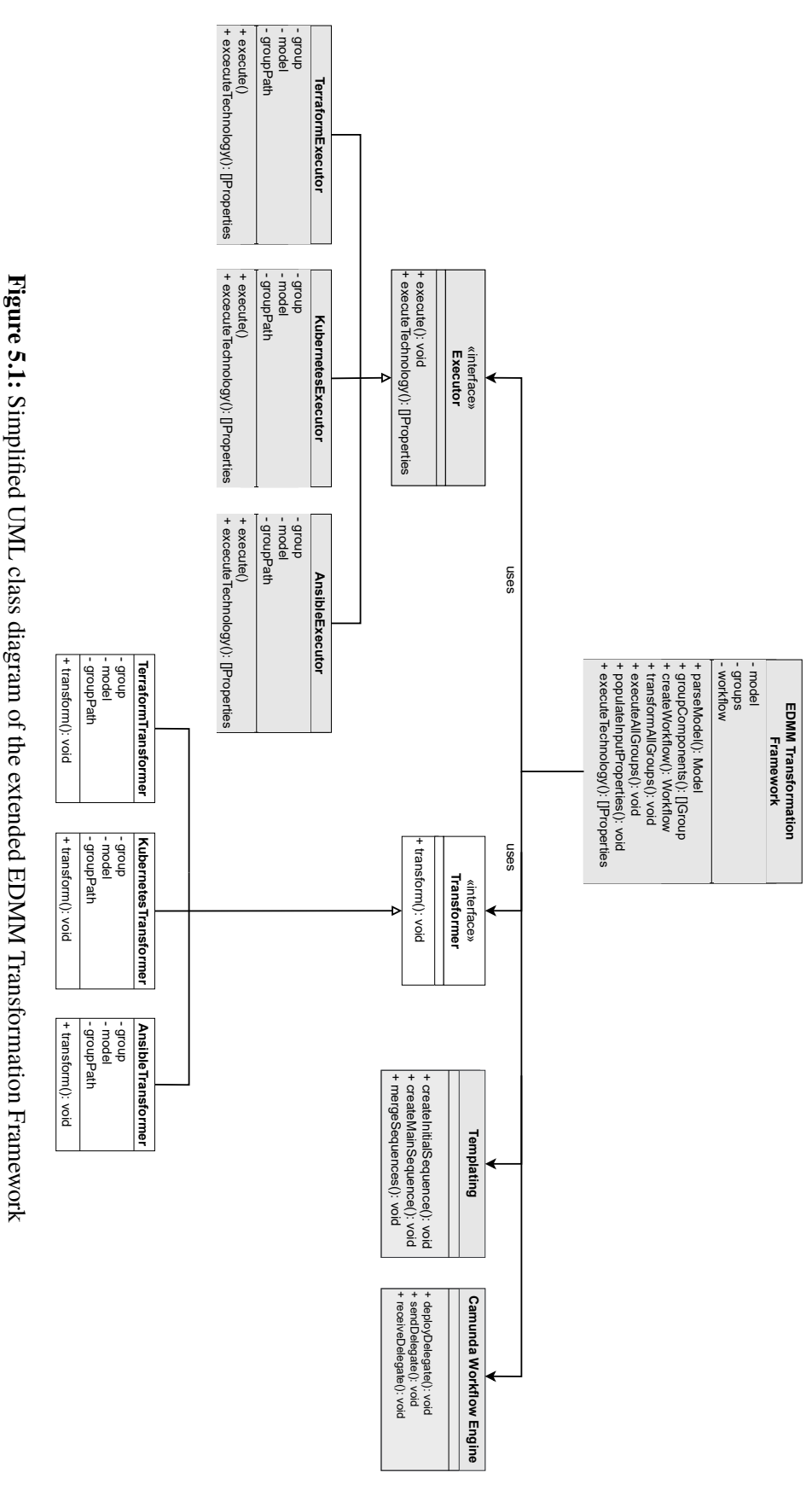

Figure 5.1: Simplified UML class diagram of the extended EDMM Transformation Framework

