

Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master's Thesis

Transformation of Technology-specific Deployment Models into Technology-agnostic Deployment Models

Marcel Weller

| | |
|-------------------------|---|
| Course of Study: | Softwaretechnik |
| Examiner: | Prof. Dr.-Ing. Steffen Becker |
| Supervisor: | Sandro Speth, M. Sc. Dr. rer. nat. Uwe Breitenbücher |
| Commenced: | November 1, 2021 |
| Completed: | April 29, 2022 |

Abstract

The deployment of software applications to the cloud is a highly complex process that leaves plenty of room for further improvement and optimization. To facilitate the automation and management of deployment processes, development teams can use various deployment technologies to create deployment models. For that, developers must acquire specialized knowledge and expertise in the deployment technologies they intend to use. Consequently, development teams create technology-specific deployment models that are difficult to comprehend for developers and other application stakeholders who do not have the required expertise. This work aims to transform technology-specific deployment models into technology-agnostic deployment models to facilitate the comprehensibility of application deployments. Technology-agnostic deployment models describe application deployments using abstract concepts that do not require experience with specific deployment technologies to comprehend them. In this work, we present the concept of a transformation framework capable of performing such a transformation process. Furthermore, we evaluate a prototypical realization of the transformation framework that supports four different deployment technologies. The results show that the transformation framework can create technology-agnostic deployment models that contain most of the application deployment information. However, we also found that the transformation framework requires significantly more development effort to support arbitrary deployment technologies. The presented concept provides a foundation for future work exploring various topics regarding the transformation of technology-specific deployment models into technology-agnostic deployment models.

Kurzfassung

Das Deployment von Softwareanwendungen in der Cloud ist ein hochkomplexer Prozess, der noch viel Raum für weitere Verbesserungen und Optimierungen lässt. Um die Automatisierung und Verwaltung von Deployment Prozessen zu erleichtern, können Entwicklungsteams verschiedene Deployment Technologien zur Erstellung von Deployment Modellen verwenden. Dazu müssen die Entwickler spezielles Fachwissen und Expertise über die zu verwendenden Deployment Technologien erwerben. Infolgedessen erstellen Entwicklungsteams technologiespezifische Deployment Modelle, die für Entwickler und andere an der Anwendung Beteiligte, die nicht über das erforderliche Fachwissen verfügen, nur schwer zu verstehen sind. Das Ziel dieser Arbeit ist die Transformation von technologiespezifischen Deployment Modellen zu technologieagnostischen Deployment Modellen, um die Verständlichkeit der Deployments von Anwendungen zu erleichtern. Technologieunabhängige Deployment Modelle beschreiben Deployments von Anwendungen mit Hilfe abstrakter Konzepte, die keine Erfahrung mit bestimmten Deployment Technologien erfordern, um sie zu verstehen. In dieser Arbeit stellen wir das Konzept eines Transformations-Frameworks vor, das in der Lage ist, einen solchen Transformationsprozess durchzuführen. Darüber hinaus evaluieren wir eine prototypische Realisierung des Transformations-Frameworks, das vier verschiedene Deployment Technologien unterstützt. Die Ergebnisse zeigen, dass das Transformations-Framework technologieunabhängige Deployment Modelle erstellen kann, die einen Großteil der Informationen über das Deployment der Anwendung enthalten. Allerdings haben wir auch festgestellt, dass das Transformations-Framework wesentlich mehr Entwicklungsaufwand erfordert, um beliebige Deployment Technologien zu unterstützen. Das vorgestellte Konzept stellt eine Grundlage für zukünftige Arbeiten dar, welche sich mit der Erforschung einer Vielzahl von Themen bezüglich der Transformation von technologiespezifischen Deployment Modellen in technologieunabhängige Deployment Modelle beschäftigen können.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Foundations and Related Work | 3 |
| 2.1 | Foundations | 3 |
| 2.2 | Related Work | 7 |
| 3 | Concept and Design | 9 |
| 3.1 | Overview of the Concept | 9 |
| 3.2 | Definitions | 10 |
| 3.3 | Meta-Model for the Technology-Agnostic Deployment Model | 12 |
| 3.4 | Concept of the Transformation Framework | 15 |
| 3.5 | Architecture | 19 |
| 4 | Prototypical Realization | 35 |
| 4.1 | Java Applications | 36 |
| 4.2 | Databases | 36 |
| 4.3 | Message Broker | 37 |
| 4.4 | User Interaction | 39 |
| 4.5 | Plugins | 41 |
| 5 | Evaluation | 49 |
| 5.1 | Design | 49 |
| 5.2 | Exemplary Technology-Specific Deployment Model | 51 |
| 5.3 | Results | 53 |
| 5.4 | Discussion | 55 |
| 5.5 | Threats to Validity | 57 |
| 6 | Conclusion | 59 |
| 6.1 | Summary | 59 |
| 6.2 | Benefits | 59 |
| 6.3 | Limitations | 60 |
| 6.4 | Lessons Learned | 60 |
| 6.5 | Future Work | 61 |
| | Bibliography | 63 |
| A | Expected Technology-Agnostic Deployment Model | 67 |
| B | Evaluation Results | 93 |
| B.1 | M1: Application Logs Summary | 93 |

| | | |
|-----|---|-----|
| B.2 | M2: Registered Plugins | 95 |
| B.3 | M3: Message Broker Definitions | 96 |
| B.4 | M5: Actual Technology-Agnostic Deployment Model | 100 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Directions of relationships between components of an application topology [WBB+20b]. | 6 |
| 3.1 | High-level concept of the transformation framework. | 9 |
| 3.2 | Relationship between deployment model and embedded deployment model. | 11 |
| 3.3 | Process of transforming a technology-specific deployment model following a static analysis approach. | 12 |
| 3.4 | Process of transforming a technology-specific deployment model following a dynamic analysis approach. | 13 |
| 3.5 | The Essential Deployment Metamodel [WBF+20]. | 13 |
| 3.6 | Phases of the transformation process. | 18 |
| 3.7 | Component diagram of the transformation framework. The databases and queues show additional information about which entities they store. There can be multiple instances of the components <i>Plugin</i> and <i>AnalysisTaskRequestQueue</i> | 19 |
| 3.8 | Unified Modeling Language (UML) Class diagram of the internal representation of the technology-specific deployment model. | 21 |
| 3.9 | UML Class diagram of the internal technology-agnostic deployment model. The original Essential Deployment Metamodel (EDMM) is extended with information for internal processing, marked in blue. | 25 |
| 3.10 | UML Sequence diagram showing the plugin registration process. | 27 |
| 3.11 | UML Class diagram for the Plugin. It is created by the <i>AnalysisManager</i> and persisted in the <i>ConfigurationsDatabase</i> during the plugin registration process. | 28 |
| 3.12 | UML sequence diagram showing the start of the transformation process. The <i>AnalysisManager</i> creates the first <i>AnalysisTask</i> and sends it to a suitable <i>Plugin</i> | 29 |
| 3.13 | UML Class diagram of the analysis task. | 30 |
| 3.14 | UML class diagram of the different messages that the <i>AnalysisManager</i> and the <i>Plugins</i> send and consume from the messaging queues during the transformation process. | 30 |
| 3.15 | UML sequence diagram showing the analysis and possible responses of a <i>Plugin</i> during the transformation process. | 31 |
| 3.16 | UML sequence diagram showing the first steps of the processing of an <i>AnalysisTaskResponse</i> by the <i>AnalysisManager</i> | 33 |
| 3.17 | UML sequence diagram showing how the <i>AnalysisManager</i> searches for and starts <i>AnalysisTasks</i> that need to be run. | 34 |
| 4.1 | UML component diagram showing the architecture of the realized prototype. | 35 |
| 4.2 | Overview of the entities in the AMQP model. | 37 |
| 4.3 | The tad-shell CLI with all available commands. | 39 |

| | | |
|-----|---|----|
| 4.4 | Execution of the transform command in the tad-shell CLI. The transformation framework prints the result of a successful transformation process to the tad-shell. | 40 |
| 4.5 | UML Class diagram of the internal Terraform model. | 43 |
| 4.6 | UML Class diagram of the meta-model for the internal representation of Kubernetes deployment objects. | 45 |
| 4.7 | UML Class diagram of the meta-model for the internal representation of Kubernetes service objects. | 46 |
| 5.1 | Component Diagram of the T2 Project [SSB22]. | 52 |
| 5.2 | Overview of the exemplary technology-specific deployment model. It shows the structure of the deployment model with all embedded deployment models and the deployment technologies. | 53 |

List of Tables

| | | |
|-----|-----------------------------------|----|
| 2.1 | Deployment Technologies | 5 |
| 5.1 | GQM model | 50 |

List of Listings

| | | |
|-----|---|-----|
| 3.1 | Example of a deployment model created with EDMM in YAML specification. | 14 |
| 4.1 | Specification of relations in the modified EDMM in YAML specification. | 40 |
| 4.2 | Syntax of the Terraform language [Has21]. | 42 |
| 4.3 | EDMM component type for a physical node in the YAML syntax. | 43 |
| 4.4 | Definition of exemplary Kubernetes deployment and service objects in YAML format. | 44 |
| 5.1 | Output of the transformation result to the Command-Line Interface (CLI) of the transformation framework. It shows the location of the technology-agnostic deployment model and the calculated values for the metrics. | 55 |
| A.1 | The expected technology-agnostic deployment model. It is the result that we expect from the transformation framework when it transforms the exemplary technology-specific deployment model of the T2 Project. | 67 |
| B.1 | Summary of the events in the application logs from the viewpoint of the analysis manager. | 93 |
| B.2 | Registered plugins in the configurations database. | 95 |
| B.3 | Created AMQP entities on the RabbitMQ message broker. | 96 |
| B.4 | The actual technology-agnostic deployment model that the transformation framework created from the transformation of the exemplary technology-specific deployment model. | 100 |

Acronyms

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

BSON Binary JSON.

CLI Command-Line Interface.

DNS Domain Name System.

DSL Domain Specific Language.

ECA Environment-Centric Artifact.

EDMM Essential Deployment Metamodel.

GQM Goal Question Metric.

GUI Graphical User Interface.

IaaS Infrastructure as a Service.

IaC Infrastructure as Code.

JAR Java Archive.

JSON Javascript Object Notation.

NCA Node-Centric Artifact.

PaaS Platform as a Service.

REST Representational State Transfer.

SaaS Software as a Service.

SQL Structured Query Language.

TOSCA Topology and Orchestration Specification for Cloud Applications.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

YAML YAML Ain't Markup Language.

1 Introduction

In recent years, both industry and research adopted many deployment technologies to meet the growing need for deployment management and automation. Software applications nowadays typically consist of several components for which different development teams or external providers are responsible [BBK+13; BBK+14; EEKS11]. For example, development teams can integrate service offerings of cloud providers with their self-written components, which makes its deployment a challenging task.

Since manual deployments are error-prone and hard to repeat in the exact same manner, it is crucial to automate the deployment process [HBF+18; WBF+20]. Deployment technologies allow to model an application deployment in a deployment model and provide means to automatically deploy it to a target deployment environment such as a specific cloud platform. With this, they enable fast and repeatable application deployments.

Individual deployment technologies can differ heavily in features and mechanisms, for example, regarding the Domain Specific Language (DSL) that they provide to create the deployment model [EBF+17]. As applications are complex and deployment technologies typically serve a specific purpose, development teams often need to apply a mix of different deployment technologies [BBK+14]. The corresponding technology-specific deployment models are hard to create, understand and modify because this requires a high level of understanding and technical expertise in the individual deployment technologies [BBK+14; EBLW17; HBF+18]. Because the application development and operation involve many people with various backgrounds, the comprehensibility of deployment models is crucial for the business's success. For the communication of the application deployment, it is sufficient to give an overview of the components, their interconnections and the required cloud resources that the deployment model describes. However, this information is often not visible in the deployment model because the technical details of the deployment technology obscure it.

Therefore, we seek to transform technology-specific deployment models into deployment models independent of deployment technologies. These technology-agnostic deployment models rely on general concepts to overview the application deployment while preserving all relevant information to deploy it. As a result, the required technical expertise to comprehend a technology-agnostic deployment model should be significantly lower.

We present a concept and architecture for a transformation framework that takes a technology-specific deployment model as input and outputs a technology-agnostic deployment model. The framework follows a plugin-based approach, where each plugin is responsible for transforming a specific deployment technology. We also provide a concept for the transformation process that the transformation framework executes. Furthermore, we select the EDMM as the meta-model for the technology-agnostic deployment model. We developed a prototypical realization of the transformation framework with four plugins to test and evaluate the presented concept. For that, we use an exemplary technology-specific deployment model that consists of different deployment technologies to represent a complex and realistic application deployment.

The evaluation result shows that the transformation framework finds most of the information in the exemplary technology-specific deployment model and can create adequate components of the technology-agnostic deployment model. However, the transformation framework cannot detect some information, which requires further improvements to both the concept and the prototypical realization. The most critical points herein are improvements in the application of the EDMM type system and exploring possibilities for the plugins to communicate findings. With the plugin-based approach, we can extend the transformation framework to include more deployment technologies or implement alternative transformation methods.

We conclude that the concept is generally applicable for the transformation of deployment models. While we identify some ways to improve it, we see the biggest challenge in implementing the transformation framework. The support for a wide range of different deployment technologies and the ability to retain all relevant information requires significant development effort of various plugins.

The main contribution of this thesis is the concept and architecture of the transformation framework and its transformation process. It shows a general approach for transforming technology-specific deployment models into technology-agnostic deployment models. Furthermore, the evaluation of the prototypical realization reveals challenges that we still need to overcome. Finally, we outline different possibilities for future work to investigate topics based on this concept.

Thesis Structure

The thesis is structured in the following chapters.

Chapter 2 – Foundations and Related Work: First, we provide information about the most important foundations for this work and related work that have already done research in a similar direction.

Chapter 3 – Concept and Design This chapter presents the main contribution of the thesis. It contains the concept and the architecture of the transformation framework.

Chapter 4 – Prototypical Realization Here, we describe how we implemented a prototype of the transformation framework.

Chapter 5 – Evaluation: This chapter describes the evaluation of the prototypical realization of the transformation framework and discusses the results.

Chapter 6 – Conclusion We conclude our thesis by outlining benefits, limitations, lessons learned and possible future work.

2 Foundations and Related Work

This chapter presents essential concepts that form the foundation of our work in Section 2.1. It continues with related work that has already explored similar topics in Section 2.2.

2.1 Foundations

We provide details on important concepts to understand the thesis in the following. First, we give an overview of essential aspects of cloud computing. After that, we describe deployment technologies and deployment models. Based on this, we provide more details on the characteristics of technology-agnostic deployment models and different approaches.

Cloud Computing

To deploy applications quickly and reliably, cloud computing is an optimal solution. *Cloud providers* offer decentralized compute resources or other services through a *cloud platform*. Customers can rent these cloud offerings on-demand. Consequently, customers do not need to provide expensive hardware or expertise on-site, which saves effort, time, and costs.

Mell and Grance [MG11] define the most important aspects of cloud computing to facilitate comparisons and discussions about topics in this area. They define three *cloud service models*: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. The cloud service models categorize approaches of cloud platforms for providing offerings to the user. IaaS offerings provide capabilities to deploy infrastructure resources like computing, network, and storage. The customer has complete control over the subsequent deployment of software components but not the underlying infrastructure. With PaaS offerings, the customer can deploy a hosting environment on the cloud platform that may consist of several provider-created software components. They can then deploy their customer-created applications in this hosting environment. The underlying infrastructure is not visible to the customer, but they can configure the software components of the hosting environment. Finally, cloud providers can deploy software applications on the cloud platform and provide it as a SaaS offering. Customers that rent such services do not have control over the deployed components and can at most configure some options of the application.

Furthermore, Mell and Grance [MG11] define four *cloud deployment models*, which describe who has access to the offerings and where the cloud platform is located. *Private clouds* restrict access to single organizations that can host the cloud platform on- or off-premise. *Community clouds* expand this concept and share access to the offerings between organizations. In contrast, offerings of *public*

clouds are open for use by everyone and reside on the premises of a cloud provider. Finally, they define *hybrid clouds*, which is a combination of several cloud platforms that can follow different cloud deployment models.

It is crucial to differentiate between the terms “cloud deployment model” and “deployment model” because they are different concepts, and we use both of them throughout the thesis. We describe the term deployment model in the following section.

Deployment Technologies and Deployment Models

The deployment process of a software application combines several activities like the release, installation, update, and removal of the whole system or its components [CFH+98]. It involves transferring the software application code to the desired *deployment environment*, which consists of infrastructure resources and optionally a set of other preinstalled software components. As it is challenging to achieve fast and repeatable application deployments by executing the deployment process manually, deployment technologies have emerged that automate the deployment process and tackle these problems [EBF+17; HBF+18; WBF+20].

Deployment technologies allow creating automatically executable models, so-called *deployment models* [EBF+17]. Deployment models describe the components, their relationships, and the required infrastructure resources of software applications [HBF+18; WBF+20]. Other literature commonly uses the term “deployment artifacts” for deployment models [EBLW17; WBB+20a]. Deployment technologies provide a DSL, which is often based on Javascript Object Notation (JSON)¹ or YAML Ain’t Markup Language (YAML)² for the creation of the deployment models [HBF+18; WBF+20]. A deployment model is a set of files that many people can manage and maintain in collaboration and through a version control system. As a result, deployment models enable a repeatable deployment process.

Deployment models follow either a *declarative* or *imperative* deployment modeling approach [EBF+17]. Declarative deployment models describe the desired state of the application deployment using structural models. The deployment technology provides a *deployment engine* that interprets the declarative deployment model and then deploys the application. Doing so enforces the desired state on the actual state of the running application. In contrast, imperative deployment models explicitly describe the execution of processes required for the deployment. As such, they make use of procedural models that a *process engine* executes. Declarative deployment models are more widely accepted in the industry [WBF+20].

Wettinger et al. [WBKL16] provide a systematic classification for *DevOps artifacts*. These are deployment models that are stored in public repositories, like Chef cookbooks or Juju charms. Development teams can reuse these DevOps artifacts to integrate them into larger software applications. They identify two classes of DevOps artifacts: Node-Centric Artifacts (NCAs) and Environment-Centric Artifacts (ECAs). A node is either a physical server, a virtual machine, or a container. NCAs are deployed on a single node and therefore consist of only one component without any relations to other components. In contrast, ECAs consist of several interconnected components that require the deployment on multiple nodes.

¹JSON homepage: <https://json.org/>

²YAML homepage: <https://yaml.org/>

Wurster et al. [WBF+20] give an overview of the most popular deployment technologies by the number of search hits on Google (see Table 2.1). Furthermore, they assign these deployment technologies to three categories: *General-Purpose*, *Provider-Specific* and *Platform-Specific*. They base this categorization on similarities found in the deployment technologies' features. The features relate to the concepts of cloud computing defined by Mell and Grance [MG11]. General-purpose technologies support all kinds of cloud service models and cloud deployment models. On the other hand, provider-specific deployment technologies are restricted to a specific cloud provider and can therefore not support the hybrid cloud deployment model. Lastly, they define platform-specific deployment technologies, which only support one or some cloud service models and rely on specific *platform bundles* like container images.

| Technology | Category |
|------------------------|-------------------|
| Puppet | General-Purpose |
| Chef | General-Purpose |
| Ansible | General-Purpose |
| Kubernetes | Platform-Specific |
| OpenStack HEAT | General-Purpose |
| Terraform | General-Purpose |
| AWS CloudFormation | Provider-Specific |
| SaltStack | General-Purpose |
| Juju | General-Purpose |
| CFEngine | Platform-Specific |
| Azure Resource Manager | Provider-Specific |
| Docker Compose | Platform-Specific |
| Cloudify | General-Purpose |

Table 2.1: Most popular deployment technologies and assigned category, as of February 2019 [WBF+20].

As deployment technologies can differ heavily in supported features, there are different use cases where it makes sense to prefer one technology over the other. Some deployment technologies are based on general principles like Infrastructure as Code (IaC) or configuration management. IaC focuses on the automated deployment of infrastructure resources by describing their provisioning through code [Mor16]. Examples of deployment technologies following the IaC principle are Terraform, Pulumi and AWS CloudFormation. Other deployment technologies, like Kubernetes or Docker Compose, focus solely on the software components of an application and do not provide means to deploy infrastructure resources. Therefore, selecting an appropriate deployment technology for creating deployment models in the given use case is crucial. Furthermore, it is possible to combine deployment models of different deployment technologies into one larger deployment model. Such a combination of deployment technologies has the advantage that we can select the best fit for each part of the application deployment. At the same time, it has the downside that the deployment model grows in complexity and the deployment process is more complicated.

2.1.1 Technology-Agnostic Deployment Models

Understanding deployment models created with the DSLs of specific deployment technologies requires both technical expertise and experience with these technologies [BBK+14; EBLW17; HBF+18]. Later manual changes, especially of architectural nature, are therefore hard to carry out. Moreover, this can result in a vendor lock-in concerning the deployment technology or cloud provider, which negatively impacts the portability of the application.

Various standards, modeling languages, and meta-models aim to overcome these disadvantages by providing means to create technology-agnostic deployment models. They describe application deployments with abstract concepts that are often based on patterns and leave out technical details [EEKS11; HBF+18; LSS+13]. Consequently, technology-agnostic deployment models are generic representations of application deployments independent of specific deployment technologies.

A prominent example is the Topology and Orchestration Specification for Cloud Applications (TOSCA), which provides an extensible meta-model for instantiating concrete, technology-agnostic deployment models [Org13]. TOSCA models application deployments in the form of directed, acyclic graphs. The vertices represent logical components of the application or underlying infrastructure resources, whereas the edges denote the relationships in between. The relationships can run in two major directions: vertical and horizontal (see Figure 2.1) [WBB+20b]. [WBB+20b].

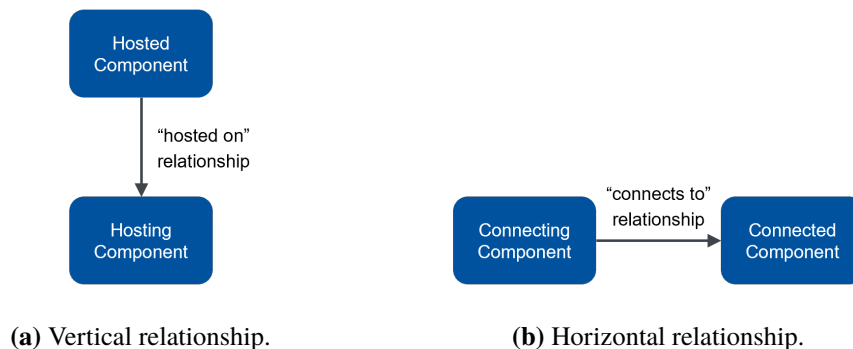


Figure 2.1: Directions of relationships between components of an application topology [WBB+20b].

Vertical relationships describe the mapping to underlying components, where one component is “hosted on” the other (see Figure 2.1a). In that sense, they form a set of components that are stacked on top of each other. At the bottom, we can commonly find components that provide infrastructure resources. In contrast, horizontal relationships describe how a component “connects to” another component or service for the exchange of information (see Figure 2.1b).

2.2 Related Work

In the following we present work that relates to the thesis topic. First, we provide information on how we searched for and selected relevant articles. Then we present the selected work and organize them in sections based on similarities in their approaches and research goals. For each related work we present an overview, point out differences, and highlight relevant concepts for the thesis.

2.2.1 Literature Research Methodology

For the research of related literature, we used the academic search engine Google Scholar. We exclusively searched for English literature. We tried different combinations of the keywords “deployment model”, “transformation”, “deployment artifact”, and “technology agnostic”. The combinations that worked the best are the following:

1. deployment model transformation
2. deployment artifact transformation
3. technology agnostic deployment

We split the selection process of the results into two steps. In the first step we looked at the first twenty results of each search string and scanned their title and abstract. This led to a preselection of potentially relevant results that we read in more detail in the second step. We often found work of the same authors that build on each other. In these cases, we tried to identify the most recent and relevant articles. To find further literature, we also looked in the references of selected work.

2.2.2 Transformation of Common Deployment Models from Public Code Repositories

In their work, Endres et al. [EBLW17] describe a methodology to crawl public code repositories for technology-specific deployment models and transform them into technology-agnostic deployment models. Their goal is to create reusable technology-agnostic deployment models from common applications and later combine them with other deployment models to build more complex applications. They focused on the deployment technology Chef, which allows the creation of technology-specific deployment models called *Chef artifacts*. In a case study, they transformed Chef artifacts into TOSCA models. For the information extraction, they used a method called *depth-first search* [Tar72]. The resulting TOSCA models lacked information about the underlying infrastructure because Chef artifacts do not explicitly describe this information. Consequently, they were not able to deploy these TOSCA models. Therefore, they suggest using a modeling tool to add the missing information manually.