Listing 5.4 Deploy BPMN Template

```
<#if deployTasks?has_content>
<#list deployTasks as dt>
<bpmn:serviceTask id="ID_${dt.step}" name="DEPLOY ${dt.tech}" camunda:delegateExpression="${r"
${deployDelegate}"}">
  <bpmn:extensionElements>
    <camunda:properties>
      <#list owner as o>
      <camunda:property name="participant" value="${o}" />
      </#list>
      <#list dt.components as component>
      <camunda:property name="component" value="${component.name}" />
      </#list>
    </camunda:properties>
  </bpmn:extensionElements>
  <bpmn:incoming>Flow_${dt.step}</bpmn:incoming>
  <bpmn:outgoing>Flow_${dt.step + 1}</bpmn:outgoing>
</bpmn:serviceTask>
</#list>
```

and as a result limited the access and execution options. With an extension, the deployment of a technology can be triggered as shown in Listing 5.3 and requires the *modelId* (Line 2) as a reference to the EDMM model. Furthermore, the *correlationId* (Line 3) defines the unique id of the current deployment. The *components* (Line 4-5) include the components that have to be deployed and *inputs* define the necessary input properties used for deployment of the respective components (Line 7-11). Using the presented JSON body, the deployment can be executed by a REST call and accessed by a workflow engine for the orchestration. The response body then contains the output properties, e.g. hostname. Further, *populateInputProperties()* has been added and manages the propagation of input properties. This method correlates with *executeTechnology()* and takes the inputs (Lines 7-11) and propagates them through the EDMM Transformation Framework. As a result, the Framework is injected with necessary properties and can start the deployment of the component. Furthermore, the *Templating* class is responsible for the mapping of the generated deployment order to the BPMN Template. The methods are also responsible for the creation of the *Initial Sequence* as well as the *Main Sequence* of the BPMN file. The created files are then merged with the *mergeSequence()* method. On the far right side of the diagram, the class *Workflow Engine*, coordinates REST calls between the workflow engine and the EDMM Transformation Framework. Consequently, when a BPMN file is executed by the workflow engine, BPMN tasks are executed consecutively referencing the corresponding implementation. A *Deploy* BPMN task would for instance reference the *deployDelegate()* implementation, whereas a *Send* BPMN task would reference a *sendDelegate()*.

BPMN Templating

For the mapping of elements to a participant-specific workflow w_i, the open-source solution Apache FreeMarker [Fre] is used. The template engine allows the generation of text (XML, HTML, etc.) based on specified templates and the passed data. The template is defined by expressions with

square brackets and an operation within, e.g. <#if deployTasks?has_content>. The expression <#if deployTasks?has_content> would for instance only be executed when the passed Java list *deployTasks* is not empty. Furthermore, the insertion of values is done through the following syntax: ${expression} and allows the mapping of Java objects to the template expression. In Listing 5.4, a FreeMarker template is shown and illustrates a *Deploy* BPMN task. In Line 2, the elements in the list *deployTasks* are printed out and mapped to the corresponding expressions through Line 3-15. This BPMN mapping is done through all elements of the list and at the end merged for the final executable BPMN workflow. For the prototypical implementation, the following templates have been created: *Initial Sequence* for the initiation of the deployment between multiple participants and *Main Sequence* for the deployment of technologies as well as communication with other participants (send, receive).

Graphical User Interface

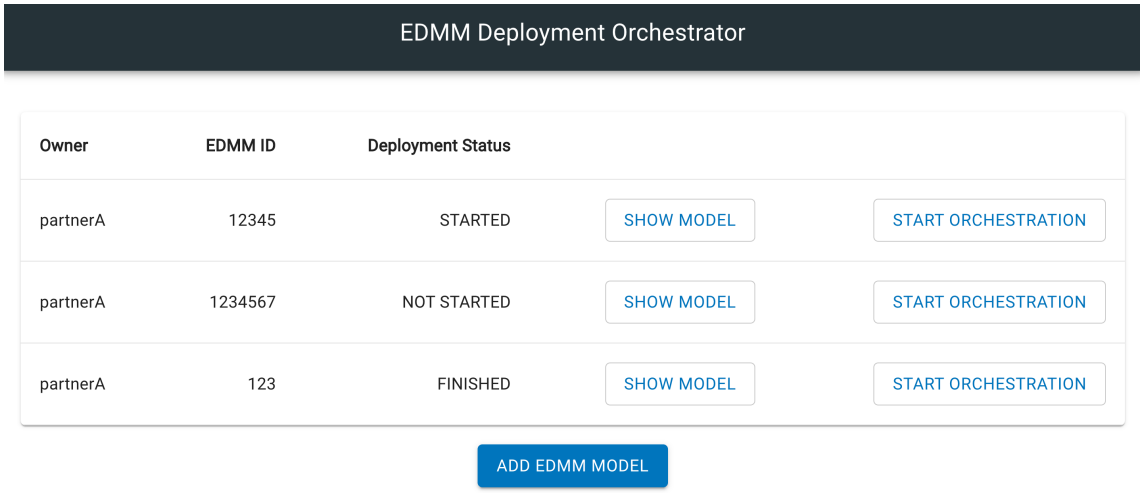

Figure 5.2: Simple Graphical User Interface for the automated management of BPMN workflow files

Since the presented approach in Chapter 4 requires a few steps to be done manually, e.g. passing the generated BPMN workflow to the workflow engine, a GUI has been implemented to automate these steps in the background. As shown in Figure 5.2, a simple GUI allows the upload of an LDM and automatically passes the model to the EDMM Transformation Framework. Based on the LDM, the EDMM Transformation Framework generates DTSMs, a BPMN workflow, and automatically uploads the generated workflow to the integrated Camunda workflow engine. Since the workflow is uploaded to the engine, the deployment can be initiated by a participant over the GUI. In addition, the GUI displays a status whether the BPMN workflow has already been uploaded to the engine or not. Furthermore, the execution and the current deployment status can be monitored. However, the deployment status is based on the participant's BPMN workflow, e.g. if the workflow has reached the end, the deployment is shown as completed, although the overall deployment is still ongoing. As a result of the GUI, the coordination of the generated BPMN workflow and the communication between the EDMM Transformation Framework is eased and the whole application deployment is further automated.

Workflow Engine

The Camunda workflow engine[3] is used for the execution of BPMN workflows. It covers most of the symbols defined in the BPMN 2.0 standard[4] and provides *Service Tasks* for custom implementation. This is done by adding the Maven dependency[5] to integrate the Java API in the application. Due to Java Delegates[6], custom Java Code can be referenced and called from BPMN tasks and allows the definition on how BPMN activities have to be executed. As a result, every BPMN task is linked to a specific Java Delegate implementation and handles the deployment with the EDMM Transformation Framework. For instance, a *Deploy* BPMN task is automatically linked to a Java class called *DeployDelegate()* and uses the properties defined in the BPMN task to create a REST body as shown in Listing 5.3 for the deployment. To summarize, the Camunda workflow engine is integrated to the EDMM Transformation Framework through the Maven dependency and acts as an interface to execute custom logic behind a BPMN task.

Listing 5.5 Structure of created transformation files in the perspective of participant B

```
multi_12345
├─ execution.plan.json
├─ bpmnExecution.plan.json
├─ bpmn
│  └─ Workflow.bpmn
├─ step0_TERRAFORM
│  ├─ files
│  │  └─ ...
│  └─ compute.tf
├─ step1_ANSIBLE
│  ├─ files
│  │  └─ ...
│  └─ playbook.yml
└─ ...
```