Wettinger et al. [WBKL16] also pursue the creation of reusable technology-agnostic deployment models from technology-specific deployment models that they crawl from public code repositories. For that purpose, they developed a framework consisting of an end-to-end toolchain. Their focus is on the deployment technologies Chef and Juju because they represent an NCA and ECA, respectively. They describe in detail the mapping from entities and operations of the deployment technologies to nodes and relations of a TOSCA model. Moreover, they describe an iterative process in which they look for dependencies in the technology-specific deployment models to other deployment models of

the same deployment technology. If they find a dependency, they add the corresponding technology-specific deployment model to the transformation process and scan it for further dependencies. The actual transformation to the TOSCA model focuses solely on the software components and leaves out information about underlying infrastructure resources. They add this information in a subsequent step using a modeling tool that is similar to the one employed by Endres et al. [EBLW17] so that they can deploy the TOSCA models.

In contrast to both related works, we do not seek to crawl public code repositories to transform technology-specific deployment models of common applications. Furthermore, we do not focus on specific deployment technologies but rather provide a general concept that works with arbitrary deployment technologies. Finally, we avoid including manual steps in the transformation process to achieve full automation. However, we acknowledge the idea of Wettinger et al. [WBKL16] for finding dependencies to other deployment models. We partly adopt this with our notion of embedded deployment models (see Section 3.2.1).

2.2.3 The EDMM Modeling and Transformation System

Based on the commonalities of the 13 most popular deployment technologies, Wurster et al. [WBF+20] created their own standard for technology-agnostic deployment models, called the EDMM (see Table 2.1). In a follow-up work, Wurster et al. [WBB+19] built a plugin-based transformation framework to transform an EDMM model into a technology-specific deployment model. The transformation framework supports all 13 most popular deployment technologies. It offers a CLI, either for user interaction or the integration of the transformation framework into an automated workflow. Each plugin provides the implementation logic for transforming an EDMM model into a technology-specific deployment model of its respective deployment technology. Thereby, the transformation framework can integrate various deployment technologies in an extensible and pluggable way. Internally, the framework represents the EDMM model as a graph, which allows the plugins to traverse the data and apply the transformation logic efficiently. Wurster et al. [WBB+20a] later extended the framework to also support transformations to PaaS- and SaaS-based deployment models, because initially it was only possible to create IaaS-based models.

This work also provides the concept and a prototypical realization of a plugin-based transformation framework, but it performs the transformation in the opposite direction. Still, we build on many of the findings by Wurster et al. [WBB+19] throughout this thesis. For example, we also build a plugin-based transformation framework and use their definitions of mappings between EDMM and the deployment technologies for the implementation of the prototypical realization.

3 Concept and Design

The overall objective of this work is to provide a concept for a transformation framework that can transform technology-specific deployment models into technology-agnostic deployment models. This chapter presents the concept and a more specific design and resulting architecture. First, we give a comprehensive overview of the concept in Section 3.1. Then we provide basic definitions of terms that we use throughout this chapter in Section 3.2. Section 3.4 presents the concept in detail, including the main requirements and design decisions leading to this concept. We then introduce the meta-model we selected for the technology-agnostic deployment model in ???. Finally, we present the architecture for the transformation framework that we designed based on the concept in Section 3.5.

3.1 Overview of the Concept

The transformation framework transforms technology-specific deployment models into technology-agnostic deployment models following a plugin-based approach (see Figure 3.1). It contains several plugins that each can analyze deployment models created with a specific deployment technology into the entities of the technology-agnostic deployment model. For example, the Terraform Plugin in Figure 3.1 can only transform deployment models created with the deployment technology Terraform. When the transformation framework receives a technology-specific deployment model as input, it determines the deployment technology used to create it and passes it to the corresponding plugin. The plugin analyzes the technology-specific deployment model and creates the technology-agnostic deployment model with the information it can find. The transformation framework stores this technology-agnostic deployment model and outputs it at the end of the transformation process.

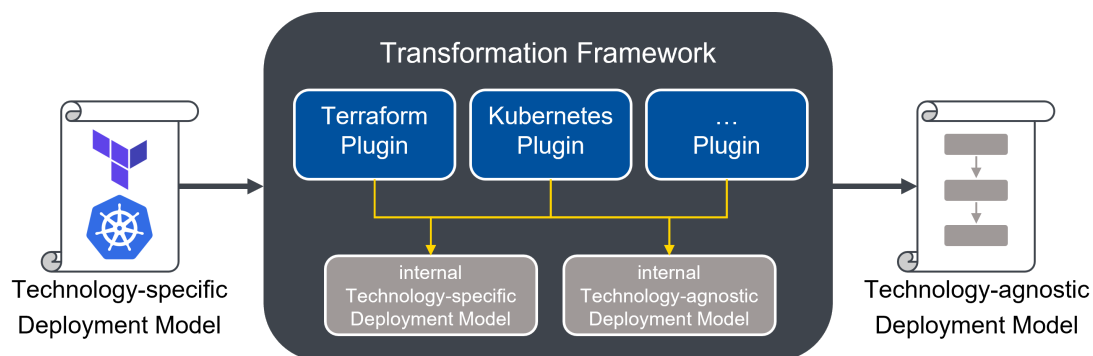


Figure 3.1: High-level concept of the transformation framework.

With this concept, the transformation framework can also transform more complex deployment models that use several deployment technologies. In that case, it distributes work packages for analyzing parts of the technology-specific deployment model across the appropriate plugins. The

plugins now collectively extend the internal technology-agnostic deployment model, adding information about their part of the technology-specific deployment model. Additionally, they also create an internal representation of the technology-specific deployment model. It helps to create a shared understanding of the technology-specific deployment model. This is important because the transformation framework may not know all deployment technologies that were used for the creation of the technology-specific deployment model at the beginning of the transformation process. Suppose plugins find a part of the technology-specific deployment model they do not understand because they were created with a different deployment technology. In that case, they can note this in the internal technology-specific deployment model. The transformation framework can then create a new work package and distribute it to an appropriate plugin. That way, the plugins can communicate relevant findings for subsequent processing steps of other plugins. Furthermore, the transformation framework can track when all parts of the technology-specific deployment model have been analyzed and determine when the transformation process is finished.

3.2 Definitions

In the first part, this section defines deployment models based on the information presented in Section 2.1. This serves as a precise specification for this work to define what we want to analyze and transform with the transformation framework. Furthermore, we define the term *embedded deployment model*, which shows how we handle more complex deployment models that are created by integrating different deployment technologies. In the second part, we define general analysis techniques that the plugins of the transformation framework can apply.

3.2.1 Deployment Model and Embedded Deployment Model

A *deployment model* prescriptively describes the deployment of an application and its required infrastructure. The term comprises scripts, configurations, files, declarative models, and imperative models created with arbitrary technologies to deploy an application automatically.

For example, TOSCA topology templates or EDMM deployment models are declarative deployment models that describe the components as well as their dependencies in the form of a typed and weighted graph. We also consider Terraform configuration files or Kubernetes configuration files as deployment models because they contain object definitions of the components following an object model defined by their DSL. A Bash script is an imperative deployment model if it contains commands for deploying an application.

An *embedded deployment model* is a deployment model that is contained in another deployment model, an *embedding deployment model*. The embedded deployment model prescriptively describes the deployment of a specific part of the embedding deployment model. A deployment model can embed any number of other deployment models. The embedded deployment model is possibly created with a different technology than the embedding deployment model and can also recursively contain further embedded deployment models.

Figure 3.2 visualizes this relation. For example, in a declarative TOSCA topology template, a node template could be of a node type that provides an embedded deployment model in the form of an install management operation with a corresponding implementation artifact, e.g., an imperative Bash script. To give another example, Terraform configuration files deploy a Kubernetes cluster and reference Kubernetes configuration files that deploy the application in this cluster.

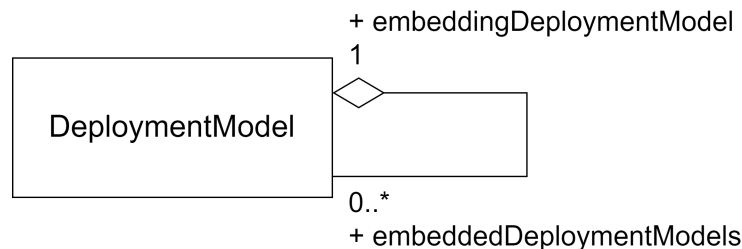


Figure 3.2: Relationship between deployment model and embedded deployment model.

A *source code artifact* implements a component's business logic and may contain additional dependencies required to execute it.

For example, a TOSCA node template that describes a Java application may contain a source code artifact in the form of a Java Archive (JAR) file. Alternatively, a Kubernetes deployment object may reference a source code artifact in the form of a Docker image. Next to the source code, source code artifacts can contain additional dependencies. For example, a JAR file may contain an embedded Apache Tomcat, which is a separate component that hosts the application. Docker images can be created based on other previously built Docker images, like operating systems. Furthermore, it is possible to copy other source code artifacts or embedded deployment models, like imperative scripts, onto the Docker image during its creation. In these cases, source code artifacts are deployment models that we need to analyze.

3.2.2 Analysis Techniques

In general, we identify two different approaches for the analysis and transformation of deployment models: 1. *static analysis* and 2. *dynamic analysis*.

Static analysis parses in the files of the technology-specific deployment model and extracts all the information that it can find. It can understand entities of the DSL of the employed deployment technologies and can directly transform them into entities of the technology-agnostic deployment model. Figure 3.3 shows how an application following a static analysis technique operates. First, this *Static Analysis Application* parses in the files of the technology-specific deployment model (1). It then analyzes the deployment model directly or through an intermediate data model and transforms it into entities of the technology-agnostic deployment model (2). After the Static Analysis Application has analyzed all files, it returns the completed technology-agnostic deployment model (3).

Dynamic analysis on the other hand, deploys the given technology-specific deployment model and then analyzes the running application. It may involve monitoring application metrics, analyzing logs, or running load tests and tracing the requests. From these results, it creates the entities of the technology-agnostic deployment model. Figure 3.4 visualizes this process. In the first step,

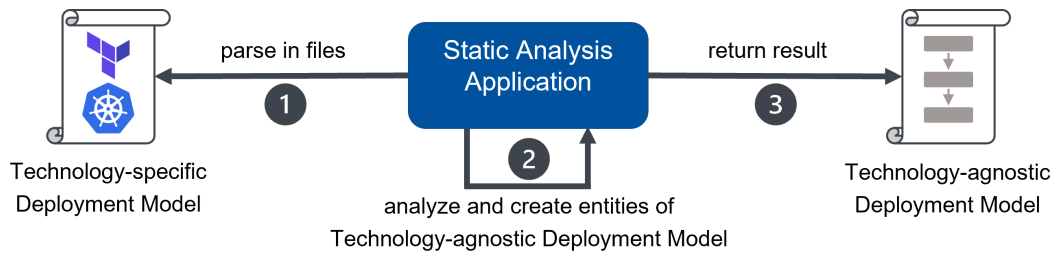


Figure 3.3: Process of transforming a technology-specific deployment model following a static analysis approach.

the *Dynamic Analysis Application* determines the deployment technologies of the technology-specific deployment model (1). This is important to create a suitable deployment environment in the second step (2). The deployment environment is a collection of infrastructure resources and software components required to run the application that the technology-specific deployment model describes. The Dynamic Analysis Application can deploy these resources by installing them on the local machine or using managed services of cloud providers. For example, a Dynamic Analysis Application provisions a Kubernetes cluster if it finds the use of the deployment technology Kubernetes in the technology-specific deployment model. In the third step, the Dynamic Analysis Application deploys the application to this deployment environment by executing the technology-specific deployment model (3). It does that by passing the technology-specific deployment model to an instance of the process or deployment engine of the utilized deployment technology. After that, the Dynamic Analysis Application analyzes the running application and creates the corresponding entities of the technology-agnostic deployment model (4). When it reaches a point where it cannot find any new information in the running application, it returns the completed technology-agnostic deployment model (5). Finally, it destroys the deployment environment by deprovisioning its software components and infrastructure resources (6).

3.3 Meta-Model for the Technology-Agnostic Deployment Model

A specific meta-model must be selected or created for the technology-agnostic deployment model. It should be able to describe all relevant parts of application deployments while being independent of any deployment technology. At the same time, it should be as simple as possible so that the instantiated technology-agnostic deployment model is easy to understand.

We selected EDMM because it fits very well to these requirements. EDMM is based on the essential parts that the most popular deployment technologies support, which was determined by a systematic analysis of those deployment technologies [WBF+20]. That way, it can express a deployment model in a technology-agnostic way while ensuring that technology-specific deployment models can be mapped and transformed to it. It is inspired by the TOSCA standard, and as the name suggests, it provides a meta-model comprised of several entities (see Figure 3.5).

The primary entity is the *Deployment Model* which contains all other entities. The essential building blocks are the *Component* and *Relation*. A Component is a physical, functional, or logical unit of an application, while a Relation is a directed physical, functional, or logical dependency between exactly two of such Components. For both of them, corresponding type entities, the *Component*

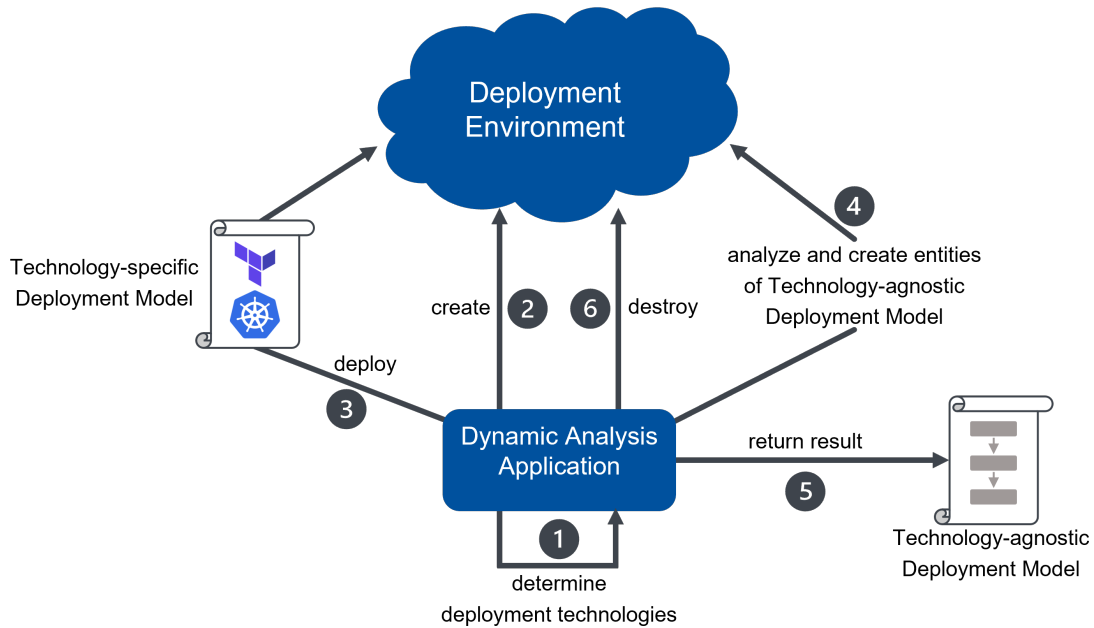


Figure 3.4: Process of transforming a technology-specific deployment model following a dynamic analysis approach.

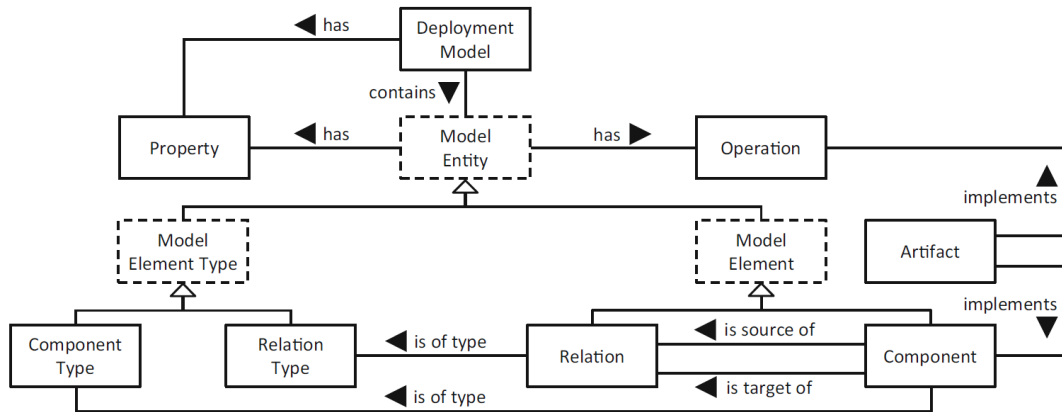


Figure 3.5: The Essential Deployment Metamodel [WBF+20].

Type and *Relation Type* are defined. They are used as templates for *Components* or *Relations* and are therefore defined across *Deployment Models* for better reusability. These entities can have *Properties* to express further configuration information. Additionally, they can have *Operations* which describe executable procedures associated with an entity, for example, an installation command. *Operations* and *Components* can be implemented by an *Artifact*. An *Artifact* is a physical piece of information like an installation script that an *Operation* may reference or a container image that contains the source code of a *Component*.

Hence, a deployment model created with EDMM serves not only as an overview of the application deployment but also contains all the information necessary to deploy the described application.

3 Concept and Design

```
1 ---
2 components:
3   pet_clinic:
4     type: web_application
5     properties:
6       db_hostname: "${db.public_address}"
7       db_user: "${db.user}"
8       db_password: "${db.password}"
9       db_schema: "${db.schema_name}"
10    artifacts:
11      - war: "./files/app.war"
12    operations:
13      configure: "./files/configure.sh"
14    relations:
15      - hosted_on: pet_clinic_tomcat
16  pet_clinic_tomcat:
17    type: web_server
18    operations:
19      create: "./files/create.sh"
20      start: "./files/start.sh"
21
22  component_types:
23    base:
24      extends: null
25      description: The base type
26      metadata: {}
27    web_server:
28      extends: base
29      properties:
30        port:
31          type: integer
32          default_value: 80
33    web_application:
34      extends: base
35
36  relation_types:
37    depends_on:
38      extends: null
39    hosted_on:
40      extends: depends_on
41    connects_to:
42      extends: depends_on
```

Listing 3.1: Example of a deployment model created with EDMM in YAML specification.

To instantiate an EDMM and therefore create a deployment model, the EDMM GitHub project provides a specification for a concrete syntax in the YAML format¹. Using this specification, they also created some example deployment models². Listing 3.1 shows a shortened version of one of those examples. It contains three main sections for components, component types, and relation types. There are two components, the *pet_clinic* component of type *web_application* and the *pet_clinic_tomcat* component of type *web_server*. Both types are defined in the section for the component types and extend the *base* component type. There is also one relation of type *hosted_on*, that is specified inside the *pet_clinic* component. It indicates, that the *pet_clinic* component is hosted on the *pet_clinic_tomcat* component.

3.4 Concept of the Transformation Framework

This section describes the most important requirements and elaborates on the design decisions that lead to the presented concept of the transformation framework.

3.4.1 Requirements

The transformation framework should fulfill the following key requirements:

Requirement 1 The transformation framework shall be able to transform a technology-specific deployment model into a technology-agnostic deployment model based on EDMM.

Requirement 2 The resulting technology-agnostic deployment model shall contain all information about the application deployment from the originating technology-specific deployment model so that no information about the application deployment is lost.

Requirement 3 The transformation framework shall be extensible so that it is possible to add support for more deployment technologies or alternative analysis techniques dynamically and with low effort.

Requirement 4 The transformation framework shall be able to transform technology-specific deployment models created from a combination of different deployment technologies.

If the transformation framework fails to find any information about the application deployment, the technology-agnostic deployment model does not correctly represent the originating technology-specific deployment model. In that case, the transformation process has failed. The same applies when the transformation framework adds incorrect information. Therefore, the completeness of the technology-agnostic deployment model is a critical factor for the success of the transformation framework. Consequently, it is not required to achieve short processing times for the transformation process.

Another important factor is the number of supported deployment technologies. The transformation framework cannot transform deployment technologies that use unsupported deployment technologies. However, there are many more deployment technologies than the 12 most popular ones described

¹EDMM in YAML Specification: <https://github.com/UST-EDMM/spec-yaml>

²EDMM examples: <https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/test/resources/templates>

by Wurster et al. [WBF+20]. If we include the analysis of source code artifacts, the amount of technologies that need to be supported is even higher. Therefore, we do not plan to cover every existing deployment technology from the start. Moreover, we plan to extend the transformation framework to support more deployment technologies over time, starting with the most popular ones. Therefore, the required time and effort to integrate the transformation logic for new deployment technologies should be as low as possible. This also benefits testing alternative analysis techniques, e.g., when comparing variations or combinations thereof to improve the transformation result.

3.4.2 Concept

Important information about the application deployment in a technology-specific deployment model may be hidden behind the peculiarities of the employed deployment technologies, especially their DSL. Because deployment technologies differ heavily in that regard, it is inescapable to run the analysis process in a technology-specific way. This ensures that the transformation framework retrieves all information and fulfills Requirement 2. Therefore, the transformation framework follows a plugin-based approach so that each plugin provides the logic to transform deployment models written with one specific deployment technology into the technology-agnostic deployment model (see Figure 3.1). Each plugin is a self-contained and decoupled service that can easily be added to and removed from the transformation framework. As a result, the transformation framework is extensible regarding the support of more deployment technologies to fulfill Requirement 3.

To fulfill Requirement 4, the transformation framework can determine the applied deployment technologies in a technology-specific deployment model. It can distribute tasks for analyzing a specific part of the technology-specific deployment model to appropriate plugins. For that purpose, it creates an internal representation of the technology-specific deployment model that it shares across all plugins (see Figure 3.1). The plugins continuously extend it with new information they find during the transformation process. The internal technology-specific deployment model contains additional meta-information, like the location of associated files, but does not contain the actual information about the application deployment. It represents a common understanding of the structure of the technology-specific deployment model and its embedded deployment models, which helps to manage and track the transformation process for more complex deployment models. Additionally, the transformation framework contains an internal technology-agnostic deployment model that the plugins continuously extend with new-found information (see Figure 3.1). The data format of the internal technology-agnostic deployment model can differ from the format of the technology-agnostic deployment model that the transformation framework outputs. We can select a suitable data format for each use case when realizing the transformation framework.

To assess the quality of the resulting technology-agnostic deployment model during and at the end of the transformation process, we need a way to measure it. The quality of the technology-agnostic deployment model refers to Requirement 2 and should state how sure we are that it includes all information about the application deployment. This results from the consideration of several factors:

1. How much of the technology-specific deployment model were we able to analyze?
2. How good do we comprehend the technology-specific deployment model?

3. How confident are we that the found information in the technology-agnostic deployment model is correct?

When the transformation framework has analyzed and correctly comprehended the technology-specific deployment model and is confident that all found information is correct, we assume that the resulting technology-agnostic deployment model has high quality. We define metrics that the transformation framework records in the internal representations of the technology-specific deployment model and the technology-agnostic deployment model. When extending the internal deployment models during the transformation process, the plugins determine and record the metrics accordingly.

The internal technology-specific deployment model contains the following metrics:

Analysis Progress Indicates if the transformation framework has analyzed the given part of the technology-specific deployment model. This is measured by a Boolean value.

Comprehensibility Gives the proportion of the technology-specific deployment model that the transformation framework can comprehend. This is measured in percent.

In the internal technology-agnostic deployment model, the plugins record the following metrics:

Confidence Measures how sure the transformation framework is that a given entity of the technology-agnostic deployment model is part of the application deployment described in the technology-specific deployment model. This metric has either the value “suspected” or “confirmed”. Confirmed means that a plugin is sure that the entity is part of the application deployment. A plugin can create an entity and mark it as suspected if it has not found any evidence about its existence but assumes that it is part of the application deployment. For example, suppose a plugin has found two components of type “database” and “web_application” respectively that share the same name. In that case, it may add a relation between those components and mark it as suspected, although it has not found any evidence about the existence of such a relation.

Type Completeness Measures the number of properties and operations that the transformation framework found for a component or relation entity in the technology-agnostic deployment model, in proportion to the number of required properties and operations that their component or relation type define. This metric is enabled through the type system of EDMM. Plugins specify generic component and relation types that contain properties and operations required for a component or relation of this type. We can calculate the type completeness of components and relations by comparing the found properties and operations to those required by its type. The plugins do not need to record this metric as an additional field in the internal technology-agnostic deployment model. Instead, the transformation framework can directly calculate it from the given entities.

We describe the implementation of these metrics in more detail in Section 3.5.

Different analysis techniques and methods are combined to ensure the extracted information’s completeness further. For example, the transformation framework may contain a Terraform plugin employing a static analysis technique and at the same time also provide another Terraform plugin that transforms deployment models using a dynamic analysis technique. Both plugins can be used and combined in the same transformation process.

A concept for the transformation process is presented in Figure 3.6. It shows the partition of the transformation process into four major transformation phases. The deployment model and all embedded deployment models are recursively analyzed separately by repeating those four transformation phases.

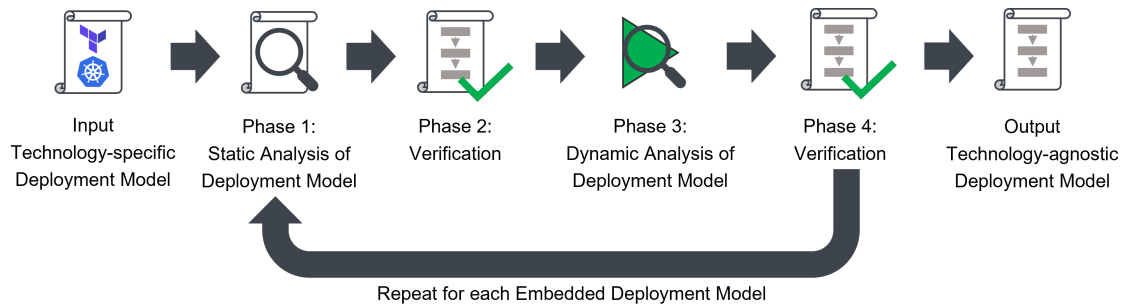


Figure 3.6: Phases of the transformation process.

For the current deployment model, the transformation framework identifies the deployment technology. It selects a corresponding plugin for the first phase, which carries out a static analysis of the deployment model. If the transformation framework does not contain an appropriate plugin, the transformation process continues with Phase 3. Otherwise, the plugin analyzes the given technology-specific deployment model and saves the result in the internal models of the transformation framework. This includes recording the previously described metrics.

After that follows Phase 2: Verification. It reviews if the technology-agnostic deployment model meets the internal requirements of the transformation framework, defined by the recorded metrics in the internal models. The transformation framework can assess how well the transformation process has worked so far from these metrics. Based on this, the transformation framework decides how to proceed with Phase 3.

In this next phase, the transformation framework analyzes the same deployment model using a dynamic analysis technique. It extends the technology-agnostic deployment model created during Phase 1 with newly found information. It ensures that we find more information about the application deployment, especially when the preceding static analysis delivered poor results. Alternatively, the transformation framework can skip Phase 3 and directly proceed with analyzing the next deployment model or outputting the technology-agnostic deployment model if it has analyzed every deployment model. It selects this option when the quality of the current technology-agnostic deployment model is high enough, meaning the static analysis worked very well and the additional dynamic analysis would be unnecessary. Alternatively, it skips Phase 3 if there is no plugin supporting the dynamic analysis of the corresponding deployment technology.

In Phase 4, the transformation framework verifies the technology-agnostic deployment model again and in the same way as in Phase 2. If it ran Phase 3, the transformation framework could see if the dynamic analysis improved the result.

When the transformation framework has analyzed every deployment model, it calculates an aggregated score for each metric. It then removes the recorded metric values from the technology-agnostic deployment model and outputs it alongside the calculated scores.

3.5 Architecture

From the concept, we designed a concrete architecture for the transformation framework. This section first gives an overview of the architecture by describing the components of the transformation framework. After that, it provides definitions for the internal deployment models, which are the internal technology-specific deployment model and the internal technology-agnostic deployment model, and the calculations of the metrics that the plugins record. The transformation framework executes two processes: The plugin registration process and the transformation process. We describe these processes in detail at the end.

3.5.1 Overview

Figure 3.7 shows an overview of the different components that make up the transformation framework. It contains three different kinds of applications (*AnalysisManager*, *ModelsService*, and *Plugins*), three databases (*ModelsDatabase*, *ConfigurationsDatabase*, and *TasksDatabase*), and two different kinds of messaging queues (*AnalysisTaskRequestQueues* and *AnalysisTaskResponseQueue*).

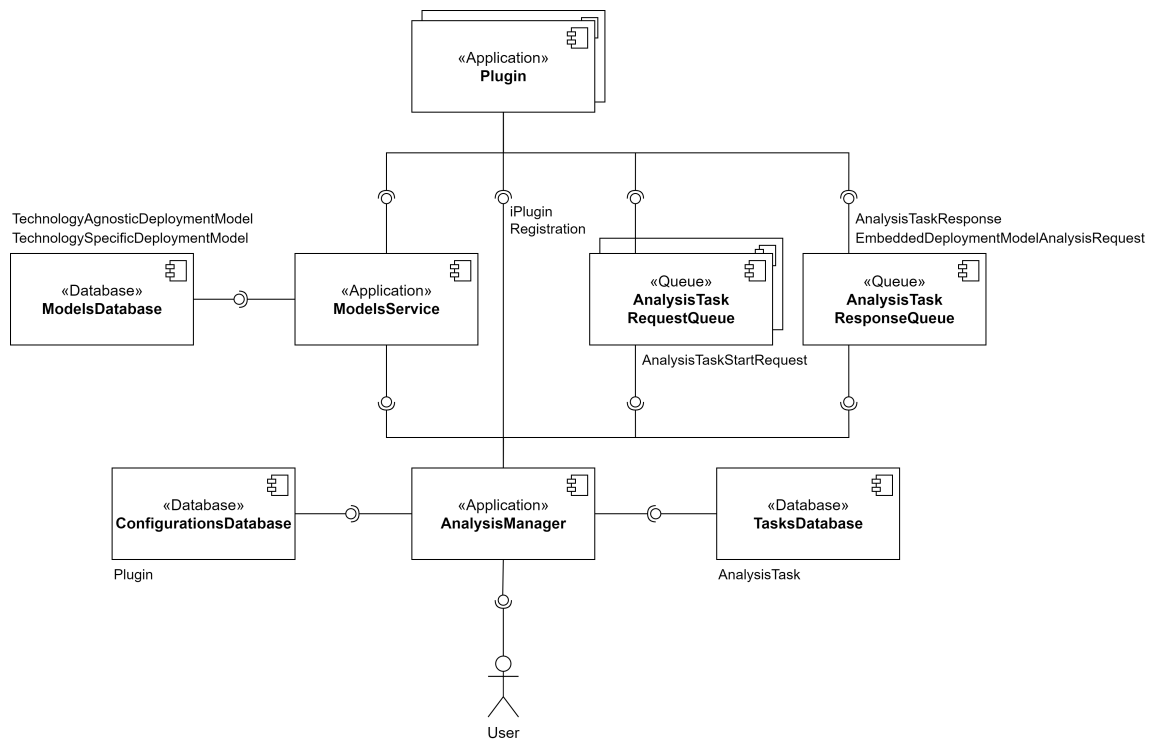


Figure 3.7: Component diagram of the transformation framework. The databases and queues show additional information about which entities they store. There can be multiple instances of the components *Plugin* and *AnalysisTaskRequestQueue*.

The *AnalysisManager* is one of the main components of the transformation framework and is responsible for the user interaction and management of the transformation process. Analogous to the previously presented concept, there can be many *Plugins* in the transformation framework. The *Plugins* carry out the actual transformation logic. They register at the *AnalysisManager* through

the *iPluginRegistration* interface, which is a Representational State Transfer (REST) endpoint. The AnalysisManager stores information about registered Plugins in the ConfigurationsDatabase. Thereby, the AnalysisManager can distribute tasks to analyze a given deployment model across the plugins. The AnalysisManager communicates with the Plugins through asynchronous messaging that the two queues enable. Each Plugin has its own AnalysisTaskRequestQueue, so that the number of Plugins is equal to the number of AnalysisTaskRequestQueues in the transformation framework. To kick off the transformation process, the AnalysisManager creates AnalysisTasks, stores them in the TasksDatabase to keep track of them, and sends corresponding AnalysisTaskStartRequest messages to a selected AnalysisTaskRequestQueue. Plugins consume AnalysisTaskStartRequest messages from their AnalysisTaskRequestQueue and then run their transformation logic. When they are finished with the transformation, they respond by sending an AnalysisTaskResponse message to the AnalysisTaskResponseQueue. The AnalysisManager consumes this message and continues with the overall transformation process. Alternatively, a Plugin may find an embedded deployment model during the analysis that it cannot analyze because it uses a different deployment technology. In that case, it pauses its analysis and sends an EmbeddedDeploymentModelAnalysisRequest message to the AnalysisTaskResponseQueue. The AnalysisManager consumes this message, creates an appropriate AnalysisTask, and sends a corresponding AnalysisTaskStartRequest message to the AnalysisTaskRequestQueue of a Plugin that can analyze the corresponding deployment technology. When this task has finished, the AnalysisManager sends an AnalysisTaskResumeRequest message to the AnalysisTaskRequestQueue of the Plugin that has initially detected the embedded deployment model and stopped its analysis. The Plugin will then resume its analysis task. Every time the AnalysisManager consumes an AnalysisTaskResponse message, it checks if it still needs to run additional tasks or resume stopped ones. The transformation process is finished when the plugins have processed all AnalysisTasks. The AnalysisManager then returns the resulting technology-agnostic deployment model to the User.

The ModelsService manages the internal deployment models, the TechnologyAgnosticDeploymentModel and the TechnologySpecificDeploymentModel, and stores are stored in the ModelsDatabase and managed by the ModelsService. The ModelsService serves as an abstraction layer between the internal deployment models and the Plugins. It can resolve conflicts between the Plugins while providing a common interface to the internal data models for the whole transformation framework. Consequently, Plugins can use this interface and do not need to implement the same functionality for creating the entities of the internal deployment models. At the same time, the ModelsService provides access to the AnalysisManager for initializing the internal deployment models and retrieving the metrics and the result.

3.5.2 Internal Deployment Models

The internal deployment models are the internal technology-specific deployment model and the internal technology-agnostic deployment model. In the following, we first introduce the meta-model for the internal technology-specific deployment model and define how we calculate the analysis progress and comprehensibility metrics from this model. Then we present the meta-model for the internal technology-agnostic deployment model, which we extended from the EDMM to store information for internal processing. This enables the calculation of the confidence and type completeness metrics, which we define afterward.

Internal Technology-Specific Deployment Model

To keep track of the transformation process and create a common understanding of the technology-specific deployment model across all plugins, an internal representation needs to be created that provides meta-information about the technology-specific deployment model. This representation is called the internal technology-specific deployment model and is presented in Figure 3.8. It is continuously created and extended throughout the transformation process by the different plugins.

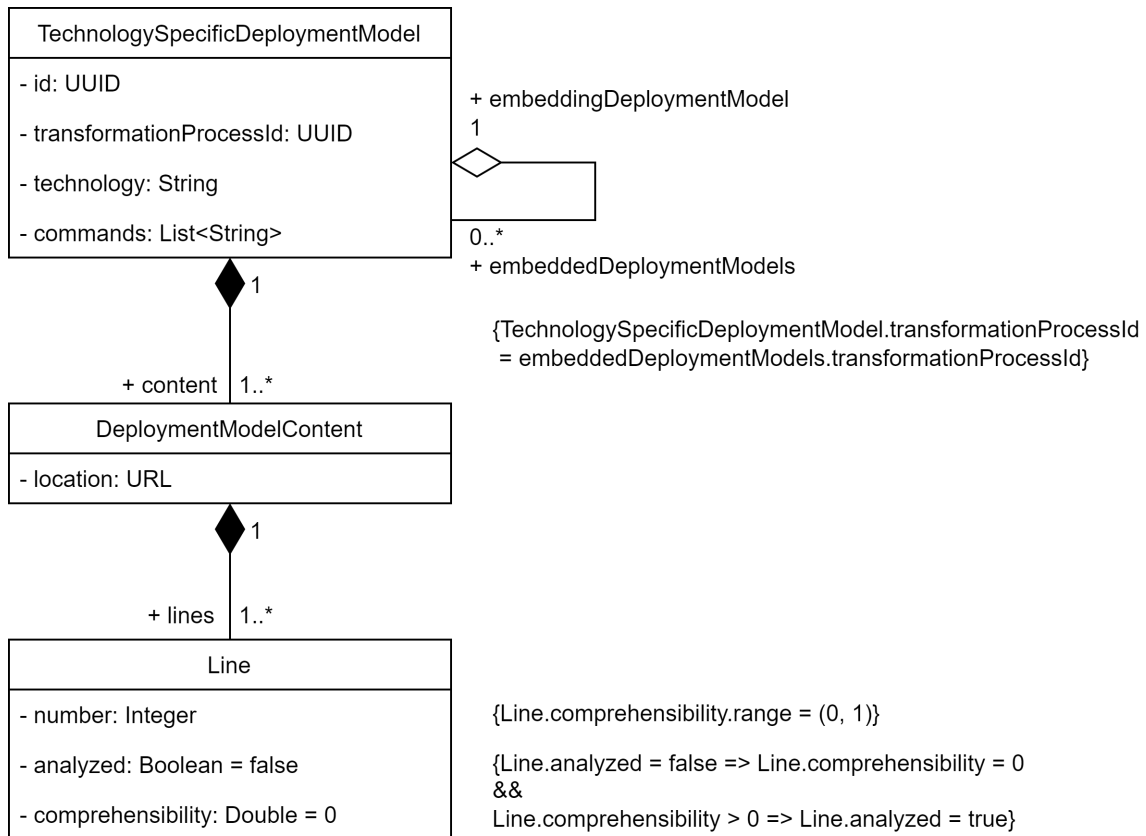


Figure 3.8: UML Class diagram of the internal representation of the technology-specific deployment model.

The *TechnologySpecificDeploymentModel* is the main entity. It can be uniquely identified by the *id* field and matched to a specific transformation process through the *transformationProcessId* field. Similar to our definition given in Figure 3.2, it can contain an arbitrary number of *embeddedDeploymentModels*, which are other *TechnologySpecificDeploymentModel* entities that must share the same *transformationProcessId*. The *technology* field of the *TechnologySpecificDeploymentModel* entity indicates the deployment technology that was used to create it. The transformation framework utilizes it to match suitable plugins for transforming the technology-specific deployment model. The *commands* field contains a set of commands that execute operations on the deployment model. It is essential for some deployment technologies because they may hold information on how developers intend to use the deployment model. For example, they may provide specific values for variables in the deployment model. Users of the transformation framework can provide these commands.

Alternatively, Plugins may find them in an embedding deployment model. If this is not the case, the `commands` field may be left empty, or the transformation framework may add a default command for the corresponding deployment technology.

Meta information about the content of the technology-specific deployment model is expressed through the entities *DeploymentModelContent* and *Line*. The *DeploymentModelContent* represents a file or a part of a file and is identified by the *location* field. This is a Uniform Resource Locator (URL) pointing to a file on the local file system or a web resource. A *TechnologySpecificDeploymentModel* can contain several *DeploymentModelContents*, distributed over different locations. The *DeploymentModelContent* consists of one or more *Lines* in sequential order. Each *Line* is identified by its position in this sequence through the field *number*. Additionally, it stores the values for calculating the analysis progress and comprehensibility metrics in the *analyzed* and *comprehensibility* fields, respectively. The *analyzed* field indicates if a plugin has analyzed the given *Line* or not. On the other hand, the *comprehensibility* field shows how well a plugin could understand the content of the technology-specific deployment model that the given *Line* refers to. This is expressed in percent through a floating-point number between zero and one. “Zero” means that the content of the *Line* was not understood, while “one” means that the *Line* was fully understood. Depending on the plugin’s implementation, it may split the content of a *Line* into several parts for analyzing it. For example, it may split a Bash command into several tokens for each given parameter. In these cases, a plugin can record that it has partly understood the content of a *Line*, e.g., only a specific proportion of the tokens. By default, plugins set the *comprehensibility* to zero. Further analysis by other plugins can only increase *comprehensibility*, not decrease it. If no plugin has analyzed the content of a *Line*, the corresponding *comprehensibility* field must have the value “zero” (not understood). Vice versa, if the *comprehensibility* indicates that the content of the *Line* is at least partially understood, it must have been analyzed by some plugin.

The following sections further explain how the transformation framework can use these values to calculate the corresponding metrics for the whole internal technology-specific deployment model.

Analysis Progress The analysis progress metric is set per *Line* by the plugins in the “*analyzed*” field. By computing it for overlying entities of the type *DeploymentModelContent* and *TechnologySpecificDeploymentModel*, the transformation framework can track the progress of the transformation process. Furthermore, it can see which parts of the technology-specific deployment model need to be analyzed. Depending on the entity type, the analysis progress metric has a different meaning. For a *Line* entity, plugins record it as a Boolean value, which means it has either analyzed the *Line* or not. In the context of a *DeploymentModelContent* or *TechnologySpecificDeploymentModel* entity, it stands for the proportion of contained *Lines* that have been analyzed.

The following formulas show how to calculate the analysis progress for a *DeploymentModelContent* (Formula 2) and a *TechnologySpecificDeploymentModel* (Formula 5), using helper functions defined in the remaining formulas:

Definitions The following abbreviations are used for instances of entities in the internal technology-specific deployment model:

$tSDM := technologySpecificDeploymentModel$
 $dMC := deploymentModelContent$
 $eDMs := embeddedDeploymentModels$

Formula 1 Function to calculate the sum of all analyzed Lines contained in a DeploymentModelContent, called the “analyzedLinesSum”:

$$analyzedLinesSum(dMC) = \sum_{n=1}^{|dMC.lines|} a_n = \begin{cases} 0, & \text{if } dMC.lines_n.analyzed = false \\ 1, & \text{if } dMC.lines_n.analyzed = true \end{cases}$$

Formula 2 Formula 1 can be used to calculate the analysis progress of a DeploymentModelContent by dividing through the amount of contained Lines:

$$dMC.analysisProgress = \frac{analyzedLinesSum(dMC)}{|dMC.lines|}$$

Formula 3 Function to calculate the “analyzedLinesSum” for a TechnologySpecificDeploymentModel by recursively calling itself for all contained embeddedDeploymentModels:

$$\begin{aligned}
 calculateAnalyzedLinesSum(tSDM) = & \\
 \sum_{n=1}^{|tSDM.content|} analyzedLinesSum(tSDM.content_n) & \\
 + \sum_{n=1}^{|tSDM.eDMs|} calculateAnalyzedLinesSum(tSDM.eDMs_n) &
 \end{aligned}$$

Formula 4 Function to calculate the total amount of Lines contained in a TechnologySpecificDeploymentModel by recursively calling itself for all contained embeddedDeploymentModels:

$$\begin{aligned}
 calculateLineCount(tSDM) = \sum_{n=1}^{|tSDM.dMC|} |tSDM.dMC_n.lines| & \\
 + \sum_{n=1}^{|tSDM.eDMs|} calculateLineCount(tSDM.eDMs_n) &
 \end{aligned}$$

Formula 5 Finally, calculate the analysis progress of a TechnologySpecificDeploymentModel by dividing its “analyzedLinesSum” through the total amount of Lines contained in the TechnologySpecificDeploymentModel:

$$tSDM.analysisProgress = \frac{calculateAnalyzedLinesSum(tSDM)}{calculateLineCount(tSDM)}$$

Comprehensibility The transformation framework records the comprehensibility metric in the field “comprehensibility” of the Line entity (see Figure 3.8). From this, it can calculate the metric for each DeploymentModelContent entity, each embeddedDeploymentModel entity, and the whole TechnologySpecificDeploymentModel entity. It describes how well the transformation framework understands the part of the original technology-specific deployment model that corresponds to the given entity. The comprehensibility metric is given in percent, based on the understood parts. When a plugin analyzes a Line, it can either comprehend it entirely, partially, or not at all.

The following formulas show how to calculate the comprehensibility of a TechnologySpecificDeploymentModel (Formula 5) and a DeploymentModelContent (Formula 2). For that, we define helper functions in the remaining formulas.

Definitions The following abbreviations are used for instances of entities in the internal technology-specific deployment model:

$tSDM := technologySpecificDeploymentModel$
 $dMC := deploymentModelContent$
 $eDMs := embeddedDeploymentModels$

Formula 1 Function to calculate the sum of comprehensibility values of all Lines contained in a DeploymentModelContent, called the “comprehensibilitySum”:

$$comprehensibilitySum(dMC) = \sum_{n=1}^{|dMC.lines|} dMC.lines_n.comprehensibility$$

Formula 2 Formula 1 can be used to calculate the comprehensibility of a DeploymentModelContent by dividing through the amount of contained Lines:

$$dMC.comprehensibility = \frac{comprehensibilitySum(dMC)}{|dMC.lines|}$$

Formula 3 Function to calculate the “comprehensibilitySum” for a TechnologySpecificDeploymentModel by recursively calling itself for all contained embeddedDeploymentModels:

$$\begin{aligned} calculateComprehensibilitySum(tSDM) = & \\ \sum_{n=1}^{|tSDM.content|} & comprehensibilitySum(tSDM.content_n) \\ + \sum_{n=1}^{|tSDM.eDMs|} & calculateComprehensibilitySum(tSDM.eDMs_n) \end{aligned}$$

Formula 4 Function to calculate the total amount of Lines contained in a TechnologySpecificDeploymentModel by recursively calling itself for all contained embeddedDeploymentModels:

$$\begin{aligned} calculateLineCount(tSDM) = & \sum_{n=1}^{|tSDM.dMC|} |tSDM.dMC_n.lines| \\ + \sum_{n=1}^{|tSDM.eDMs|} & calculateLineCount(tSDM.eDMs_n) \end{aligned}$$

Formula 5 Finally, calculate the comprehensibility of a TechnologySpecificDeploymentModel by dividing its “comprehensibilitySum” through the total amount of Lines contained in the TechnologySpecificDeploymentModel:

$$tSDM.comprehensibility = \frac{calculateComprehensibilitySum(tSDM)}{calculateLineCount(tSDM)}$$

Internal Technology-Agnostic Deployment Model

As stated in Section 3.3, the transformation framework should create and output a technology-agnostic deployment model following the EDMM. For the internal technology-agnostic deployment model, we present a meta-model in Figure 3.9, which is based on EDMM and can store additional information that we need for the internal processing.

All the main entities of the EDMM are still in place (see Figure 3.5). We rename the DeploymentModel entity into *TechnologyAgnosticDeploymentModel* to distinguish it clearly from the technology-specific deployment models. The fields marked in blue are only needed during the transformation process and the ModelsService removes them when outputting the technology-agnostic deployment model. The *transformationProcessId* in the TechnologyAgnosticDeploymentModel entity allows the transformation framework to assign the internal technology-agnostic deployment model to a specific transformation process. Furthermore, the Property, Operation, Relation, Component, and Artifact entities contain a field for storing the confidence metric.