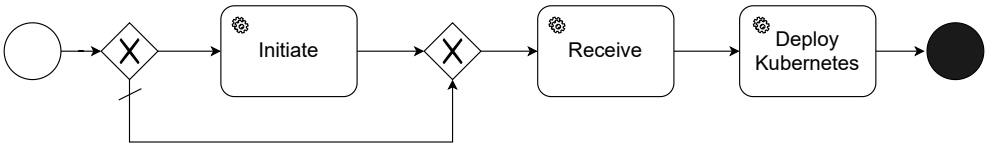

Figure 5.3: Generated BPMN workflow of participant A

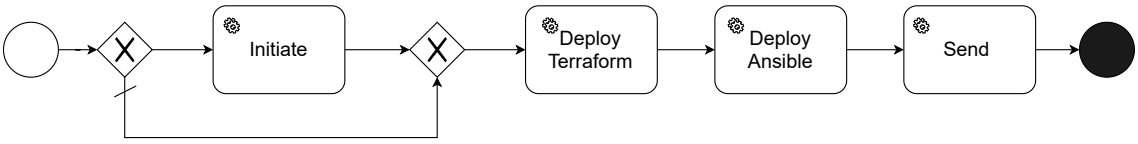

Figure 5.4: Generated BPMN workflow of participant B

5.3 Case Study

In this section, the usage of the presented approach in Chapter 4 is validated based on a simplified case. The case study follows the deployment scenario illustrated in Figure 2.6 and defines three deployment technologies and two participants. In the course of this work, the EDMM Transformation Framework has been extended as depicted in Figure 4.11.

First, a GDM is created by composing the structure of the overall application. This is done by defining all components, their respective participants and the relations between the components. The *participant* block, shown in Listing 5.1, defines all components and their participants they belong to. In addition, property values referencing property values of another component can be defined using the presented notation ${component.value}. Secondly, the GDM is shared through all participants with each participant annotating their own LDM along the *technology_regions* and additional information of the components that were not provided upfront. After all LDMs have been created, each participant uploads the LDM to their respective EDMM Transformation Framework by using the GUI. The EDMM Transformation Framework then parses the given model and generates the desired DTSMs by using the technology-specific plugins in the *Transformation* component. Furthermore, the *Orchestration* component generates a deployment order, creating *bpmnExecution.plan.json*, by sorting the LDGs topologically. Using the deployment order, the *Workflow Generation and Management* component generates a participant-specific BPMN workflow and maps the LDGs to the BPMN task templates. A resulting file structure of Participant B is depicted in Listing 5.5 and shows the generated deployment order *bpmnExecution.plan.json* in a JSON format. This JSON contains necessary operations and property values for the mapping to a BPMN Template. After the mapping of the JSON file to the BPMN templates is done, a *Workflow.bpmn* is created with all BPMN tasks to deploy and orchestrate the components of multiple participants. The generated workflow of participant B is shown in Figure 5.4 and includes *Deploy* tasks for the technology execution and *Send* tasks for the cross-participant data exchange. On the other side, the generated workflow of participant A, shown in Figure 5.3, contains a *Receive*

[3] https://camunda.com/de/products/camunda-bpm/bpmn-engine/
[4] https://www.omg.org/spec/BPMN/2.0.2
[5] https://mvnrepository.com/artifact/org.camunda.bpm
[6] https://docs.camunda.org/manual/7.14/user-guide/process-engine/delegation-code/java-delegate

task, which waits for input property values of the other participant before the *Deploy* Kubernetes task can be executed. At last, the deployment can be initiated by an arbitrary participant using the GUI. The corresponding DTSMs, e.g. all resource files under the *step0_TERRAFORM* and *step1_ANSIBLE* directories, are executed through the workflow engine. Now, as illustrated in the scenario in Figure 2.6, the deployment group *Terraform* is deployed as a *Ubuntu 18.04* VM by Participant B and returns the property value *Hostname* as an output, which is further used for the deployment group *Ansible*. After *DB* and *DBMS* have been deployed, property values annotated with the syntax ${component.value} are collected and sent through the *Camunda Engine* component to Participant A. After Participant A receives the necessary property values of Participant B, the deployment is continued and the Deploy task *Kubernetes* is executed. The overall application deployment is finished, when all BPMN workflows have reached its end event and the workflow engine is shut down.

6 Conclusion and Future Work

This work presented an approach for a decentralized cross-organizational application deployment using multiple different deployment technologies. Multiple participants are able to model the overall structure of an application and annotate the components the respective participant is responsible for. The model is then divided into participant-specific local models, where deployment technologies and property values can be added for the data exchange with other participants. Based on this local model, a participant-specific workflow is generated and used for a workflow engine for the automated deployment execution. Together, all workflows form implicitly the deployment choreography.

For this, an existing work based on the EDMM model [WBL+21] has been extended for the annotation of multiple participants and different deployment technologies. Moreover, for the creation of BPMN workflows, a templating approach was introduced to map deployment groups to BPMN tasks. These BPMN tasks included custom implementation to either deploy the respective deployment technology or communicate with other participants for the data exchange. Lastly, a workflow engine was integrated to handle the orchestration of BPMN tasks as well as the execution of deployment technologies. To validate the presented approach, a prototypical implementation based on a simplified case study was demonstrated.