Additionally, we add fields to the entities of the EDMM because we need this information to instantiate a concrete model. We obtained this information from the textual description of the EDMM by Wurster et al. [WBF+20] and especially their specification for a concrete syntax in the YAML format³. Each Property represents a key-value pair and has corresponding fields to store

³EDMM in YAML Specification: <https://github.com/UST-EDMM/spec-yaml>

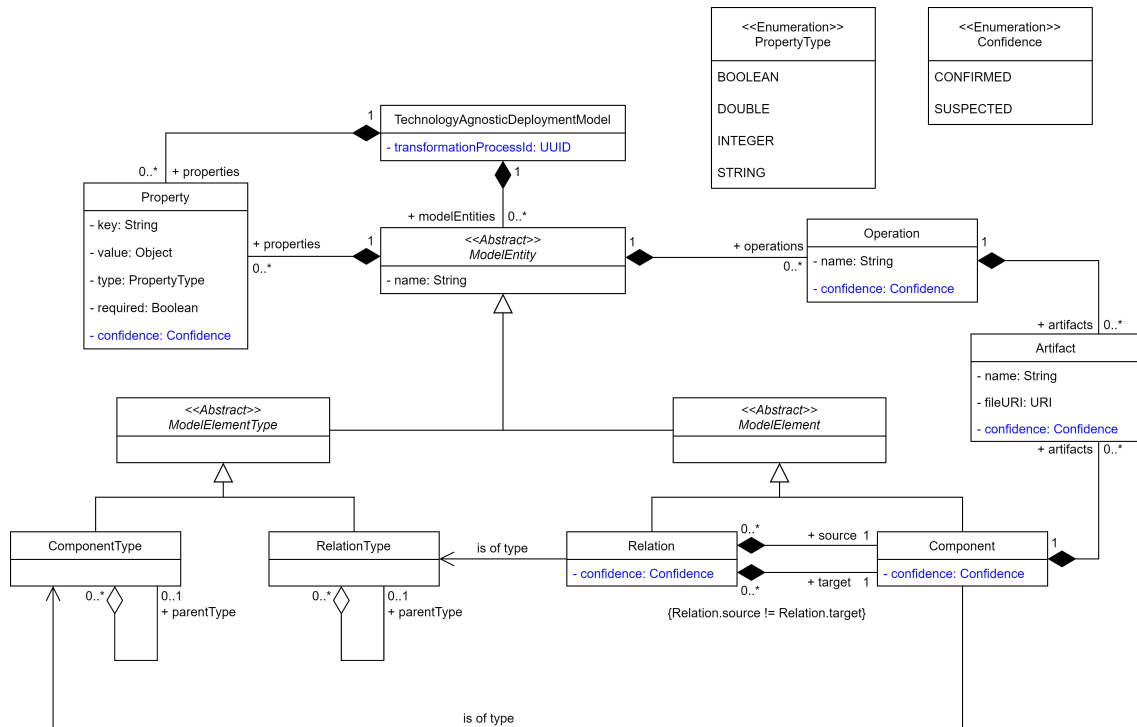


Figure 3.9: UML Class diagram of the internal technology-agnostic deployment model. The original EDMM is extended with information for internal processing, marked in blue.

this information. If the Property is contained in a ComponentType or RelationType, we can use the *value* field to express a default value for the Property. The data type of the value is variable and defined per Property by its *type* field, which can either be a Boolean, double, integer, or string. With the *required* field, we can indicate if a Component or Relation must include this Property or if it is optional. Entities of type ModelEntity, Operation, and Artifact contain a field *name* to differentiate between entities of the same type. Artifacts have a field *fileURI* that stores a Uniform Resource Identifier (URI) to the actual file that an Artifact is representing. We add an aggregation to the ComponentType and the RelationType for specifying a *parentType*. It enables reusing type definitions for building type hierarchies and leads to more concise technology-agnostic deployment models. An example for this can be seen in Listing 3.1 where the parentType is expressed through the *extends* field. Finally, we add a constraint, which defines that a Component cannot have a Relation to itself. This is based on its definition, stating that a “relation is a [...] dependency between exactly two components” [WBF+20].

We can calculate the confidence and type completeness metrics from the internal technology-agnostic deployment model. The latter does not require an additional field as the confidence metric does. Instead, the transformation framework can calculate it by comparing the Relations and Components with their assigned types. We present the calculations for both metrics in the following.

Confidence The confidence metric gives us a better understanding of how many entities in the technology-agnostic deployment model we can be sure that they are part of the application deployment. To calculate this, the transformation framework computes the percentage of AnnotatedProp-

erties, AnnotatedOperations, AnnotatedRelations, AnnotatedComponents, and AnnotatedArtifacts that have a confidence set to “confirmed”. We do this by dividing the sum of all confirmed entities by the sum of all entities, as shown in this formula:

$$confidence = \frac{sumOf ConfirmedEntities}{sumOf Entities}$$

The plugins do not record the confidence for all types of entities in the internal technology-agnostic deployment model. The ComponentType and RelationType entities do not represent actual information about the application deployment because they only serve as templates for Relations and Components. For the DeploymentModel entity, it makes no sense to record the confidence either because there is always only one instance per internal technology-agnostic deployment model. Therefore, we exclude these three entities from calculations of the confidence metric.

Type Completeness We can calculate the type completeness metric for each Relation and Component. It represents the percentage of required Properties and Operations, given by the type definition, present in a given Relation or Component. For that, we need to sum up the required Properties and Operations of the ComponentType or RelationType. This gives us the “sumOfRequiredPropertiesAndOperations”. Then, we need to look at the Component or Relation and count the number of these required Properties and Operations that are present, resulting in the “sumOfActualPropertiesAndOperations”. We ignore optional Properties. By dividing the sumOfActualPropertiesAndOperations by the sumOfRequiredPropertiesAndOperations for a given entity, we can calculate the type completeness:

$$entity.typeCompleteness = \frac{sumOf ActualPropertiesAndOperations(entity)}{sumOf RequiredPropertiesAndOperations(entity.type)}$$

The type completeness of the whole technology-agnostic deployment model (abbreviated in the following formulas with “tadm”) is composed of two values. The first value represents the percentage of required Properties and Operations in relation to the present ones in all Relations and Components for the whole technology-agnostic deployment model combined. This gives a better understanding about the ability of the plugins to find all the required information. Its calculation is similar to the calculation for each entity on its own, as shown above. The difference is that we do not calculate the sumOfRequiredPropertiesAndOperations and sumOfActualPropertiesAndOperations for each entity but rather as a sum over all entities.

$$tadm.typeCompletenessVal1 = \frac{sumOf ActualPropertiesAndOperations(tadm)}{sumOf RequiredPropertiesAndOperations(tadm)}$$

The second value gives the percentage of Components and Relations that are type complete. From an excellent result (close to “1”), we can see that almost all entities are type complete, while a lower result (close to “0”) indicates that we miss information in almost all entities. This means, that they contain all required Properties and Operations, so that $typeCompleteness(entity) = 1$. We show this in the following formulas:

$$\begin{aligned} numberOfTypeCompleteEntities(tadm) = & \\ & |tadm.relations \text{ where } typeCompleteness(relation) == 1| \\ & + |tadm.components \text{ where } typeCompleteness(component) == 1| \\ tadm.typeCompletenessVal2 = & \frac{numberOfTypeCompleteEntities(tadm)}{|tadm.relations|+|tadm.components|} \end{aligned}$$

From both of these values combined, we can assess the general ability of the plugins to find information in the technology-specific deployment model. If both values are high, we can infer that the plugins were very good at finding information, whereas low results indicate the opposite. More interesting are the cases when both values show significant differences in their results. If the `typeCompletenessVal1` is much lower in relation to the `typeCompletenessVal2`, some entities may miss almost all the required information. When the `typeCompletenessVal1` is much higher than the `typeCompletenessVal2`, we can interpret that most entities miss at least some information.

3.5.3 Plugin Registration Process

The `AnalysisManager` distributes tasks for transforming the technology-specific deployment model over the `Plugins` of the transformation framework. For that purpose, the `AnalysisManager` needs to know which `Plugins` are available, especially their deployment technology and the type of analysis technique they support. The plugin registration process enables this (see Figure 3.10). The `Plugins` send all necessary information to the `AnalysisManager` when we deploy them. We can deploy the `Plugins` alongside the `AnalysisManager` and the other entities of the transformation framework. However, it is also possible to add `Plugins` at a later point in time to an already running instance of the transformation framework.

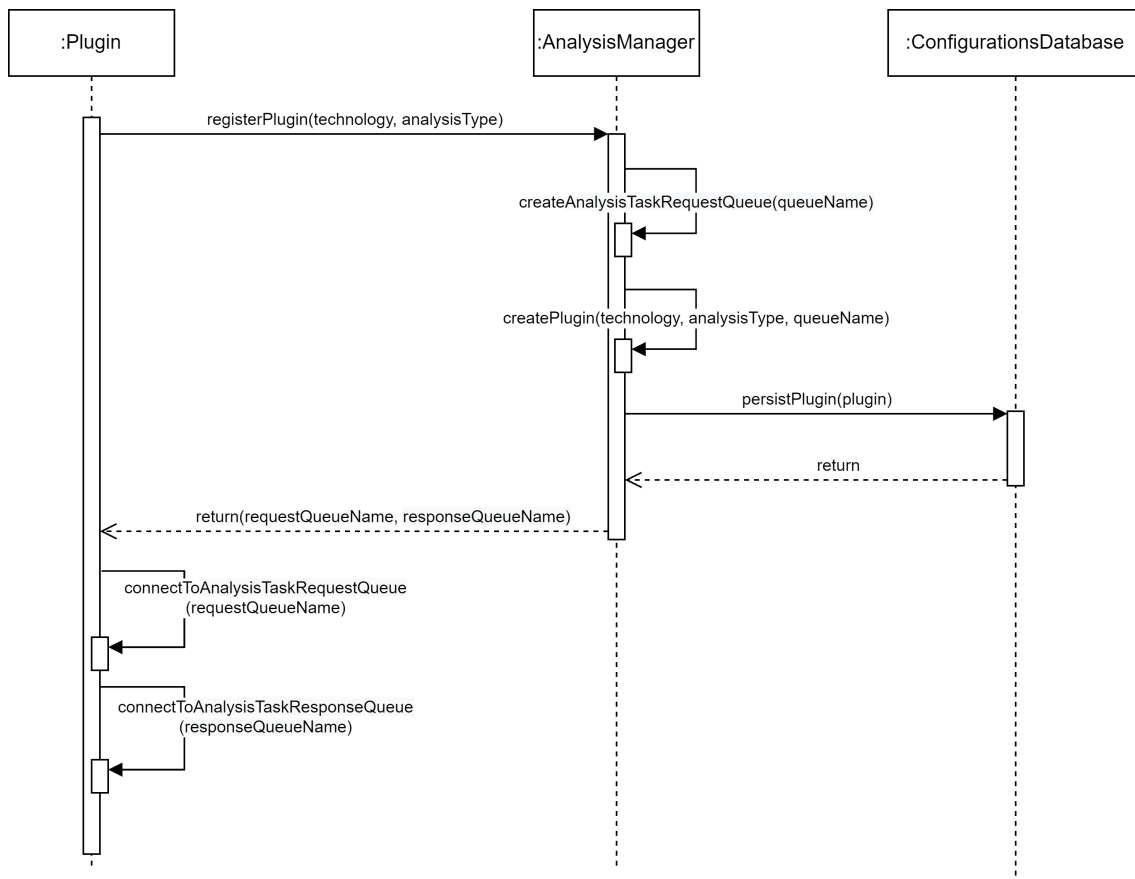


Figure 3.10: UML Sequence diagram showing the plugin registration process.

In the first step of the plugin transformation process, a Plugin calls the corresponding endpoint of the AnalysisManager. It provides information about the deployment technology and the analysis type. The AnalysisManager then first creates and provisions a new AnalysisTaskRequestQueue for the Plugin. It saves the name of the queue alongside the technology and the analysis type in an entity called *Plugin* and persists it in the ConfigurationsDatabase. A UML class diagram for this Plugin entity is presented in Figure 3.11. The field *analysisType* has either the value “STATIC” or “DYNAMIC” corresponding to the two different analysis techniques defined in Section 3.2.2. Persisting the *queueName* is essential because it later provides the information for the AnalysisManager on where to send an analysis task when it selects this Plugin.

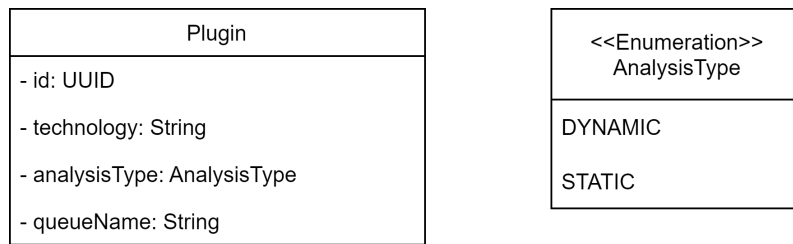


Figure 3.11: UML Class diagram for the Plugin. It is created by the AnalysisManager and persisted in the ConfigurationsDatabase during the plugin registration process.

The AnalysisManager then returns the name of the newly created AnalysisTaskRequestQueue and the name of the already running AnalysisTaskResponseQueue. When we deploy the AnalysisManager, it provisions the AnalysisTaskResponseQueue. From this information, the Plugin then establishes a connection to both queues. After that, the Plugin is ready to receive messages from the AnalysisTaskRequestQueue and send messages to the AnalysisTaskResponseQueue. With that, the plugin registration process is finished.

3.5.4 Transformation Process

The transformation process starts with a user supplying a technology-specific deployment model and ending with the transformation framework returning the result in the form of the created technology-agnostic deployment model. This process involves all components of the transformation framework (see Figure 3.7) and follows the phases presented in Figure 3.6. Because this encompasses many steps, we split the modeling of the transformation process into several diagrams. In these diagrams, we use the abbreviations TSDM and TADM for the technology-specific deployment model and the technology-agnostic deployment model, respectively. Generally, the transformation framework can run several transformation processes but not in parallel.

Figure 3.12 shows the first steps of the transformation process, focusing on the AnalysisManager. The User starts the process by calling the AnalysisManager. They provide information about the technology-specific deployment model that they want to transform. The information comprises the deployment technology of the deployment model, commands for executing it, and one or more locations in the form of URLs where the deployment model is located. The AnalysisManager starts with looking for a suitable Plugin for analyzing the technology-specific deployment model. For that, it queries the ConfigurationsDatabase which stores information about available Plugins. In alignment with the conceptualized transformation phases, it first looks for a Plugin following a static

analysis approach. If there is no suitable Plugin, the analysis manager skips this phase and looks for a Plugin that implements a dynamic analysis technique. In case the AnalysisManager is not able to find any Plugin that can transform the given technology-specific deployment model, it ends the transformation process. Then it responds to the User with an error message.

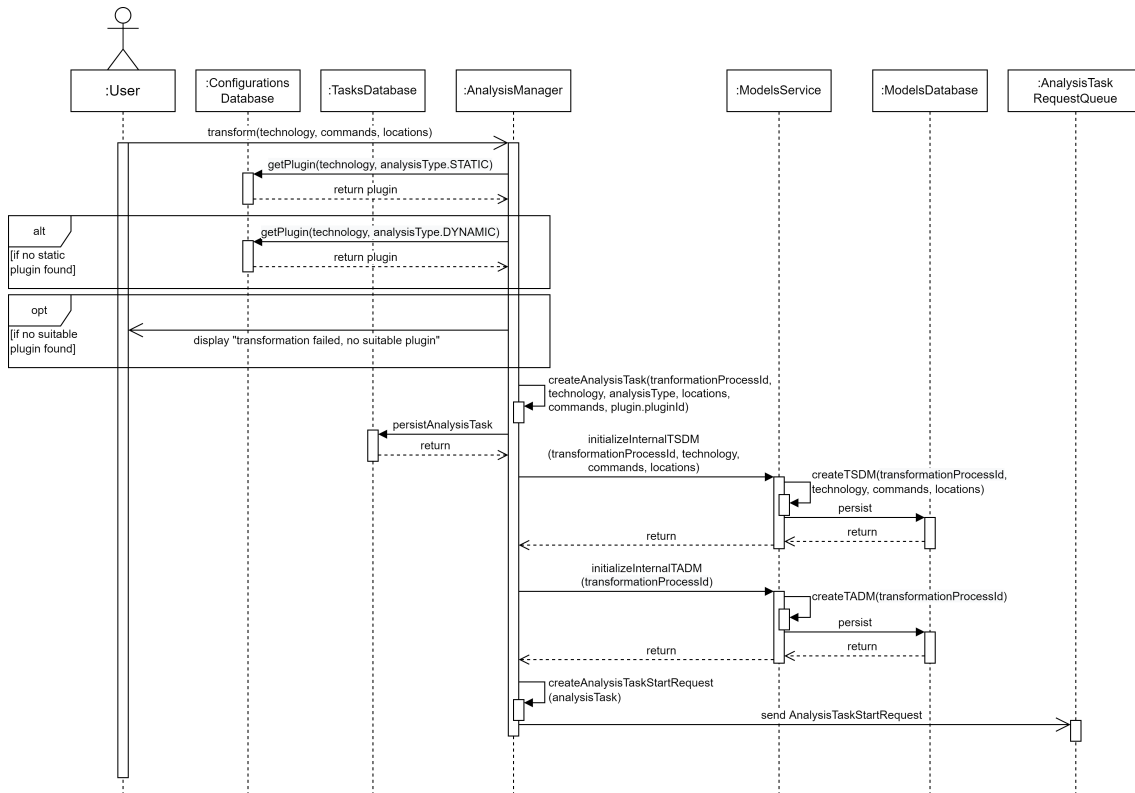


Figure 3.12: UML sequence diagram showing the start of the transformation process. The AnalysisManager creates the first AnalysisTask and sends it to a suitable Plugin.

Otherwise, the AnalysisManager continues with creating the first AnalysisTask. Figure 3.13 shows a UML class diagram of such an AnalysisTask. It contains a unique *taskId* and a *transformationProcessId* that the AnalysisManager generates to correlate the AnalysisTask to the current transformation process. In addition, it includes the information about the technology-specific deployment model given by the User. The AnalysisManager maps the locations to a class *DeploymentModelLocation*, that can optionally provide a *startLineNumber* and an *endLineNumber* to limit the area of a given file. The AnalysisTask has a field *status* which helps to keep track of the current state of processing. For the initial AnalysisTask is sets the status to “RUNNING”. Finally, the AnalysisTask contains the *pluginId* of the Plugin that it selected for the analysis.

The AnalysisManager persists this initial AnalysisTask in the TasksDatabase (see Figure 3.12). Then it sends a request to the ModelsService for initializing the internal technology-specific deployment model and the internal technology-agnostic deployment model. It provides the previously created transformationProcessId and further information about the technology-specific deployment model. The ModelsService creates the internal deployment models and persists them in the ModelsDatabase. Finally, it creates an AnalysisTaskStartRequest message from the AnalysisTask and

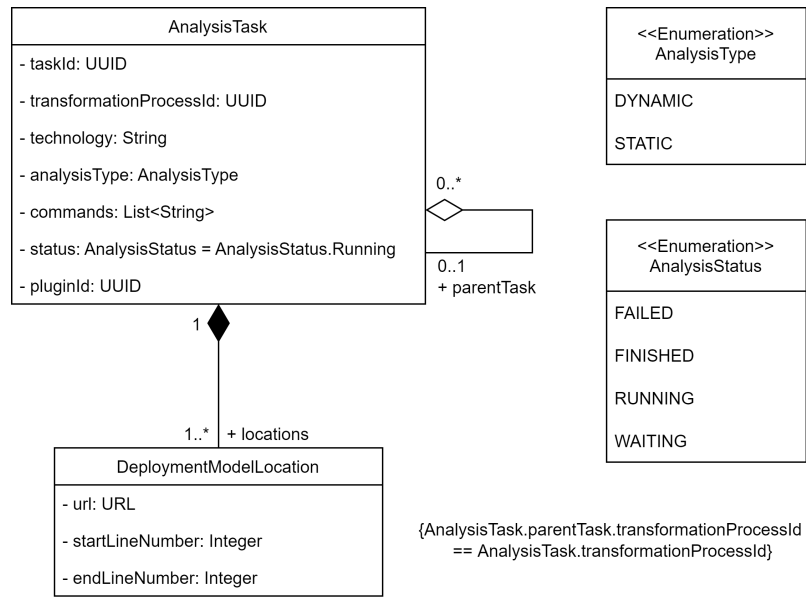


Figure 3.13: UML Class diagram of the analysis task.

sends it to the AnalysisTaskRequestQueue of the selected Plugin. Figure 3.14 shows a model of the AnalysisTaskStartRequest message alongside the other types of messages that are involved in the transformation process at a later stage.

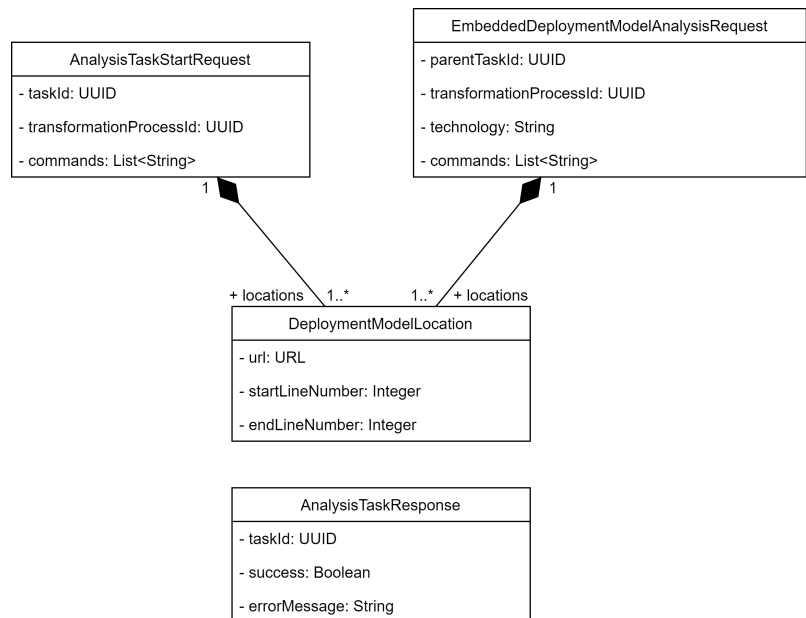


Figure 3.14: UML class diagram of the different messages that the AnalysisManager and the Plugins send and consume from the messaging queues during the transformation process.

The transformation process continues with the selected Plugin consuming the AnalysisTaskStartRequest from its AnalysisTaskRequestQueue. The Plugin then starts with the analysis of the technology-specific deployment model. The processing steps of the Plugin are the second part of the transformation process that we present in Figure 3.15.

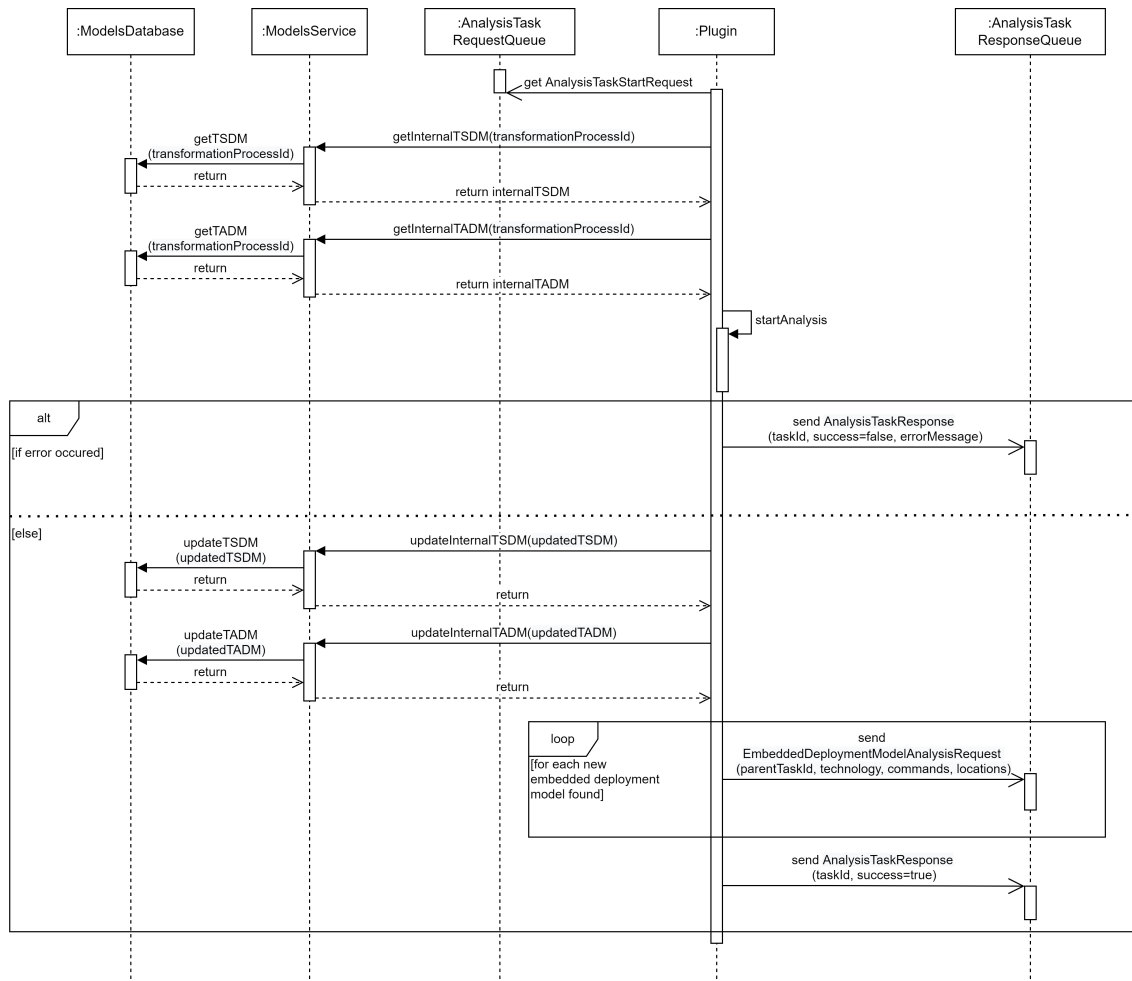


Figure 3.15: UML sequence diagram showing the analysis and possible responses of a Plugin during the transformation process.