The presented approach is currently limited to EDMM and its defined YAML syntax. For future work, TOSCA, which is heavily used in research but barely used in industry, could be combined with this approach to target a wider audience in both fields. Wurster et al. [WBF+20] showed that EDMM is a subset of TOSCA and therefore showed that future work on this subject is possible. Furthermore, the EDMM Transformation Framework has only been extended for the automated deployment of the three deployment technologies Terraform, Ansible, and Kubernetes. This issue could be further tackled by adding more execution plugins to enable flexibility on the deployment technology decision. At last, the prototypical implementation presented a GUI for the management of generated BPMN workflows. However, the functionalities are limited and could be integrated for the future with further features for a better user experience and usability.

Bibliography

[Ans21] Ansible, ed. *Ansible is Simple IT Automation*. https://www.ansible.com/. (Accessed on 03/09/2021). 2021 (cit. on p. 15).

[App21] S.-b. P. C. Application, ed. *GitHub - spring-projects/spring-petclinic: A sample Spring-based application*. https://github.com/spring-projects/spring-petclinic. (Accessed on 03/09/2021). 2021 (cit. on p. 24).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA–a runtime for TOSCA-based cloud applications". In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 692–695 (cit. on p. 15).

[BBK+14] 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 pp. 16, 26).

[BBKL14] 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. 17).

[BCS18] A. Brogi, A. Canciani, J. Soldani. "Fault-aware management protocols for multi-component applications". In: *Journal of Systems and Software* 139 (2018), pp. 189–210 (cit. on p. 15).

[CCDT17] D. Calcaterra, V. Cartelli, G. Di Modica, O. Tomarchio. "A framework for the orchestration and provision of cloud services based on TOSCA and BPMN". In: *International Conference on Cloud Computing and Services Science*. Springer. 2017, pp. 262–285 (cit. on p. 26).

[Che21] Chef, ed. *Chef: Enabling the Coded Enterprise through Infrastructure, Security and Application Automation*. https://www.chef.io/. (Accessed on 03/09/2021). 2021 (cit. on pp. 15, 19).

[Clo] A. Cloudformation. *AWS CloudFormation – Infrastruktur als Code und AWS-Ressourcenbereitstellung*. https://aws.amazon.com/de/cloudformation/. (Accessed on 03/09/2021) (cit. on p. 19).

[CT12] M. Chinosi, A. Trombetta. "BPMN: An introduction to the standard". In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134 (cit. on p. 22).

[DJ18] R. Dukaric, M. B. Juric. "BPMN extensions for automating cloud environments using a two-layer orchestration approach". In: *Journal of Visual Languages & Computing* 47 (2018), pp. 31–43 (cit. on p. 26).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. imperative: two modeling patterns for the automated deployment of applications". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. 2017, pp. 22–27 (cit. on pp. 17, 18).

[EEKS11] T. Eilam, M. Elder, A. V. Konstantinou, E. Snible. "Pattern-based composite application deployment". In: *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. IEEE. 2011, pp. 217–224 (cit. on p. 26).

[Eng21] C. W. Engine, ed. *Workflow und Decision Automation | Camunda*. https://camunda.com/de/. (Accessed on 03/09/2021). 2021 (cit. on p. 22).

[Fre] Freemarker. *FreeMarker Java Template Engine*. https://freemarker.apache.org/. (Accessed on 03/09/2021) (cit. on p. 51).

[GMMC13] 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. 25).

[HLB19] R. Hentschel, C. Leyh, T. Baumhauer. "Critical success factors for the implementation and adoption of cloud services in SMEs". In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 2019 (cit. on p. 17).

[Kub21] Kubernetes, ed. *Kubernetes*. https://kubernetes.io/de/. (Accessed on 03/09/2021). 2021 (cit. on p. 19).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011) (cit. on p. 15).

[Mic21] Microsoft. *Azure Resource Manager*. https://docs.microsoft.com/de-de/azure/azure-resource-manager/management/overview. (Accessed on 03/09/2021). 2021 (cit. on p. 19).

[OAS21] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html. (Accessed on 03/09/2021). 2021 (cit. on p. 15).

[OMG11] OMG. "Business Process Model and Notation (BPMN) Version 2.0." In: (2011) (cit. on pp. 16, 22).

[PKT+20] G. Pierantoni, T. Kiss, G. Terstyanszky, J. DesLauriers, G. Gesmier, H.-V. Dang. "Describing and processing topology and quality of service parameters of applications in the cloud". In: *Journal of Grid Computing* 18.4 (2020), pp. 761–778 (cit. on p. 25).

[Pup] Puppet. *Powerful infrastructure automation and delivery | Puppet*. https://puppet.com/. (Accessed on 03/09/2021) (cit. on p. 19).

[SBKL20] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Method, formalization, and algorithms to split topology models for distributed cloud application deployments". In: *Computing* 102.2 (2020), pp. 343–363 (cit. on pp. 35, 36).

[SIA19] J. Sandobalin, E. Insfran, S. Abrahão. "ARGON: A Model-Driven Infrastructure Provisioning Tool". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2019, pp. 738–742 (cit. on p. 25).

[SVW+18] M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, F. De Turck. "Orchestrator conversation: Distributed management of cloud applications". In: *International Journal of Network Management* 28.6 (2018), e2036 (cit. on p. 25).

[Ter21] Terraform, ed. *Terraform by HashiCorp*. https://www.terraform.io/. (Accessed on 03/09/2021). 2021 (cit. on pp. 15, 19).

[WBB+19] 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. 15, 21, 30, 47).

[WBF+20] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. "The essential deployment metamodel: a systematic review of deployment automation technologies". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 63–75 (cit. on pp. 15, 17–21, 29, 36, 43, 57).

[WBK+20] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. "Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2020, pp. 20–35 (cit. on pp. 15, 16, 23, 24, 37).

[WBL+21] M. Wurster, U. Breitenbücher, F. Leymann, K. Wild, F. Diez. "Automating the Deployment of Distributed Applications by Combining Multiple Deployment Technologies". In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, 2021 (cit. on pp. 16, 23, 24, 30, 33, 35–37, 47, 49, 57).

[Whi04] S. A. White. "Introduction to BPMN". In: *Ibm Cooperation* 2.0 (2004), p. 0 (cit. on p. 22).

All links were last followed on March 09, 2021.

A Appendix

A.1 Extended EDMM Model for the case study

```
multi_id: 12345
owner: partnerB

orchestration_technology:
    terraform:
        - ubuntu_db
    ansible:
        - dbms
        - db

participants:
  partnerB:
    endpoint: http://localhost:5000
    components:
      - ubuntu_db
      - db
      - dbms
  partnerA:
    endpoint: http://localhost:6000
    components:
      - ubuntu_app
      - pet_clinic
      - pet_clinic_tomcat

components:
    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

db:
    type: mysql_database
    properties:
        hostname: ${ubuntu_db.hostname}
        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:
        hostname: ${ubuntu_db.hostname}
        root_password: petclinic
    operations:
        create: ./files/mysql_dbms/create.sh
        start: ./files/mysql_dbms/start.sh
    relations:
        - hosted_on: ubuntu_db

ubuntu_app:
    type: compute
    properties:
        machine_image: ubuntu
        instance_type: large
        key_name: key
        priv_key_path: ./files/ubuntu/key.pem
    artifacts:
        - provider: ./files/ubuntu/openstack.json

ubuntu_db:
    type: compute
    properties:
        hostname:
        machine_image: ubuntu
        instance_type: large
        key_name: key
        priv_key_path: ./files/ubuntu/key.pem
    artifacts:
        - provider: ./files/ubuntu/openstack.json
```