First, the Plugin retrieves both internal deployment models from the ModelsService. After that, it runs the analysis of the technology-specific deployment model. The exact actions and workflow of this analysis depend on the specific implementation of the Plugin. In general, it covers the following steps:

1. Parse in and analyze the technology-specific deployment model from the given locations.
2. Update the internal technology-specific deployment model with new information and set the analysis progress and comprehensibility metrics.
3. Update the internal technology-agnostic deployment model with new EDMM entities and set the confidence metric.

If an error occurs during the analysis, the Plugin creates and sends an `AnalysisTaskResponse` message (see Figure 3.14) to the `AnalysisTaskResponseQueue`. To indicate that the analysis has failed, the Plugin sets the `success` field of the `AnalysisTaskResponse` message to “false”. It adds an `errorMessage` that describes the error and its cause. If the analysis was successful, the Plugin sends the modified internal deployment models to the `ModelsService`, which then updates them in the `ModelsDatabase`. If the Plugin discovered embedded deployment models during the analysis, it would create corresponding `EmbeddedDeploymentModelAnalysisRequest` messages (see Figure 3.14) and send them to the `AnalysisTaskResponseQueue`. The `EmbeddedDeploymentModelAnalysisRequest` message contains information about the embedded deployment model, like the deployment technology, possible commands, and the locations. Furthermore, the Plugin writes the `taskId` and the `transformationProcessId` of the `AnalysisTaskStartRequest` that it consumed in the beginning to the `parentTaskId` and the `transformationProcessId` fields, respectively. Finally, the Plugin sends an `AnalysisTaskResponse` message to the `AnalysisTaskResponseQueue`, this time with the `success` field set to “true”.

The `AnalysisManager` listens to the `AnalysisTaskResponseQueue`. When it consumes an `EmbeddedDeploymentModelAnalysisRequest` message, it extracts the information and creates a new `AnalysisTask` for analyzing the embedded deployment model. The `parentTaskId` is used to add the `AnalysisTask`, that the `EmbeddedDeploymentModelAnalysisRequest` resulted from, as the `parentTask` to the newly created `AnalysisTask` (see Figure 3.13). Both `AnalysisTasks` have to share the same `transformationProcessId`. The `AnalysisManager` persists the new `AnalysisTask` in the `TasksDatabase` with the `status` field set to “WAITING”.

When the `AnalysisManager` gets an `AnalysisTaskResponse`, it updates the status of the `AnalysisTask` with the given `taskId` to either “FAILED” or “FINISHED” based on the `success` field (see Figure 3.16). The `AnalysisManager` now has several options on how to proceed with the transformation process:

1. The previous `AnalysisTask` used a Plugin with a static analysis technique and the `AnalysisManager` has to find a Plugin using a dynamic analysis technique.
2. There are other `AnalysisTasks` with a status set to “WAITING” and the `AnalysisManager` has to run on of them.
3. The transformation process is finished and the `AnalysisManager` has to return the result to the User.

To make the right choice, the `AnalysisManager` examines the current conditions. If the `analysisType` of the previous `AnalysisTask` is set to “STATIC”, it queries the `ConfigurationsDatabase` for a Plugin with the same technology but with an `analysisType` set to “DYNAMIC” (see the second part of Figure 3.16). If the `AnalysisManager` finds a Plugin, it creates a new `AnalysisTask` and sends a corresponding `AnalysisTaskStartRequest` to this Plugin. Otherwise, it checks if the current conditions meet the second option.

For that, the `AnalysisManager` queries the `TasksDatabase` for `AnalysisTasks` with status “WAITING”, as shown in Figure 3.17. It prioritizes `AnalysisTasks` that contain the previous `AnalysisTask` as their `parentTask`. If it finds one, it searches for a Plugin that can analyze it, prioritizing the static analysis technique. Then it sends an `AnalysisTaskStartRequest` to the corresponding `AnalysisTaskRequestQueue`. If there is no suitable Plugin, the `AnalysisManager` sets the status of the `AnalysisTask` to “FAILED” and queries the `TasksDatabase` for another waiting `AnalysisTask`.

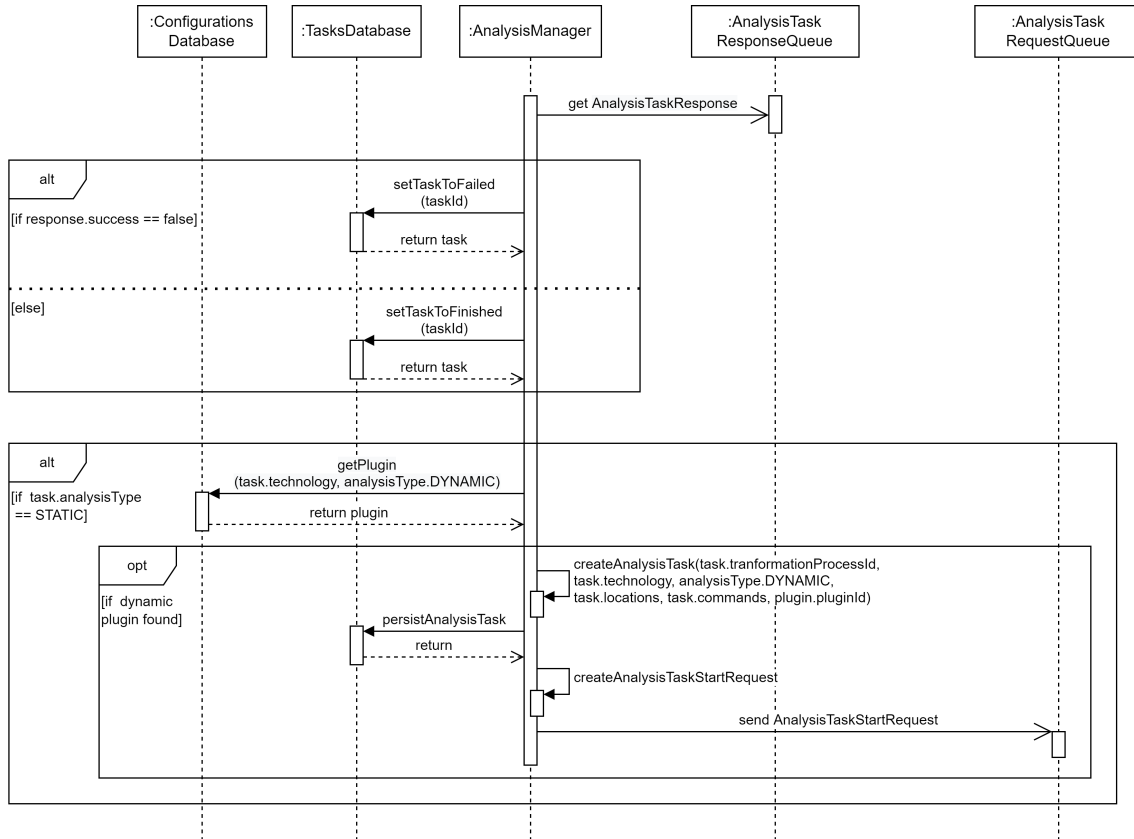


Figure 3.16: UML sequence diagram showing the first steps of the processing of an AnalysisTaskResponse by the AnalysisManager.

When the status of all AnalysisTasks is either “FAILED” or “FINISHED”, the AnalysisManager recognizes that the transformation process is finished. In this case, it calls the ModelsService to get the result of the transformation process. The ModelsService calculates the metrics from the internal deployment models. Then it transforms the internal technology-agnostic deployment model to an EDMM conforming format by removing the confidence metric and the transformationProcessId. It exports the technology-agnostic deployment model as a file to a path on the local file system. After that, the ModelsService sends a response to the AnalysisManager that contains the path together with the calculated metrics. The AnalysisManager forwards this information to the User.

3 Concept and Design

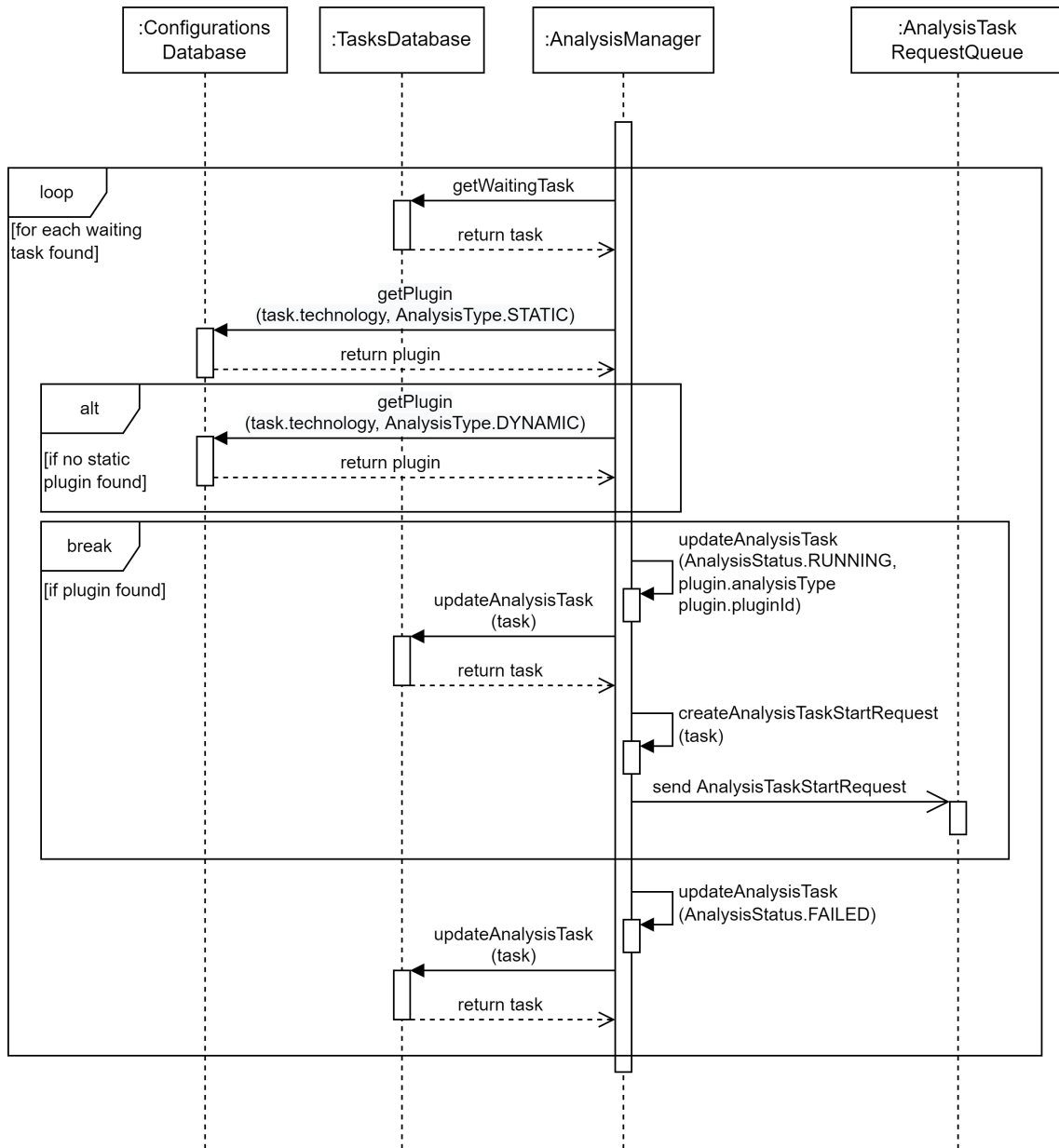


Figure 3.17: UML sequence diagram showing how the AnalysisManager searches for and starts AnalysisTasks that need to be run.

4 Prototypical Realization

In this chapter, we present a prototypical realization of the transformation framework. It implements the concepts and follows the designed architecture. We use it to review and improve the concepts and it serves as an example of a possible implementation. Figure 4.1 presents an overview of the components that we created.

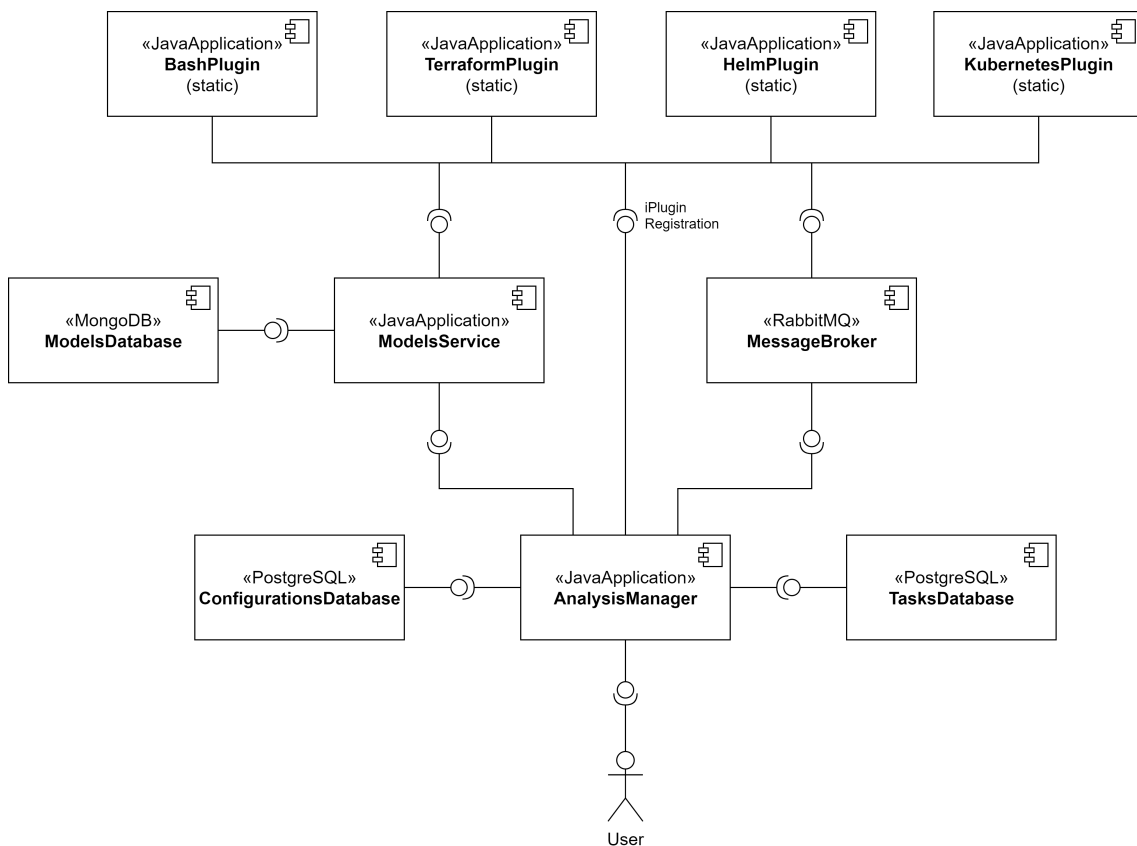


Figure 4.1: UML component diagram showing the architecture of the realized prototype.

The plugins, the models service, and the analysis manager are self-written applications in the Java programming language. The analysis manager provides a CLI for the user interaction. We realized four plugins for the deployment technologies Terraform, Kubernetes, Helm, and Bash. Because the dynamic analysis technique is more complex than the static analysis technique, we focused on the latter to provide support for a broader range of different deployment technologies. For each of the databases, we selected an appropriate database management system. In the case of the models database we decided to use MongoDB, whereas we selected PostgreSQL for realizing the configurations database and the tasks database. The communication between the analysis manager

and the plugins for the transformation process is realized through asynchronous messaging and modeled in the architecture of the transformation framework (see Figure 3.7) by the components of type “Queue”. We realized this by deploying a message broker software called RabbitMQ that handles the message routing between the applications.

The following sections give more details about applied technologies and how we implemented the Java applications (Section 4.1), the databases (Section 4.2), and the message broker (Section 4.3). Furthermore, Section 4.4 describes how we realized the user interaction with the prototypical implementation of the transformation framework through a CLI. The last section provides detailed information about each plugin with focus on their mapping between the entities of their deployment technology to the entities of EDMM (see Section 4.5).

4.1 Java Applications

For the applications in the transformation framework, which are the analysis manager, the models service, and the plugins, we selected the Java-based framework Spring Boot. It allows to quickly and easily create high-quality modern applications [VMw22d]. For this purpose, it offers so-called “starter dependencies” for the integration with various tools like databases or message brokers and supplies them with a default configuration. We selected Maven¹ as the build tool for all applications. It integrates the starter dependencies into the application. Furthermore, Spring Boot directly embeds a servlet container like Tomcat, which hosts the application when deployed. Through the website <https://start.spring.io/> we generated a base project for each application which already included most dependencies. This allowed us to directly focus on the development of the application code.

We version the application code in GitHub repositories under a private organization called “UST-TAD”². It also hosts a repository called “deployment-config”, which contains a Docker-Compose deployment model for deploying third-party services that are part of the transformation framework and integrate with the self-written applications. The deployment model includes the database systems and a message broker, which we describe in more detail in the following sections.

4.2 Databases

The transformation framework contains three different databases (see Figure 3.7). The analysis manager uses the configurations database and the tasks database to store objects of type Plugin and AnalysisTask, respectively. The models service uses the models database to store the internal deployment models.

For the tasks database and the configurations database, we use PostgreSQL [The22d]. It is a relational and open source database system based on Structured Query Language (SQL). We selected it because both the Plugin (see Figure 3.11) and AnalysisTask (see Figure 3.13) objects are not very complex

¹Maven: <https://maven.apache.org/>

²GitHub repositories for the application code: <https://github.com/UST-TAD>

and do not require any special features. PostgreSQL is a popular and widely used solution that we consider sufficiently reliable and performant for both databases. For the deployment of the databases, we use the official Docker image³.

In case of the models database, we decided to use the document database MongoDB. In MongoDB, objects are stored in documents in the Binary JSON (BSON) format [Mon21]. This is a format close to JSON and also YAML. We chose the latter as the format of the technology-agnostic deployment model that the transformation framework outputs to the user (see Section 4.4). Therefore, we see MongoDB as a potentially good fit for the models database because it stores the technology-agnostic deployment model in a format similar to the output format. Each internal technology-agnostic deployment model and internal technology-specific deployment model is one document that contains all nested objects. The only exception are the embedded deployment models in the internal technology-specific deployment model, which are self-references. Each embedded deployment model is a separate document and the embedding deployment model references it through the `id` field. MongoDB organizes documents in collections. We store all documents of the internal technology-specific deployment model in one collection. The internal technology-agnostic deployment model uses another collection. We deploy the configurations database using the official Docker image⁴.

4.3 Message Broker

For realizing the queues and managing the message routing, we selected the RabbitMQ message broker. It is a widely used and open source solution for messaging that supports multiple programming languages for client applications and messaging protocols [VMw22b]. The messaging protocol we used is the Advanced Message Queuing Protocol (AMQP). It enables applications to communicate through messaging based on an entity model presented in Figure 4.2.

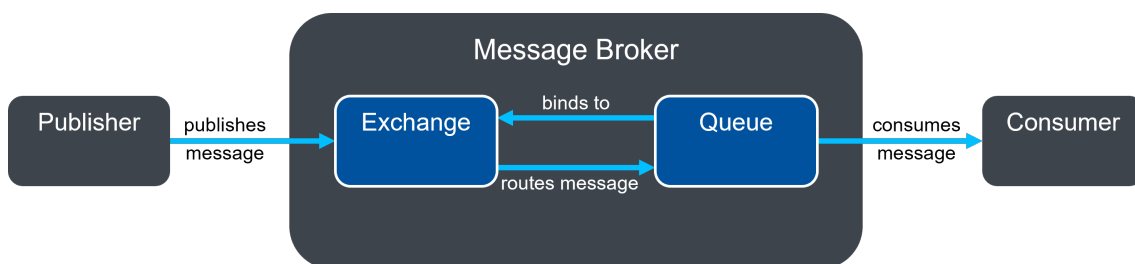


Figure 4.2: Overview of the entities in the AMQP model.

Publishers and *consumers* are applications that connect to the *message broker*, while *exchanges* and *queues* are entities, that can be created and configured in the *message broker* [VMw22a]. Publishers create messages and publish them to exchanges. Exchanges route copies of the message to queues that have a *binding* on the exchange. The message broker then sends this message to consumers that subscribe to this queue. Alternatively, consumers can consume messages from queues on demand by actively pulling them.

³PostgreSQL Docker image on DockerHub: https://hub.docker.com/_/postgres

⁴MongoDB Docker image on DockerHub: https://hub.docker.com/_/mongo

The routing of messages depends on the exchange type and the specification of the binding. The AMQP provides four different types of exchanges:

Fanout Exchanges simply route messages to all queues that have a binding on them.

Direct Exchanges route messages based on routing keys. Bindings of queues to direct exchanges contain a specific routing key. Messages sent to direct exchanges also contain a routing key. Direct exchanges compare the routing keys of messages and bindings and send messages to the corresponding queues of matching bindings.

Topic Exchanges work similar to direct exchanges but allow routing of messages based on more complex routing keys. Instead of specifying just one value, the routing keys of topic exchanges consist of several values, each belonging to a specific topic. Queues bind to topic exchanges by specifying a value for each topic and only receive messages matching this combination.

Headers Exchanges route messages based on attributes in the message header. The queue bindings must contain matching values for these attributes. Headers exchanges allow combining several attributes while specifying them as independent parameters.

Based on this, we implemented the creation of AMQP entities for the realization of the `AnalysisTaskRequestQueue` and the `AnalysisTaskResponseQueue` components presented in Figure 3.7. We deploy the RabbitMQ message broker using the official and publicly available Docker image⁵ and Docker Compose. It only contains some default exchanges provided by RabbitMQ. When the analysis manager starts, it connects to the RabbitMQ message broker and creates the required AMQP entities. The analysis manager and the plugins use the Spring AMQP project. It provides functionality to connect to a running message broker that uses the AMQP protocol, create AMQP entities and send and receive messages [VMw22c]. For realizing the `AnalysisTaskResponseQueue`, the analysis manager creates a fanout exchange and one queue that it binds to the exchange. It then subscribes to the queue by specifying a so-called “listener” on a function. Whenever a new message arrives in the queue, the message broker invokes the listener function and supplies the message. The realization of the `AnalysisTaskRequestQueues` is more complex because the analysis manager has to create the queues dynamically when a new plugin registers. At startup, the analysis manager only creates a headers exchange. When a plugin registers, it creates a new queue and binds it to the headers exchange based on two attributes. These are the name of the deployment technology and the supported analysis technique of the plugin. The analysis manager then responds to the plugin with the names of the newly created queue and the fanout exchange of the `AnalysisTaskResponseQueue`. Subsequently, the plugin registers a new listener for subscribing to its queue. It also registers the fanout exchange so that it can later publish messages to it. When the analysis manager sends an `AnalysisTaskStartRequest` message, it adds the deployment technology and the analysis technique as parameters to the message header. Consequently, the headers exchange can route it to the correct queue. The plugins respond to the analysis manager by sending messages to the fanout exchange. The fanout exchange always routes the messages to the queue of the `AnalysisTaskResponseQueue` because it is the only queue with a binding to it. Additionally, the plugins add a “format indicator” to the header of the messages so that the analysis manager can distinguish `AnalysisTaskResponse` messages from `EmbeddedDeploymentModelAnalysisRequest` messages when it consumes them from the `AnalysisTaskResponseQueue`.

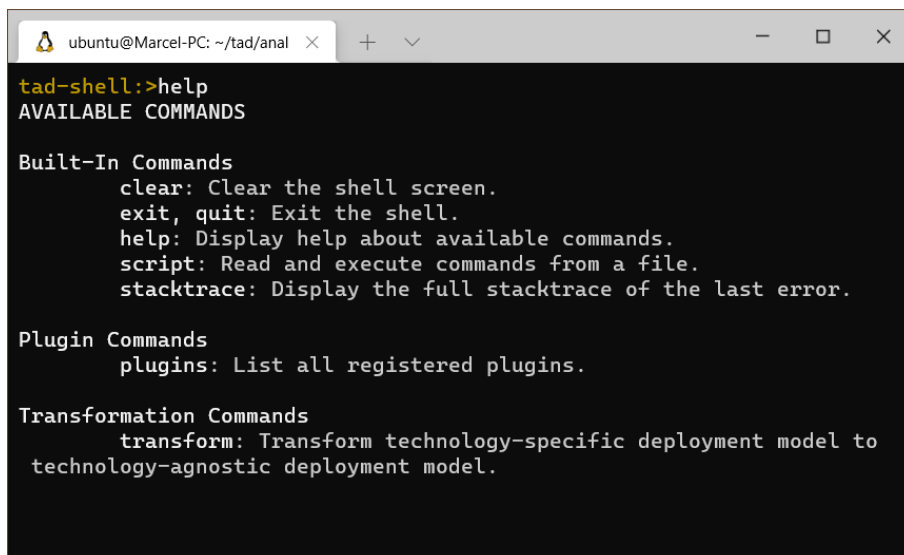
⁵RabbitMQ Docker image on DockerHub: https://hub.docker.com/_/rabbitmq

4.4 User Interaction

The user interaction with the transformation framework mainly comprises two different actions:

1. The user passes information about a technology-specific deployment model to the transformation framework to start the transformation process.
2. At the end of the transformation process, the transformation framework presents the result to the user.

As this is not very complex, we wanted to keep things simple and decided to integrate a custom CLI into the analysis manager application. For that, we used the project Spring Shell, which integrates seamlessly into Spring Boot projects and provides the baseline for a fully functional CLI [VMw22e]. It offers an annotation-driven programming model for implementing custom commands. When starting the analysis manager, it launches a CLI where a user can input commands to interact with the transformation framework.

A screenshot of a terminal window titled 'ubuntu@Marcel-PC: ~/tad/anal'. The prompt is 'tad-shell:>'. The user has entered 'help', and the output is as follows:

```
tad-shell:>help
AVAILABLE COMMANDS

Built-In Commands
  clear: Clear the shell screen.
  exit, quit: Exit the shell.
  help: Display help about available commands.
  script: Read and execute commands from a file.
  stacktrace: Display the full stacktrace of the last error.

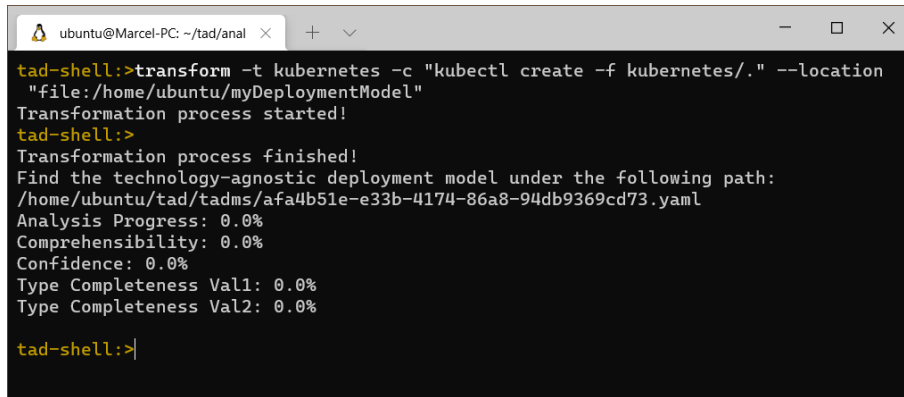
Plugin Commands
  plugins: List all registered plugins.

Transformation Commands
  transform: Transform technology-specific deployment model to
             technology-agnostic deployment model.
```

Figure 4.3: The tad-shell CLI with all available commands.

Figure 4.3 shows this CLI, which we call “tad-shell” and an overview about available commands, displayed by executing the “help” command. The help command is one of six built-in commands that the Spring Shell project provides by default. In addition, we added two custom commands. The “plugins” command returns a list of registered plugins. With this, a user can assess if the transformation framework contains the required plugins to transform a technology-specific deployment model before starting the transformation process.

The user starts the transformation process by executing the “transform” command. Figure 4.4 shows an exemplary execution, where the user provided the deployment technology “kubernetes”, a command to execute the deployment model, and a URL to its location on the local file system as parameter values. In this case, the transformation framework found a suitable plugin and responded with “Transformation process started!”. After a while, the transformation process finished successfully, and the transformation framework printed the result to the CLI. The result contains a path to the technology-agnostic deployment model and the calculated metrics.



```

ubuntu@Marcel-PC: ~/tad/anal
tad-shell:>transform -t kubernetes -c "kubectrl create -f kubernetes/." --location
"file:/home/ubuntu/myDeploymentModel"
Transformation process started!
tad-shell:>
Transformation process finished!
Find the technology-agnostic deployment model under the following path:
/home/ubuntu/tad/tadms/afa4b51e-e33b-4174-86a8-94db9369cd73.yaml
Analysis Progress: 0.0%
Comprehensibility: 0.0%
Confidence: 0.0%
Type Completeness Val1: 0.0%
Type Completeness Val2: 0.0%

tad-shell:>|

```

Figure 4.4: Execution of the transform command in the tad-shell CLI. The transformation framework prints the result of a successful transformation process to the tad-shell.

```

1 ---
2 properties: [...]
3 components: [...]
4 relations:
5   - <Name of Relation>:
6     type: "name of relation type"
7     description: "description"
8     source: "name of source component"
9     target: "name of target component"
10    operations: [...]
11 component_types: [...]
12 relation_types: [...]

```

Listing 4.1: Specification of relations in the modified EDMM in YAML specification.

Because technology-agnostic deployment models can potentially become large, printing them to the CLI is not a good solution. Therefore, the transformation framework should output the technology-agnostic deployment model to a file on the local file system. Additionally, it facilitates further processing or evaluation steps by the user. We decided to use the EDMM in YAML specification (see Section 3.3) as the format for the outputted technology-agnostic deployment model because it is concise and human-readable. However, we made one change to the EDMM in YAML specification. The official specification stores each relation under its source component, as shown in Listing 3.1. We changed that to store the relations in a list directly under the deployment model next to the components, component types, and relation types, as shown in Listing 4.1. With this, we are in line with the EDMM and can express additional information, like the operations that a relation contains.

For the transformation from the internal format to the YAML format, we used the library Jackson⁶ and its YAML module to specify serialization classes for each EDMM entity.

⁶Jackson Project Home: <https://github.com/FasterXML/jackson>

4.5 Plugins

The plugins of the transformation framework implement and execute the actual transformation logic. For the prototypical realization, we developed plugins for the deployment technologies Terraform, Kubernetes, Helm, and Bash.

The selected deployment technologies cover a variety of use cases. Therefore, each plugin shows a different approach to analyzing technology-specific deployment models. Terraform is an IaC tool that focuses on infrastructure resources. In contrast, Kubernetes is based on containerized applications and only supports the deployment of software components. Helm is a deployment technology based on Kubernetes, which provides reusable and pre-configured Kubernetes deployment models. Bash is the only selected deployment technology following an imperative deployment modeling approach. We do not provide a plugin for Docker or other container technologies. Therefore, the plugins add Docker images that they find as EDMM artifacts to EDMM components.

The following sections describe the implementation of the plugins and their specific transformation logic. For that, we first present the plugin template in Section 4.5.1. After that follow sections describing the Terraform (Section 4.5.2), Kubernetes (Section 4.5.3), Helm (Section 4.5.4) and Bash (Section 4.5.5) plugins. For each of those sections, we first provide a brief introduction to the deployment technology with focus on the specifics of the deployment models. Then we describe the mapping from the deployment technology to the EDMM entities that we want to create. Finally, we describe the prototypical implementation of the plugin, based on the defined mappings.

4.5.1 Plugin Template

Some features are shared among the plugins so that all plugins implement the same functionality to a certain degree. This includes the initial plugin registration process, accessing the endpoint of the models service to get and update the internal deployment models, and the requests and responses they receive and send via the message broker. Therefore, we created a template project using Java Spring Boot that implements this functionality⁷. We can now duplicate this project and just need to make minor modifications to create a new plugin. For example, we created environmental variables for the supported deployment technology and the analysis type, and need to adjust their values. The plugin uses these variables to correctly set the parameters of the binding to the request queue. The “README.md” file lists all of these necessary modification steps. As a result, we do not need to bother with implementing the same functionality repeatedly and can directly focus on implementing the actual transformation logic.

4.5.2 Terraform Plugin

Terraform is an IaC tool with a primary focus on deploying infrastructure resources like compute, storage, and networking, but also offers capabilities to deploy software components [Has22]. A Deployment model in Terraform is called *Terraform configuration*. It contains files with the file name extension “.tf”, and it is possible to distribute it among several files and directories. A Terraform

⁷GitHub repository of plugin template project: <https://github.com/UST-TAD/plugin-template-springboot>

```
1 <BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
2     # Block body
3     <IDENTIFIER> = <EXPRESSION> # Argument
4 }
```

Listing 4.2: Syntax of the Terraform language [Has21].

configuration mainly consists out of *resource*, *variable* and *provider* definitions, that are specified in the *Terraform language* [Has21], a custom data format similar to JSON. Each resource in the Terraform configuration describes one component that should be deployed, detailed by *arguments*. Resources have a specific *resource type*, which provides a set of arguments that need to be specified. We can assign values to the arguments in the definition of the resource or through the specification of variables. Every resource of the Terraform configuration can reuse these variables. Providers add resource types and optionally *data sources*, which enable Terraform to access the Application Programming Interfaces (APIs) of certain services or platforms [Has22]. By default, Terraform does not provide any resource types. Therefore, a Terraform configuration must contain at least one provider definition. For instance, the “Azure Provider” allows specifying resources of resource types that represent offerings of the Azure cloud platform. Additionally, it provides access to the “Azure Resource Manager API”, a data source that Terraform uses to deploy the described resources.

Similar to Wurster et al. [WBF+20], we map resources and resource types to components and component types in EDMM, respectively. Additionally, we map arguments to properties and resolve variables accordingly. The Terraform plugin parses Terraform configurations into an internal representation. It then transforms this internal representation into entities of the internal technology-agnostic deployment model. For the first step, the plugin needs to analyze the files of the Terraform configuration, which uses the Terraform language. The expression in Listing 4.2 describes the syntax of the Terraform language.

It mainly consists out of *blocks* and *arguments*. Blocks act as containers for information and have a *block type*. The block type indicates, if the block is a resource, provider, or variable definition. Furthermore, the block type defines the number of *block labels* that need to be specified. For instance, a resource requires two block labels. The first block label defines the resource type, while the second one specifies its name. A block contains an arbitrary number of arguments and nested blocks in its block body. An argument consists out of an *identifier* and an *expression*.

The Terraform plugin parses these expressions into the internal Terraform model shown in Figure 4.5. While doing so, it also updates the internal technology-specific deployment model. The plugin only extracts information needed for the transformation to the technology-agnostic deployment model. Primarily, these are the resource definitions that it parses into the *Resource* entities. Analogous to the syntax of the Terraform language, they are identified by their *resourceType* and *resourceName* and contain *Blocks* and *Arguments*. As there are currently over 1700 providers available [Has22], the Terraform plugin supports only a subset of them. Therefore, the plugin also parses in provider definitions into *Provider* entities to be able to determine, if it supports them. If the Terraform plugin comes upon a provider or resource type during the analysis that it does not support, it marks this in the internal technology-specific deployment model by setting the comprehension of the respective lines to zero. Additionally, the plugin stores variable definitions in *Variable* entities.

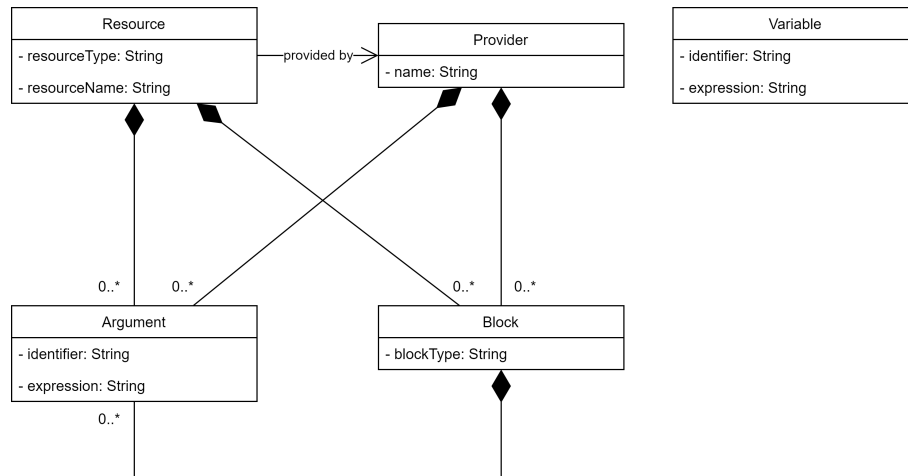


Figure 4.5: UML Class diagram of the internal Terraform model.

```

1 physical_node:
2   extends: "-"
3   description: "A physical node providing compute and storage resources
4   "
5   properties:
6     - cpu_count:
7       type: "INTEGER"
8       required: false
9     - ram_GiB:
10      type: "INTEGER"
11      required: false
12     - storage_GiB:
13      type: "INTEGER"
14      required: false
15   operations: []
  
```

Listing 4.3: EDMM component type for a physical node in the YAML syntax.

In the second step, the Terraform plugin analyzes the Resource entities and creates corresponding entities of the internal technology-agnostic deployment model. If an Argument uses a variable in an expression, the plugin can resolve it by looking at the Variable entities. We present an example for an EDMM component type that the Terraform plugin can create in Listing 4.3.

4.5.3 Kubernetes Plugin

Kubernetes facilitates the deployment of containerized applications [The22c]. For that, it offers the *Kubernetes cluster*, which is a platform consisting of several components that can host containerized applications. It provides various features that ease the deployment and operation of containerized applications, like desired state management or load balancing of network traffic. Kubernetes describes application deployments in declarative *configuration files*. They contain definitions of

4 Prototypical Realization

```
1 ---
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: my-deployment
6 spec:
7   selector:
8     matchLabels:
9       app: my-deployment
10  replicas: 2
11  template:
12    spec:
13      containers:
14        - name: my-deployment
15          image: myImage
16          env:
17            - name: env_variable
18              value: "value"
19          ports:
20            - name: "my-containerport"
21              containerPort: 8080
22 ---
23 apiVersion: v1
24 kind: Service
25 metadata:
26   name: my-service
27 spec:
28   ports:
29     - name: "my-port"
30       port: 80
31       targetPort: 8080
32   selector:
33     app: my-deployment
```

Listing 4.4: Definition of exemplary Kubernetes deployment and service objects in YAML format.

Kubernetes objects either in the YAML or JSON format [The22b]. Kubernetes provides different types of Kubernetes objects for describing, among others, which containerized applications should run on the cluster, how they can be accessed, and which resources are available. The most important Kubernetes object types are the *deployment* and the *service*. Deployments describe a containerized application, while services reference a deployment and describe how it can be accessed. Listing 4.4 shows an example of a deployment and service definition in one YAML file.

The upper part shows the definition of the deployment object with the name “my-deployment”. With the *replicas* field, it specifies that Kubernetes should deploy two instances of this application in the Kubernetes cluster. A deployment can describe one or more containers that should run together. The definition of a container includes its name, the Docker image, optional environmental variables, and the port under which it should be accessible. The lower part of Listing 4.4 shows the definition of a Kubernetes service called “my-service”. It specifies a *selector* that Kubernetes uses

to find a deployment object with a matching selector field. Furthermore, the service defines a port named “my-port”, that exposes the application on port “80”. The *targetPort* field specifies to which container port on the matched deployment object it should map.

The Kubernetes plugin transforms Kubernetes configuration files in three phases:

1. Parse in the Kubernetes configuration files and update technology-specific deployment model
2. Create EDMM components and corresponding component types
3. Find and create EDMM relations

In the first phase, the Kubernetes plugin parses in the Kubernetes configuration files. It extracts the relevant information and transforms it into an intermediary data model, similar to the Terraform plugin. For each Kubernetes object type, it provides a different meta-model. Figure 4.6 shows the meta-model for the deployment object. It retains the *name* of the deployment and the number of *replicas*. Furthermore, it contains one or more *labels* and *container*. Each container has a *name*, an *image*, *container ports* and *environment variables* that refer to the respective fields in the original Kubernetes deployment object.

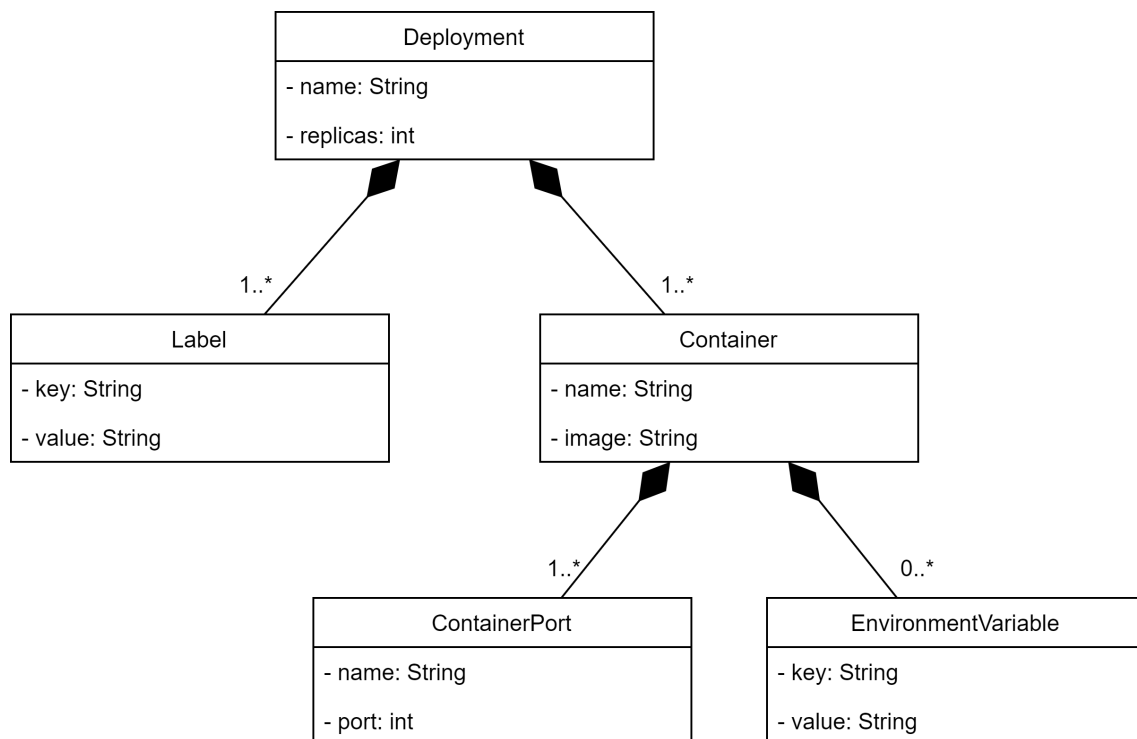


Figure 4.6: UML Class diagram of the meta-model for the internal representation of Kubernetes deployment objects.

We present the meta-model for Kubernetes service objects in Figure 4.7. Here, the plugin extracts the *name* of the service and the *selectors* for matching Kubernetes deployment objects. Additionally, it stores information about the specified ports in the *ServicePort* entity.

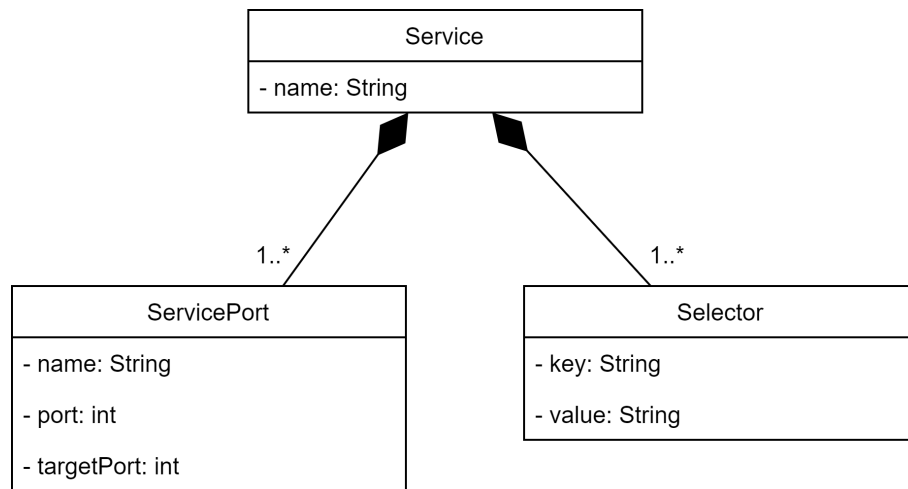


Figure 4.7: UML Class diagram of the meta-model for the internal representation of Kubernetes service objects.

With that, the Kubernetes plugin currently supports the analysis and transformation of the Kubernetes object types deployment, service, and *stateful set*. Kubernetes stateful sets are similar to Kubernetes deployment objects and also describe the deployment of a containerized application. The main difference is that Kubernetes provides persistent storage for containerized applications that stateful sets describe. As a result, their definitions in the configuration files share the same base properties. Therefore, the Kubernetes plugin can transform stateful sets with the presented method by treating them in the same way as Kubernetes deployment objects. The plugin marks definitions of other Kubernetes object types that it does not support in the technology-specific deployment model as not comprehended.

After the Kubernetes plugin has analyzed all configuration files, it proceeds with the second phase. Here, it iterates over the internal representations of the deployments that it found previously. For each container of a deployment, it creates a new EDMM component. From the image field, it creates an EDMM artifact and adds it to the component. It converts the replicas field and the environment variables into EDMM properties. Furthermore, the plugin searches for services that expose the deployment. It checks, if a service selector matches one of the labels of the deployment. For each match, it creates properties from the service port and the container port. Finally, it creates a new EDMM component type for the newly created component. This component type contains copies of all properties, but with an empty value field.

In the third phase, the Kubernetes plugin tries to find relations between the newly created components. It can identify two types of relations: “hosted on” and “connects to”. This is based on the definition of relationship directions shown in Figure 2.1. The corresponding EDMM relation types are already present in the technology-agnostic deployment model because the models service adds them when initializing a new technology-agnostic deployment model. First off, the Kubernetes plugin searches for a component in the technology-agnostic deployment model that can host the newly created components. Since Kubernetes relies on containerized applications, this must be a component of type “container runtime”. A container runtime is a software responsible for running containerized applications. If it finds a suitable component, the plugin creates appropriate relations of type “hosted on”. Secondly, the Kubernetes plugin looks into the properties of the newly created components

to find “connects to” relations to other components, for example, specified by a URL. Within the Kubernetes cluster, the containerized applications can connect via the names of the deployments or services, since Kubernetes provides a Domain Name System (DNS) for that purpose. Therefore, the Kubernetes plugin can extract the host part of a given URL and can scan the internal representation of Kubernetes deployments and services for an object with the same name. If it finds one, it creates a “connects to” relation between the components. Finally, the plugin updates the technology-agnostic deployment model with all newly found EDMM entities.

4.5.4 Helm Plugin

Helm is a package manager for Kubernetes [Hel22]. Helm deployment models are called Helm *charts*. They bundle Kubernetes configuration files and can be created and shared as reusable deployment models. This is similar to the notion of “DevOps Artifacts” defined by Wettinger et al. [WBKL16]. Helm charts can be installed on a Kubernetes cluster to deploy the application that the contained Kubernetes configuration files describe. The Kubernetes configuration files in a Helm chart are called *templates* because Helm allows adding variables in the place of actual values. By providing values for these variables during the installation of a Helm chart, Helm creates the effective Kubernetes configuration files from the templates.

Consequently, each Helm chart embeds a Kubernetes deployment model. The Helm Plugin extracts the Kubernetes configuration files and creates a new embedded deployment model analysis request for the Kubernetes plugin. For that, it needs to extract the effective Kubernetes configuration files with filled-in variables. It does that by executing the “helm template” command with the commands provided in the analysis task start request, which returns a YAML file with all Kubernetes objects. The Helm plugin then saves this YAML file on the local file system. It adds the location of the file to a newly created embedded deployment model of the internal technology-specific deployment model. The Kubernetes plugin can later retrieve the file from this location and analyze it.

4.5.5 Bash Plugin

The primary focus of Bash is not to deploy software applications. Bash is a kind of shell, that enables the creation of so-called “Bash-scripts”. A shell comprises a programming language and a command interpreter [RF20]. Bash-scripts contain Bash commands that a *Bash interpreter* can execute. Consequently, Bash can be used for many purposes. This includes creating Bash-scripts that contain commands for deploying applications. Therefore, Bash-scripts are potential imperative deployment models created with the deployment technology Bash. Furthermore, many deployment technologies provide their own commands that can be interpreted and executed by Bash. For example, the deployment technology Kubernetes provides a command line tool called “kubectl”, for communicating with Kubernetes clusters [The22a]. This allows deploying Kubernetes deployment models from the command line or from a Bash-Script.