A.2 Automatically generated BPMN workflow of participant B based on the templating approach

```
  <?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:bpmndi="http:
//www.omg.org/spec/BPMN/20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:di="http://www.omg.org/spec/DD
/20100524/DI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="http://
bpmn.io/schema/bpmn" exporter="Camunda Modeler" exporterVersion="4.2.0">
  <bpmn:process id="workflow" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>InitiateFlow_1</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:exclusiveGateway id="Gateway_1" default="InitiateFlow_4">
      <bpmn:incoming>InitiateFlow_1</bpmn:incoming>
      <bpmn:outgoing>InitiateFlow_2</bpmn:outgoing>
      <bpmn:outgoing>InitiateFlow_4</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:sequenceFlow id="InitiateFlow_1" sourceRef="StartEvent_1" targetRef="Gateway_1" />
    <bpmn:sequenceFlow id="InitiateFlow_2" sourceRef="Gateway_1" targetRef="Initiate_1">
      <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">${initiator == true}</
bpmn:conditionExpression>
    </bpmn:sequenceFlow>
    <bpmn:exclusiveGateway id="Gateway_2">
      <bpmn:incoming>InitiateFlow_3</bpmn:incoming>
      <bpmn:incoming>InitiateFlow_4</bpmn:incoming>
      <bpmn:outgoing>Flow_0</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:sequenceFlow id="InitiateFlow_3" sourceRef="Initiate_1" targetRef="Gateway_2" />
    <bpmn:sequenceFlow id="InitiateFlow_4" sourceRef="Gateway_1" targetRef="Gateway_2" />
    <bpmn:serviceTask id="Initiate_1" name="Initiate" camunda:delegateExpression="${
initiateDelegate}">
      <bpmn:extensionElements>
        <camunda:properties>
        </camunda:properties>
      </bpmn:extensionElements>
      <bpmn:incoming>InitiateFlow_2</bpmn:incoming>
      <bpmn:outgoing>InitiateFlow_3</bpmn:outgoing>
    </bpmn:serviceTask>
```

```
<bpmn:serviceTask id="ID_0" name="DEPLOY TERRAFORM" camunda:delegateExpression="${
deployDelegate}">
  <bpmn:extensionElements>
    <camunda:properties>
      <camunda:property name="participant" value="http://localhost:5000" />
      <camunda:property name="component" value="ubuntu_db" />
    </camunda:properties>
  </bpmn:extensionElements>
  <bpmn:incoming>Flow_0</bpmn:incoming>
  <bpmn:outgoing>Flow_1</bpmn:outgoing>
</bpmn:serviceTask>

<bpmn:serviceTask id="ID_1" name="DEPLOY ANSIBLE" camunda:delegateExpression="${deployDelegate
}">
  <bpmn:extensionElements>
    <camunda:properties>
      <camunda:property name="participant" value="http://localhost:5000" />
      <camunda:property name="component" value="dbms" />
      <camunda:property name="component" value="db" />
    </camunda:properties>
  </bpmn:extensionElements>
  <bpmn:incoming>Flow_1</bpmn:incoming>
  <bpmn:outgoing>Flow_2</bpmn:outgoing>
</bpmn:serviceTask>
<bpmn:sequenceFlow id="Flow_0" sourceRef="Gateway_2" targetRef="ID_0" />
<bpmn:sequenceFlow id="Flow_1" sourceRef="ID_0" targetRef="ID_1" />

<bpmn:serviceTask id="ID_2" name="SEND" camunda:delegateExpression="${sendDelegate}">
  <bpmn:extensionElements>
    <camunda:properties>
      <camunda:property name="participant" value="http://localhost:6000" />
      <camunda:property name="input" value="DB_USER" />
      <camunda:property name="input" value="DB_PORT" />
      <camunda:property name="input" value="DB_HOSTNAME" />
      <camunda:property name="input" value="DB_PASSWORD" />
      <camunda:property name="component" value="db" />
    </camunda:properties>
  </bpmn:extensionElements>
  <bpmn:incoming>Flow_2</bpmn:incoming>
  <bpmn:outgoing>Flow_3</bpmn:outgoing>
</bpmn:serviceTask>
<bpmn:sequenceFlow id="Flow_2" sourceRef="ID_1" targetRef="ID_2" />

<bpmn:endEvent id="EndEvent">
  <bpmn:incoming>Flow_3</bpmn:incoming>
</bpmn:endEvent>
<bpmn:sequenceFlow id="Flow_3" sourceRef="ID_2" targetRef="EndEvent" />

</bpmn:process>
</bpmn:definitions>
```

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