With the Bash plugin, we focused on detecting embedded deployment models for the technologies Terraform, Helm, and Kubernetes. The Bash plugin can recognize commands of the command line tools provided by each deployment technology. In the case of Kubernetes, the plugin scans for “kubectl create” and “kubectl apply” commands which deploy Kubernetes configuration files [The22a]. If it finds one, it creates and adds a new embedded deployment model to the internal

technology-specific deployment model with the command added to the list of commands. Both `kubectl` commands allow to optionally specify a path to a file or directory which contains Kubernetes deployment models. The Bash plugin can detect this and transform the path to a `DeploymentModelLocation` entity and add it to the locations of the embedded deployment model. The analysis of Terraform commands works analogously to the one for Kubernetes. In a given Bash-script, the plugin looks for “`terraform init`”, and “`terraform apply`” commands and can extract locations of embedded deployment models from the commands. If several commands refer to the same location, the plugin adds them to the same embedded deployment model. The detection of commands for installing Helm charts works differently. Here, the Bash plugin scans for commands starting with “`helm install`” and adds them to the commands list of a newly created embedded deployment model. Additionally, the plugin can detect commands for adding chart repositories. It adds such a command to every embedded deployment model using Helm that it created after the detection of the command. Consequently, the Helm plugin can access charts from external repositories.

5 Evaluation

This chapter presents an evaluation of the transformation framework with respect to its ability to transform technology-specific deployment models into technology-agnostic deployment models.

In Section 5.1 we describe the design of the evaluation with the help of a Goal Question Metric (GQM) model. This is followed by Section 5.2, where we present an exemplary technology-specific deployment model that we can transform with the transformation framework to generate data for the evaluation. Then we present the results of the evaluation (Section 5.3) and interpret them (Section 5.4). Finally, we discuss possible threats to the validity of the result (Section 5.5).

5.1 Design

For the design of the evaluation we created a model based on the GQM paradigm [BCR94]. We present this model in Table 5.1. The overall goal is to evaluate the ability to transform technology-specific deployment models to technology-agnostic deployment models of the transformation framework. We define two questions and corresponding metrics to answer them. The measurement of these metrics requires performing a transformation of an exemplary technology-specific deployment model with the transformation framework. We present such an exemplary technology-specific deployment model in Section 5.2.

For Q1, we want to assess if the transformation framework correctly implements the plugin registration process and the transformation process, which we presented in Section 3.5. For that, we read out logs of the Java applications (M1) and inspect the registered plugins in the configurations database (M2) and the created AMQP entities on the message broker (M3). In the case of the Java applications, we add statements at specific locations in the code that print messages to the console output. These locations are the endpoints of the application so that it outputs a message every time it receives or sends data. It includes both the REST and messaging endpoints. In the case of the analysis manager, we also output logs when the user executes a transform command. Additionally, it outputs an error message when it cannot find a suitable plugin for transforming a selected analysis task. As a result, we should see in which order the applications are invoked and if this conforms to the conceptualized processes. With the “plugins” command offered by tad-shell in the analysis manager, we can retrieve the list of registered plugins. To measure M2, we deploy all plugins and execute this command. The returned list of registered plugins indicates, if the individual plugin registration processes were successful and the analysis manager can send analysis tasks to them. In addition, we review if the transformation framework correctly creates the AMQP entities on the RabbitMQ message broker, as described in Section 4.3. The analysis manager is responsible for creating these entities during the plugin registration process. The correct creation of these entities is crucial for enabling the communication of the plugins with the analysis manager during

| | | |
|----------|---|--|
| Goal | Purpose Issue Object (process) Viewpoint | Evaluate the ability to transform technology-specific deployment models to technology-agnostic deployment models of the transformation framework from the viewpoint of the developer. |
| Question | Q1 | Does the transformation framework execute the conceptualized plugin registration and transformation processes correctly? |
| Metrics | M1 M2 M3 | Assessment of logs of Java applications. Inspection of registered plugins in the configurations database. Inspection of created AMQP entities on the RabbitMQ message broker. |
| Question | Q2 | Does a resulting technology-agnostic deployment model contain all relevant information of the originating technology-specific deployment model? |
| Metrics | M4 M5 M6 | Differences between the internal technology-specific deployment model at the end of the transformation process and the actual structure of the technology-specific deployment model. Differences between a technology-agnostic deployment model from the transformation result and the expected technology- agnostic deployment model. Assessment of the metrics in the transformation result (analysis progress, comprehensibility, confidence and type completeness). |

Table 5.1: GQM model for the evaluation of the transformation framework.

the transformation process. RabbitMQ allows exporting definitions of the entities that have been created, including the definitions of exchanges, queues, and bindings. Based on this, we should be able to assess if the transformation framework correctly created the AMQP entities.

Q2 focuses on the resulting technology-agnostic deployment model. Here we want to evaluate if it contains all the information about the application deployment that the technology-specific deployment model describes. In the first step, we need to assess if the transformation framework can detect all parts of the technology-specific deployment model. The metric M4 represents this. It measures if the internal technology-specific deployment model contains correct and complete information about the structure of the technology-specific deployment model that the transformation framework transformed. For that, we run a transformation process with the exemplary technology-specific deployment model and export the internal technology-specific deployment model from the models database in the JSON format. We can then compare this to the structure of the exemplary technology-specific deployment model presented in Figure 5.2. This shows if the transformation framework could identify all embedded deployment models and associated files. The metric M5 measures the differences between the actual technology-agnostic deployment model from the transformation result and the expected outcome. For that, we first create an EDMM model in the YAML format for the exemplary technology-specific deployment model. This model contains the complete information about the application deployment that we expect the transformation framework to find. We present this expected technology-agnostic deployment model in its entirety in Listing A.1.

After running the transformation with the exemplary technology-specific deployment model, we compare the actual technology-agnostic deployment model from the transformation result with the expected one. For that, we use Visual Studio Code, which integrates a tool for comparing files¹. We ignore minor differences in naming, ordering of list elements, and indentation, as long as it conforms to the YAML format. The number of differences shows, how well the transformation worked. Additionally, we can have a closer look at these differences to detect areas where the transformation framework struggled or could not find the correct information. Finally, we interpret the metrics in the transformation result to assess the quality of the created technology-agnostic deployment model (M6). The detailed definition and calculation of the metrics is given in Section 3.5.

5.2 Exemplary Technology-Specific Deployment Model

To test and evaluate the transformation framework, we need an exemplary technology-specific deployment model. It is crucial that the application described by the deployment model is of adequate complexity, meaning it consists of several interconnected services with different purposes. This ensures completeness of the transformation methods and significant results from the evaluation.

For that purpose, we use a reference architecture called “T2 Project” [SSB22] and create a technology-specific deployment model for it. Figure 5.1 shows an overview of the different components of the T2 Project. It follows a microservice architectural style with eight services and four databases. The communication between the services is realized through the Saga pattern, which uses messaging to ensure ACID properties for transactions spanning multiple services and databases [Ric21]. An implementation of the T2 Project reference architecture can be found on GitHub². The services are all based on Java, while different database technologies are used. The project also provides capabilities for running JMeter load tests for simulating network traffic.

Based on this, we created a technology-specific deployment model for the T2 Project. The GitHub page of the T2 Project already provides a repository with Kubernetes and Docker-Compose deployment models³. We decided to use the Kubernetes deployment model as a baseline for the exemplary technology-specific deployment model. The Kubernetes deployment model deploys the eight services and a PostgreSQL database on an existing Kubernetes cluster. Furthermore, the documentation gives instructions on installing Helm charts for deploying a MongoDB database and a Kafka message broker to complete the deployment of the T2 Project [SSB22]. The deployed instance of the PostgreSQL database hosts both the saga instance repository and the product repository (see Figure 5.1). Likewise, the MongoDB instance deployed with Helm is responsible for the cart and order repositories. As this does not conform with the presented architecture, we made some changes to the Kubernetes deployment model. First, we added Kubernetes configuration files for a second PostgreSQL database and changed the configuration of the services accordingly so that they connect to the correct database. Secondly, we created a Bash-script that installs two MongoDB Helm charts (one for the cart repository and one for the order repository) and the Kafka message broker. Consequently, we had to adjust the configuration of the services accordingly as well.

¹How to do a Diff in VS Code: <https://vscode.one/diff-vscode/>

²GitHub repositories of the T2 Project: <https://github.com/t2-project>

³GitHub repository with Kubernetes and Docker-Compose deployment models of the T2 Project: <https://github.com/t2-project/kube>

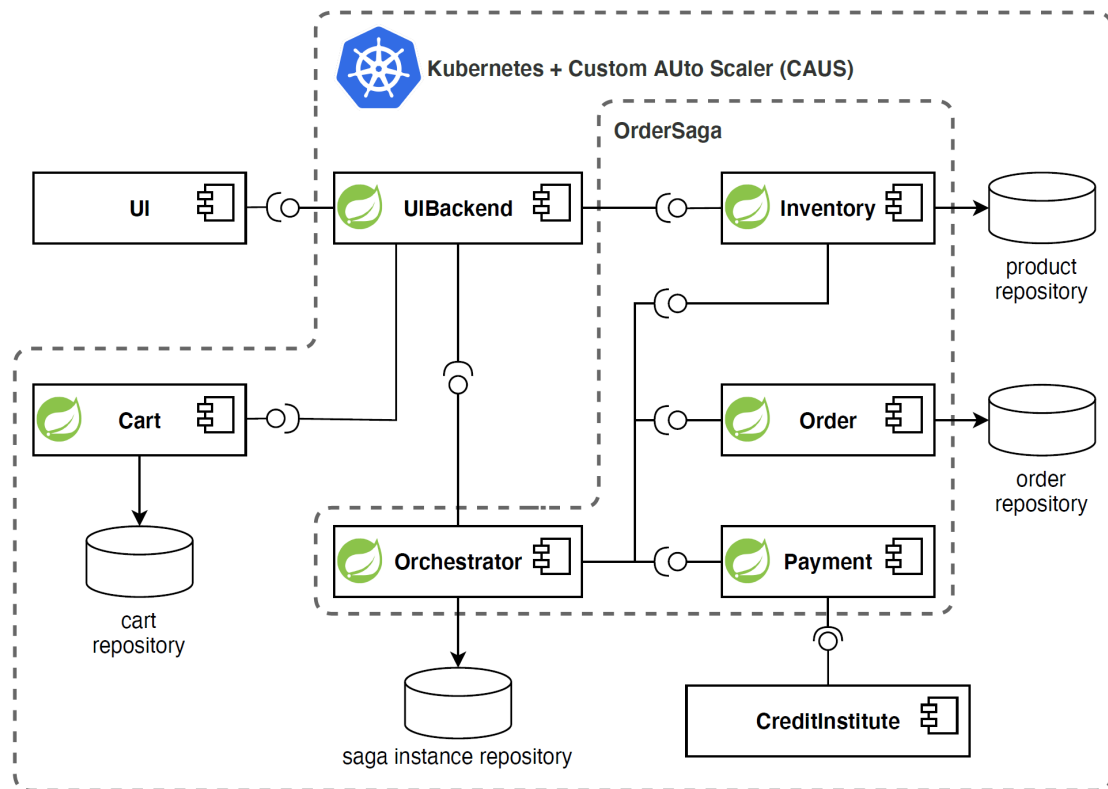


Figure 5.1: Component Diagram of the T2 Project [SSB22].

The Bash script also deploys the Kubernetes configuration files to a Kubernetes cluster. As a result, we just need to execute one command for deploying the complete deployment model. To further automate the deployment of the T2 Project, we created a Terraform deployment model that deploys a Kubernetes cluster on the Azure cloud platform. Azure is a public cloud, which mainly provides PaaS offerings. We extended the Bash-script so that it first executes the Terraform deployment model to deploy the Kubernetes cluster on Azure. The following Helm and Kubernetes commands deploy the services on this Kubernetes cluster. We provide this exemplary technology-specific deployment model in a GitHub repository⁴. Figure 5.2 gives an overview of the structure of this deployment model.

The Bash-script, called “azure-start.sh”, is the overarching deployment model that executes five embedded deployment models. These are the Terraform deployment model in the “terraform/” folder, the Kubernetes deployment model in the “k8/” folder, and the three Helm charts with the names “mongo-order”, “mongo-cart”, and “kafka”. The Helm charts contain further embedded Kubernetes deployment models under their “templates/” folder. Kafka contains an embedded Helm chart for Zookeeper, which is a service that Kafka relies on for storing configuration data.

⁴GitHub repository with the exemplary technology-specific deployment model: <https://github.com/Well15a/kube/tree/terraform>

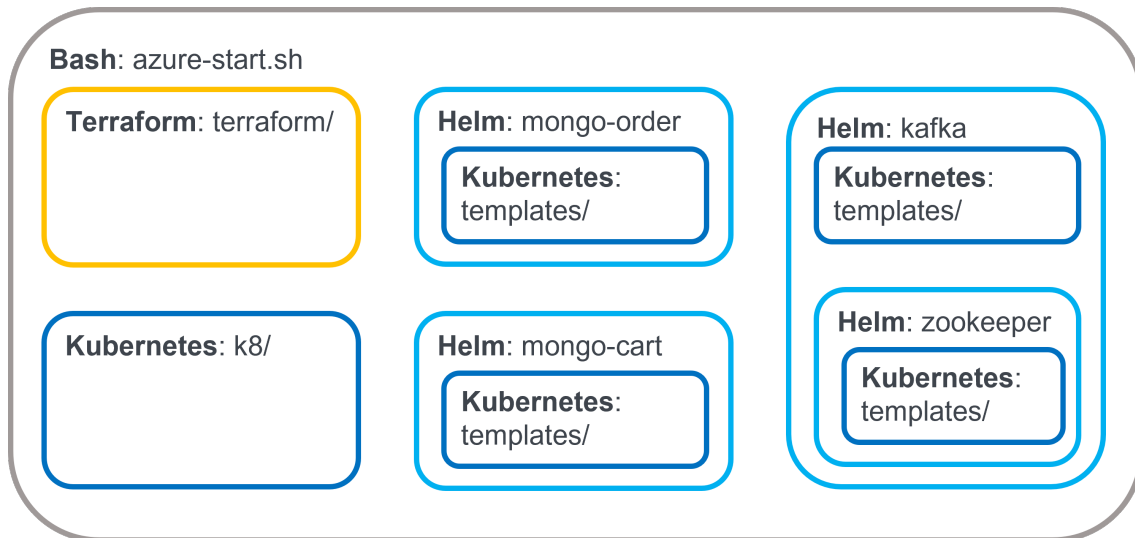


Figure 5.2: Overview of the exemplary technology-specific deployment model. It shows the structure of the deployment model with all embedded deployment models and the deployment technologies.

Based on this, we manually created a technology-agnostic deployment model that we expect as a result from the transformation framework (see Listing A.1). For that, we created an EDMM component for each service and database of the T2 Project and added properties from environmental variables and port definitions that we could find in the deployment models. From the Kubernetes cluster described in the Terraform deployment model, we expect that the transformation framework finds a component representing the provided infrastructure resources, an operating system component, and a container runtime component that enables running containerized applications. Except for the database components, we created a component type for each component. The database components should share component types for a PostgreSQL database and a MongoDB database. We looked at the documentation of the T2 Project and the files of the deployment models to determine which relations should be present between the components. Additionally, we deployed the T2 Project and investigated the provisioned services. Furthermore, we executed the provided JMeter load tests and used the tracing tool Jaeger to determine request routes for confirming and finding new relations.

5.3 Results

For the generation of the evaluation results, we started all services of the transformation framework and transformed the exemplary technology-specific deployment model. The files that we retrieved correspond to the metrics in Table 5.1. We provide the most important parts of this result in Appendix B. For the complete result with all files, refer to the repository on Zenodo⁵ or GitHub⁶. During the run we did not observe any errors or unexpected behavior.

⁵Zenodo repository with complete evaluation results: <https://doi.org/10.5281/zenodo.6503667>

⁶GitHub repository with complete evaluation results: <https://github.com/UST-TAD/evaluation-results>

The log files of the Java applications (M1) show the flow of actions during the plugin registration and the transformation process. Listing B.1 shows a summary of the most important logged events from the viewpoint of the analysis manager. From the logs of the analysis manager and the plugins, we can see that all plugins registered at the analysis manager. The list of registered plugins in the configurations database (M2) confirms this, because it contains one entry for each plugin (see Listing B.2). The definitions of the AMQP entities on the RabbitMQ message broker (M3) reflect this as well (see Listing B.3). It contains definitions of a fanout exchange for the analysis task response queue and a headers exchange for the analysis task request queue (see Listing B.3, line 77). For each plugin, it contains a queue (see Listing B.3, line 40) and a corresponding binding (see Listing B.3, line 97) to the headers exchange. The binding definition includes the binding arguments that the transformation framework correctly set to the corresponding analysis technique and deployment technology.

In the log file of the analysis manager, we can observe the point when we issued the transform command (see Listing B.1, line 6). The log files show that the analysis manager then sent a request to the models service for initializing the internal deployment models. After that, the analysis manager created the first analysis task and sent it to the Bash plugin. The plugin received this request and started with the transformation. It requested the internal deployment models from the models service and later sent requests to update them. It found five embedded deployment models and sent corresponding embedded deployment model analysis requests to the analysis manager. The transformation was successful and the Bash plugin notified the analysis manager about it. The analysis manager then tried to find a dynamic plugin for the Bash deployment technology but was unsuccessful. Therefore, the analysis manager continued with sending the analysis tasks of the embedded deployment models. First, the analysis manager requested the analysis of the Terraform deployment model, then the three Helm charts, and finally the Kubernetes deployment model. For each task, the analysis manager waited for a successful response and then searched for a plugin using the dynamic analysis technique. As this was always unsuccessful, the analysis manager proceeded with the next analysis task. In the cases of the Helm charts, the Helm plugin detected the embedded Kubernetes deployment models during the analysis. For each analysis task that the Helm plugin successfully completed, the analysis manager directly sent an analysis task for the analysis of the detected embedded Kubernetes deployment model to the Kubernetes plugin. Finally, the analysis manager requested the results from the models service.

The internal technology-specific deployment model (M4) points out which parts of the exemplary technology-specific deployment model the transformation framework could detect. It shows one big difference from the general structure presented in Figure 5.2. The embedded Helm chart for the Zookeeper service, including its embedded Kubernetes deployment model, is missing in the internal technology-specific deployment model. Apart from this, all other embedded deployment models were detected. For the Terraform deployment model in the “terraform” directory and the Kubernetes deployment model in the “k8/” directory, the transformation framework could identify all relevant files and add them as deployment model contents.

At the end of the transformation process, the analysis manager prints the result to the tad-shell. This result is presented in Listing 5.1. It contains the path to the created technology-agnostic deployment model, which we provide in Listing B.4. The comparison of this actual technology-agnostic deployment model to the expected one that we created manually (see Listing A.1) yields the following differences (M5). First, the transformation framework was not able to find the expected “connects to” relation from the “kafka” component to the “kafka-zookeeper” component.

```
1 Transformation process finished!
2 Find the technology-agnostic deployment model under the following path:
3 /home/ubuntu/tad/tadms/fc6750c3-dda2-4d04-8116-5808b7f37324.yaml
4 Analysis Progress: 99.50877192982456%
5 Comprehensibility: 64.98245614035088%
6 Confidence: 91.33858267716536%
7 Type Completeness Val1: 99.4535519125683%
8 Type Completeness Val2: 98.24561403508771%
```

Listing 5.1: Output of the transformation result to the CLI of the transformation framework. It shows the location of the technology-agnostic deployment model and the calculated values for the metrics.

Additionally, both components miss one property each that are specified as environmental variables in their respective Kubernetes deployment object definitions. It also did not add the service endpoints with their respective port numbers of the zookeeper service as properties to the “kafka-zookeeper” component. Furthermore, we expected the transformation framework to define one component type for a PostgreSQL database and one component type for a MongoDB database. The components should then be able to reuse this type. Instead, the transformation framework created a separate type for each database. The components “postgres-orchestrator” and “postgres-inventory” have distinct component types with almost identical property definitions. The same applies for the “mongo-order” and “mongo-cart” components.

In addition to the file path of the technology-agnostic deployment model, Listing 5.1 shows the calculated values of the metrics (M6). The analysis progress and both type completeness metrics are at almost 100 percent. The confidence metric is a bit lower and at around 91 percent while the comprehensibility metric has the worst result with almost 65 percent.

5.4 Discussion

From the results, we conclude that the transformation framework is generally able to transform technology-specific deployment models into technology-agnostic deployment models. The application logs show that the transformation framework follows the conceptualized processes for the plugin registration and the transformation. Furthermore, the analysis manager correctly registers the plugins and persists them in the configurations database, and creates all required AMQP entities on the RabbitMQ message broker.

However, although the transformation framework was able to find most of the information in the technology-specific deployment model, it did not create a complete technology-agnostic deployment model. The result shows differences between the internal technology-agnostic deployment model and the originating technology-specific deployment model, as well as between the expected and actual technology-agnostic deployment models. We can trace back the differences of the internal technology-specific deployment model to the Helm plugin. It cannot detect embedded Helm charts that a given Helm chart depends on and create an embedded deployment model for it. However, the transformation framework could still detect the Zookeeper component that the embedded Helm chart in the Kafka Helm chart describes. The reason for this is that the “template” command executed by

the Helm plugin outputs not only the Kubernetes configuration files of the given Helm chart but also includes all Kubernetes configuration files of embedded Helm charts. Consequently, the transformation framework was still able to detect all information. Only the internal technology-specific deployment model is not a correct representation of the structure of the originating technology-specific deployment model. This may seem unproblematic for now but may become an issue when the transformation framework needs to execute further transformation steps, e.g., for transforming embedded Docker images. Therefore, we need to improve the Helm plugin so that it can detect embedded Helm charts, which should enable the transformation framework to transform them separately.

In the case of the differences between the actual and expected technology-agnostic deployment models, we suspect that the missing relation between the Zookeeper and Kafka components relates to the missing properties in these components. Because the Zookeeper component misses information about external ports, the Kubernetes plugin may not have been able to detect the relation. To find the cause for these missing properties and fix this issue, we require further tests and improvements to the Kubernetes plugin.

The type system for components and relations introduced by EDMM can be used to improve the conciseness and comprehensibility of the technology-agnostic deployment model. By creating one component type for each component, the transformation framework does not leverage the potential of this system. Therefore, the transformation framework requires improvements in this regard. Ideally, it should be able to detect similar components and create shared component types. Alternatively, the transformation framework could provide a predefined component type hierarchy that the plugins then use to match found components to one of the types in this hierarchy. However, the definition of such a general hierarchy applicable in various specific use cases is not trivial and requires further research.

From the metrics in the transformation result, we can assess the quality of the technology-agnostic deployment model. The analysis progress metric shows a result of almost 100 percent. The missing percentages come from initial contents that were created but not replaced or removed properly during the analysis. Therefore, we can infer that the transformation framework analyzed the detected part of the technology-specific deployment model entirely.

In contrast, the comprehensibility metric has the lowest score with about 65 percent. This shows that the transformation framework was not able to comprehend a significant part of the technology-specific deployment model and could therefore not use it for the transformation into EDMM entities. By looking at the internal technology-agnostic deployment model in the models database, we can see that the not comprehended parts mainly refer to Kubernetes objects of types that the Kubernetes plugin does not support. The other plugins also show some parts that they could not understand. Here we can identify the potential to improve the transformation framework and the completeness of the resulting technology-agnostic deployment model. For that, we need to extend the plugins so that they can understand more concepts that the respective deployment technologies offer. Especially the Kubernetes plugin could benefit from the support of more Kubernetes object types.

The transformation framework is primarily confident about the EDMM entities in the technology-agnostic deployment model, as the confidence metric score of about 91 percent indicates. EDMM entities with a confidence set to “suspected” are mainly relations between components that different plugins created. One example of this are the “hosted on” relations between the components discovered by the Kubernetes plugin and the container runtime component that the Terraform plugin

created. This indicates that the plugins struggle to communicate findings that may be important for the analysis of other plugins. For this purpose, we initially introduced the internal technology-specific deployment model. We need to investigate if this model needs to be adjusted or if there are other possibilities to denote this information for the future processing of other plugins.

Both type completeness metrics are almost at 100 percent. This is no surprise as each component has their own component type with respective property definitions. Therefore, the significance of these metrics for assessing the quality of the technology-agnostic deployment model is low. For this to improve, the transformation framework first needs to change its approach to the EDMM type system, as described above.

In summary, we assert that the technology-agnostic deployment model is of good quality. However, it is not complete as some information is missing. Especially the plugins leave room for further improvements.

In comparison with the requirements described in Section 3.4.1, we conclude that the transformation framework completely fulfills the requirements 1, 3, and 4. The evaluation showed that the transformation framework was able to transform the technology-specific deployment model and correctly implemented the conceptualized processes (see Requirement 1). The resulting technology-agnostic deployment model contains most of the entities that we expected, but some information is missing. Therefore, the transformation framework requires further development effort and improvements so that it can create complete technology-agnostic deployment models that do not miss any information about the application deployment, as described by Requirement 2. On the other hand, the transformation framework is extensible and able to transform technology-specific deployment models created from a combination of different deployment technologies through the plugin-based approach (see Requirement 3 and 4).

5.5 Threats to Validity

In the following, we present threats to the validity of this evaluation, based on the classification given by Runeson and Höst [RH09].

5.5.1 Construct Validity

For the evaluation, we provided a prototypical realization of the presented concept. This presents just one specific way on how the concept can be realized. Naturally, it is possible to implement this in various ways, for example using different technologies or alternative transformation approaches for the plugins. Consequently, it is possible that such an alternative implementation of the concepts could lead to a different outcome of the evaluation. However, as long as the concepts are correctly implemented, the outcome should not differ too much. Therefore, we evaluated if the prototypical realization conforms to the conceptualized processes that we defined.

5.5.2 Internal Validity

The log files of the Java applications show process steps that refer to locations in the code that we selected. This may not show the actual process flow because it is possible that processing steps that deviate from the concept happen between individual log statements. Therefore, we made sure to cover all endpoints of the applications so that we are able to recognize all calls from and to external services. To confirm this or to gain additional insights into the process flow of the transformation framework, it could be beneficial to investigate it with the help of a tracing tool like Jaeger.

The information about the application deployment in the expected technology-agnostic deployment model is limited to our knowledge of the technology-specific deployment model. It may deploy additional components or communication between certain services are happening that we were unable to discover. Therefore, it is possible that some information is missing in both the expected and the actual technology-agnostic deployment model that we are unaware of. To counter this, we inspected the exemplary technology-specific deployment model as thoroughly as possible, which included the actual deployment of the T2 Project.

The score of the metrics is calculated based on the values that the plugins record in the internal deployment models. If the plugins do not implement this as intended, the metrics might not be representative for what we expect and may falsify the result. We tried to counter this by manually inspecting the recorded metrics in the internal deployment models in the models database. This investigation showed that the plugins seem to record the metrics as intended.

5.5.3 External Validity

For the evaluation, we transformed a technology-specific deployment model of one specific application. The results of this cannot be generalized for technology-specific deployment models of other applications, as they may use concepts of deployment technologies that the plugins currently do not support. Furthermore, we only developed four plugins. Therefore, the usage of any deployment technology that is not supported by the transformation framework would lead to substantially worse results. In this first realization of the transformation framework we do not intend to develop a solution that works for every use case. Rather, we want to confirm that the presented concepts are suitable and therefore conducted a thorough evaluation with one technology-specific deployment model.

5.5.4 Reliability

In this chapter, we evaluated the result of one specific run for transforming the exemplary technology-specific deployment model with the transformation framework. To confirm the reliability of these results, we repeated this run several times. This showed that the calculated score of the metrics and the content of the technology-agnostic deployment model are always the same. Therefore, we assert that the evaluation result is reliable in this regard.

6 Conclusion

To conclude the thesis, we first summarize our contribution and results in Section 6.1. After that we discuss benefits (Section 6.2) and limitations (Section 6.3) of the presented work. Finally, we describe lessons learned during the research on this topic in Section 6.4 and conclude with an outlook on possible future work in Section 6.5.

6.1 Summary

The main contribution of the thesis is a concept and architecture for a plugin-based transformation framework to transform technology-specific deployment models into technology-agnostic deployment models based on the EDMM. The transformation framework can detect embedded deployment models and transform them with an appropriate plugin. It also supports the addition and combination of alternative transformation methods, e.g., mixing static and dynamic analysis techniques to achieve the best possible results. We described in detail the transformation process that the transformation framework should implement. Additionally, we provided definitions of metrics that the transformation framework includes in the transformation result so that users can assess the quality of the technology-agnostic deployment model.

We implemented this concept in a prototype that contains four plugins for different deployment technologies. With this prototypical realization, we evaluated the concepts by transforming a technology-specific deployment model that describes an example application based on a reference architecture. This evaluation showed that the transformation framework can produce technology-agnostic deployment models of good quality and that the concepts work. However, the evaluation also showed some weaknesses of the transformation framework, especially in the implementation of the plugins. To address these issues, we discussed ways to improve the transformation framework.

6.2 Benefits

Previous, related work focused mainly on the transformation of one or few specific deployment technologies. With this work, we provide a more general approach that enables the addition of support for arbitrary technologies. We see it as a baseline for the transformation of technology-specific deployment models to technology-agnostic deployment models. We can now build on the provided concepts for future work, improve the prototype of the transformation framework based on the evaluation result and extend it. Other researchers or developers can use the core components of the prototypical realization of the transformation framework and provide their own plugins for their use cases and applied deployment technologies.

6.3 Limitations

The transformation frameworks' ability to transform deployment models is limited by the number of supported deployment technologies. Therefore, we provided an extensible architecture that allows adding plugins to support more deployment technologies.

Another limitation is that the concept does not provide means to deploy a resulting technology-agnostic deployment model. Because we focus on the transformation, this is out of scope. We chose the EDMM as the meta-model, which does not provide a deployment engine for deploying it to a deployment environment. Therefore, a subsequent deployment of the technology-agnostic deployment model is currently not possible. To enable this, we can use the EDMM transformation system presented in [WBB+19] to transform the technology-agnostic deployment model back into a technology-specific deployment model. However, we did not test this and cannot guarantee that a resulting deployment completely conforms to the original technology-specific deployment model.

6.4 Lessons Learned

The realization of the prototype showed, that achieving transformations that retain all relevant information for different deployment models with arbitrary deployment technologies requires lots of development effort. Since there are many deployment technologies in use today, we would have to develop many plugins accordingly. Therefore, we should focus on the most popular deployment technologies first. Furthermore, each deployment technology provides different concepts and corresponding challenges. Consequently, the approach and implementation of the transformation logic differ drastically between the plugins. This increases the development effort because it is hard to reuse parts of existing implementations. Additionally, many deployment technologies provide many concepts and entities. For example, Terraform allows using over 1700 providers that each contain their own set of resource types, data sources, and more. To avoid defining distinct mappings to EDMM entities for each resource type, we need to find more general approaches for the transformation in such cases, for example, through the dynamic analysis technique. We also need to take into account that deployment technologies evolve. With updates, they may introduce new concepts and change or deprecate existing ones. Therefore, it is crucial to update the plugins for these new changes. Because of this, some plugins may also need to differentiate between major versions of the deployment technology that a given deployment model uses.

If a deployment model uses managed services of a cloud provider, it is often hard to find out what they deploy in detail. This is especially difficult in the cases of PaaS and SaaS offerings. Cloud providers abstract the underlying resources from the user so that they do not have to care about them. However, for the transformation it is crucial to know this. The transformation framework should create appropriate EDMM components that are independent of specific cloud providers.

From the evaluation, we learned that we need to make further improvements to the plugins, the usage of the EDMM type system, and the communication of findings between the plugins.

6.5 Future Work

For future work, we can identify several interesting topics. First, it is important to improve the prototypical realization based on the findings of the evaluation. After that, it would be interesting to conduct a broader evaluation that includes the transformation of several technology-specific deployment models. From the findings, we can then assess if we can generalize on the results of the first evaluation.

Future work may continue the development of the prototypical realization of the transformation framework. This includes reworking and extending the existing plugins and the addition of new plugins for the support of more deployment technologies. Furthermore, the development of plugins that follow a dynamic analysis approach can contribute to the improvement of the transformation framework. This will enable further research in several ways. First, it can conduct a comparison between static and dynamic analysis techniques. Findings may show certain areas or categories of deployment technologies where one technique outperforms the other. Alternatively, future work can research the effect of combining these techniques to improve the transformation results of a specific deployment technology.

Another interesting research topic is the improvement of the EDMM component type hierarchy that the transformation framework creates. In Section 5.4 we already outlined two options. The first option is to change the plugin implementation so that they can detect commonalities in the components and create shared component types. The other option focuses on defining a general component type hierarchy that the plugins can use for transforming arbitrary deployment models. It would be interesting to know if there is already related research on this topic, how such a hierarchy looks like, and if it helps to improve the results of the transformation framework.

Moreover, future work may focus on the end user of the transformation framework. For the improvement of the metrics in the transformation result, it is helpful to evaluate how they support the end user in assessing the quality of the technology-agnostic deployment model. In the case of the general user experience, it could be beneficial to replace the existing tad-shell CLI with a more powerful Graphical User Interface (GUI). This would enable the implementation of more advanced features, like the real-time tracking of the transformation process with the current status of tasks, discovered embedded deployment models, and plugins.

We can also identify interesting topics for the conduction of user studies. One possible topic is to research if users can comprehend the resulting technology-agnostic deployment model of the transformation result better than the originating technology-specific deployment model. Alternatively, a user study may investigate if there are better alternatives to the EDMM in YAML specification that we currently use as a textual concrete syntax for the technology-agnostic deployment model. In this case, the aforementioned GUI would enable to include graphical syntaxes in this comparison.

Finally, future work could also investigate whether the model actually contains enough information to deploy it and how this is possible. For this, they could test and compare approaches that deploy the technology-agnostic deployment model directly, or transform it into an intermediate format first.

Bibliography

- [BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. “Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies”. In: *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*. Springer, 2013. doi: [10.1007/978-3-642-41030-7_9](https://doi.org/10.1007/978-3-642-41030-7_9) (cit. on p. 1).
- [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: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, Mar. 2014, pp. 87–96. doi: [10.1109/IC2E.2014.56](https://doi.org/10.1109/IC2E.2014.56) (cit. on pp. 1, 6).
- [BCR94] V. R. Basili, G. Caldiera, H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 49).
- [CFH+98] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, A. L. Wolf. *A characterization framework for software deployment technologies*. Tech. rep. Colorado State Univ Fort Collins Dept of Computer Science, 1998 (cit. on p. 4).
- [EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. “Declarative vs. imperative: two modeling patterns for the automated deployment of applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS). 2017, pp. 22–27 (cit. on pp. 1, 4).
- [EBLW17] C. Endres, U. Breitenbücher, F. Leymann, J. Wettinger. “Anything to Topology - A Method and System Architecture to Topologize Technology-specific Application Deployment Artifacts.” In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SCITEPRESS – Science and Technology Publications, Lda, 2017, pp. 180–190. doi: [10.5220/0006305302080218](https://doi.org/10.5220/0006305302080218) (cit. on pp. 1, 4, 6–8).
- [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 pp. 1, 6).
- [Has21] HashiCorp, Inc. *Terraform Language Documentation*. Dec. 15, 2021. URL: <https://www.terraform.io/language> (cit. on p. 42).
- [Has22] HashiCorp, Inc. *What is Terraform?* Jan. 27, 2022. URL: <https://www.terraform.io/intro> (cit. on pp. 41, 42).

- [HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. “Pattern-based Deployment Models and Their Automatic Execution”. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing UCC 2018, 17–20 December 2018, Zurich, Switzerland*. IEEE Computer Society, 2018, pp. 41–52. DOI: [10.1109/UCC.2018.00013](https://doi.org/10.1109/UCC.2018.00013) (cit. on pp. 1, 4, 6).
- [Hel22] Helm Authors. *Helm Architecture*. 2022. URL: <https://helm.sh/docs/topics/architecture/> (cit. on p. 47).
- [LSS+13] H. Lu, M. Shtern, B. Simmons, M. Smit, M. Litoiu. “Pattern-based deployment service for next generation clouds”. In: *2013 IEEE Ninth World Congress on Services*. IEEE, 2013, pp. 464–471 (cit. on p. 6).
- [MG11] P. Mell, T. Grance. “The NIST definition of cloud computing”. In: (2011) (cit. on pp. 3, 5).
- [Mon21] MongoDB, Inc. *Introduction to MongoDB*. 2021. URL: <https://www.mongodb.com/docs/manual/introduction/> (cit. on p. 37).
- [Mor16] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. Safari Books Online. O’Reilly Media, 2016. ISBN: 9781491924396. URL: <https://books.google.de/books?id=BIhRDAAAQBAJ> (cit. on p. 5).
- [Org13] Organization for the Advancement of Structured Information Standards (OASIS). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Jan. 2013. URL: <http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html> (cit. on p. 6).
- [RF20] C. Ramey, B. Fox. *Bash Reference Manual*. Dec. 21, 2020. URL: <https://www.gnu.org/software/bash/manual/bash.pdf> (cit. on p. 47).
- [RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empir. Softw. Eng.* 14.2 (2009), pp. 131–164 (cit. on p. 57).
- [Ric21] C. Richardson. *Pattern: Saga*. 2021. URL: <https://microservices.io/patterns/data/saga.html> (cit. on p. 51).
- [SSB22] S. Speth, S. Stieß, S. Becker. “A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis”. In: *Companions Proceedings of 19th IEEE International Conference on Software Architecture (ICSA-C 2022)*. IEEE, Mar. 2022 (cit. on pp. 51, 52).
- [Tar72] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160 (cit. on p. 7).
- [The22a] The Kubernetes Authors. *Command line tool (kubect1)*. 2022. URL: <https://kubernetes.io/docs/reference/kubect1/> (cit. on p. 47).
- [The22b] The Kubernetes Authors. *Understanding Kubernetes Objects*. Feb. 24, 2022. URL: <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/> (cit. on p. 44).
- [The22c] The Kubernetes Authors. *What is Kubernetes?* Apr. 4, 2022. URL: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> (cit. on p. 43).

- [The22d] The PostgreSQL Global Development Group. *What Is PostgreSQL?* 2022. URL: <https://www.postgresql.org/docs/14/intro-what-is.html> (cit. on p. 36).
- [VMw22a] VMware, Inc. *AMQP 0-9-1 Model Explained*. 2022. URL: <https://www.rabbitmq.com/tutorials/amqp-concepts.html> (cit. on p. 37).
- [VMw22b] VMware, Inc. *RabbitMQ*. 2022. URL: <https://www.rabbitmq.com/> (cit. on p. 37).
- [VMw22c] VMware, Inc. *Spring AMQP*. 2022. URL: <https://spring.io/projects/spring-amqp> (cit. on p. 38).
- [VMw22d] VMware, Inc. *Spring Boot*. 2022. URL: <https://spring.io/projects/spring-boot> (cit. on p. 36).
- [VMw22e] VMware, Inc. *Spring Shell*. 2022. URL: <https://spring.io/projects/spring-shell> (cit. on p. 39).
- [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: *ICSOC Workshops*. 2019, pp. 294–298 (cit. on pp. 8, 60).
- [WBB+20a] M. Wurster, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, J. Soldani. “Technology-Agnostic Declarative Deployment Automation of Cloud Applications”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2020, pp. 97–112 (cit. on pp. 4, 8).
- [WBB+20b] M. Wurster, U. Breitenbücher, A. Brogi, F. Leymann, J. Soldani. “Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications.” In: *CLOSER*. 2020, pp. 171–180 (cit. on p. 6).
- [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. 1, 4, 5, 8, 12, 13, 16, 24, 25, 42).
- [WBKL16] J. Wettinger, U. Breitenbücher, O. Kopp, F. Leymann. “Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel”. In: *Future Generation Computer Systems* 56 (2016), pp. 317–332 (cit. on pp. 4, 7, 8, 47).

All links were last followed on April 28, 2022.

A Expected Technology-Agnostic Deployment Model

```
1 ---
2 properties: []
3 components:
4   - default:
5     type: "physical_node"
6     description: null
7     properties:
8       - cpu_count: 4
9       - ram_GiB: 16
10      - storage_GiB: 32
11     operations: []
12     artifacts: []
13   - default-operating-system:
14     type: "operating_system"
15     description: null
16     properties:
17       - name: "Ubuntu"
18       - version: "18.04"
19       - os_family: "Linux"
20     operations: []
21     artifacts: []
22   - default-container-runtime:
23     type: "container_runtime"
24     description: null
25     properties:
26       - name: "containerd"
27     operations: []
28     artifacts: []
29   - mongo-cart:
30     type: "mongodb"
31     description: null
32     properties:
33       - container-port_mongodb: 27017
34       - MONGODB_DISABLE_SYSTEM_LOG: "\"no\""
35       - ALLOW_EMPTY_PASSWORD: "\"yes\""
36       - BITNAMI_DEBUG: "\"false\""
37       - MONGODB_SYSTEM_LOG_VERBOSITY: "\"0\""
38       - MONGODB_ENABLE_IPV6: "\"no\""
39       - MONGODB_DISABLE_JAVASCRIPT: "\"no\""
40       - MONGODB_ENABLE_JOURNAL: "\"yes\""
41       - MONGODB_ENABLE_DIRECTORY_PER_DB: "\"no\""
42       - external-port_mongodb: "27017:mongodb"
```

A Expected Technology-Agnostic Deployment Model

```
43     operations: []
44     artifacts:
45       - docker_image:
46         name: docker.io/bitnami/mongodb:4.4.10-debian-10-r44
47         fileURI: "-"
48   - mongo-order:
49     type: "mongodb"
50     description: null
51     properties:
52       - container-port_mongodb: 27017
53       - BITNAMI_DEBUG: "false"
54       - MONGODB_DISABLE_SYSTEM_LOG: "\"no\""
55       - ALLOW_EMPTY_PASSWORD: "\"yes\""
56       - BITNAMI_DEBUG: "\"false\""
57       - MONGODB_SYSTEM_LOG_VERBOSITY: "\"0\""
58       - MONGODB_ENABLE_IPV6: "\"no\""
59       - MONGODB_DISABLE_JAVASCRIPT: "\"no\""
60       - MONGODB_ENABLE_JOURNAL: "\"yes\""
61       - MONGODB_ENABLE_DIRECTORY_PER_DB: "\"no\""
62       - external-port_mongodb: "27017:mongodb"
63     operations: []
64     artifacts:
65       - docker_image:
66         name: docker.io/bitnami/mongodb:4.4.10-debian-10-r44
67         fileURI: "-"
68   - kafka-zookeeper:
69     type: "zookeeper"
70     description: null
71     properties:
72       - container-port_client: 2181
73       - container-port_follower: 2888
74       - container-port_election: 3888
75       - ZOO_4LW_COMMANDS_WHITELIST: "\"srvr, mntr, ruok\""
76       - ALLOW_ANONYMOUS_LOGIN: "\"yes\""
77       - ZOO_SYNC_LIMIT: "\"5\""
78       - ZOO_MAX_SESSION_TIMEOUT: "\"40000\""
79       - ZOO_TICK_TIME: "\"2000\""
80       - ZOO_PRE_ALLOC_SIZE: "\"65536\""
81       - ZOO_SERVERS: "kafka-zookeeper-0.kafka-zookeeper-headless.default.
82         svc.cluster.local:2888:3888:1"
83       - ZOO_AUTOPURGE_RETAIN_COUNT: "\"3\""
84       - ZOO_SNAPCOUNT: "\"100000\""
85       - ZOO_LOG_LEVEL: "\"ERROR\""
86       - ZOO_MAX_CLIENT_CNXNS: "\"60\""
87       - ZOO_INIT_LIMIT: "\"10\""
88       - BITNAMI_DEBUG: "\"false\""
89       - ZOO_ENABLE_AUTH: "\"no\""
90       - ZOO_DATA_LOG_DIR: "\"\""
91       - ZOO_LISTEN_ALL_IPS_ENABLED: "\"no\""
92       - ZOO_PORT_NUMBER: "\"2181\""
93       - ZOO_AUTOPURGE_INTERVAL: "\"0\""
94       - ZOO_HEAP_SIZE: "\"1024\""
95       - POD_NAME: kafka-zookeeper
```

```

95     - external-port_tcp-client: "2181:client"
96     - external-port_follower: "2888:follower"
97     - external-port_tcp-election: "3888:election"
98     operations: []
99     artifacts:
100    - docker_image:
101      name: docker.io/bitnami/zookeeper:3.7.0-debian-10-r188
102      fileURI: "-"
103  - kafka:
104    type: "kafka"
105    description: null
106    properties:
107      - external-port_kafka-client: 9092
108      - external-port_kafka-internal: 9093
109      - MY_POD_NAME: "kafka"
110      - KAFKA_CFG_SOCKET_RECEIVE_BUFFER_BYTES: "\"102400\""
111      - KAFKA_CFG_SOCKET_SEND_BUFFER_BYTES: "\"102400\""
112      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP: "\"INTERNAL:PLAINTEXT,
CLIENT:PLAINTEXT\""
113      - KAFKA_CFG_OFFSETS_TOPIC_REPLICATION_FACTOR: "\"1\""
114      - KAFKA_CFG_LOG_FLUSH_INTERVAL_MS: "\"1000\""
115      - KAFKA_CFG_ZOOKEEPER_CONNECTION_TIMEOUT_MS: "\"6000\""
116      - KAFKA_CFG_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: "\"1\""
117      - KAFKA_CFG_LOG_SEGMENT_BYTES: "\"1073741824\""
118      - KAFKA_CFG_DEFAULT_REPLICATION_FACTOR: "\"1\""
119      - KAFKA_CFG_SUPER_USERS: "\"User:admin\""
120      - KAFKA_CFG_ADVERTISED_LISTENERS: "\"INTERNAL://$(MY_POD_NAME).
kafka-headless.default.svc.cluster.local:9093,CLIENT://$(MY_POD_NAME).kafka
-headless.default.svc.cluster.local:9092\""
121      - KAFKA_CFG_LISTENERS: "\"INTERNAL://:9093,CLIENT://:9092\""
122      - KAFKA_CFG_TRANSACTION_STATE_LOG_MIN_ISR: "\"1\""
123      - KAFKA_CFG_LOG_RETENTION_HOURS: "\"168\""
124      - KAFKA_CFG_LOG_FLUSH_INTERVAL_MESSAGES: "\"10000\""
125      - BITNAMI_DEBUG: "\"false\""
126      - KAFKA_CFG_ZOOKEEPER_CONNECT: "\"kafka-zookeeper\""
127      - KAFKA_CFG_NUM_NETWORK_THREADS: "\"3\""
128      - KAFKA_CFG_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "\"true\""
129      - KAFKA_CFG_NUM_RECOVERY_THREADS_PER_DATA_DIR: "\"1\""
130      - KAFKA_CFG_LOG_RETENTION_BYTES: "\"1073741824\""
131      - KAFKA_CFG_SOCKET_REQUEST_MAX_BYTES: "\"104857600\""
132      - KAFKA_INTER_BROKER_LISTENER_NAME: "\"INTERNAL\""
133      - KAFKA_CFG_NUM_IO_THREADS: "\"8\""
134      - ALLOW_PLAINTEXT_LISTENER: "\"yes\""
135      - KAFKA_CFG_LOG_RETENTION_CHECK_INTERVALS_MS: "\"300000\""
136      - KAFKA_LOG_DIR: "\"/opt/bitnami/kafka/logs\""
137      - KAFKA_CFG_NUM_PARTITIONS: "\"1\""
138      - KAFKA_CFG_AUTHORIZER_CLASS_NAME: "\"\""
139      - KAFKA_CFG_DELETE_TOPIC_ENABLE: "\"false\""
140      - KAFKA_CFG_MESSAGE_MAX_BYTES: "\"1000012\""
141      - KAFKA_VOLUME_DIR: "\"/bitnami/kafka\""
142      - KAFKA_CFG_LOG_DIRS: "\"/bitnami/kafka/data\""
143      - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE: "\"true\""
144      - KAFKA_HEAP_OPTS: "\"-Xmx1024m -Xms1024m\""

```

A Expected Technology-Agnostic Deployment Model

```
145     - external-port_tcp-internal: "9093:kafka-internal"
146     - external-port_tcp-client: "9092:kafka-client"
147     operations: []
148     artifacts:
149     - docker_image:
150         name: docker.io/bitnami/kafka:2.8.1-debian-10-r57
151         fileURI: "-"
152 - cdcservice:
153     type: "cdcservice-type"
154     description: null
155     properties:
156     - container_port: 8080
157     - EVENTUATELOCAL_CDC_READER_NAME: "PostgresPollingReader"
158     - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
159     - SPRING_PROFILES_ACTIVE: "EventuatePolling"
160     - SPRING_DATASOURCE_PASSWORD: "eventuate"
161     - SPRING_DATASOURCE_TEST_ON_BORROW: "\"true\""
162     - SPRING_DATASOURCE_VALIDATION_QUERY: "SELECT 1"
163     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
164     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181"
165     - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
166     - EVENTUATE_OUTBOX_ID: "\"1\""
167     - JAVA_OPTS: "-Xmx64m"
168     - SPRING_DATASOURCE_USERNAME: "eventuate"
169     - "'8099'": "8099:8080"
170     operations: []
171     artifacts:
172     - docker_image:
173         name: "eventuateio/eventuate-cdc-service:0.12.0.RELEASE"
174         fileURI: "-"
175 - uibackend:
176     type: "uibackend-type"
177     description: null
178     properties:
179     - container_port: 8080
180     - T2_ORCHESTRATOR_URL: "http://orchestrator-cs/order/"
181     - T2_INVENTORY_URL: "http://inventory-cs/inventory/"
182     - T2_RESERVATION_ENDPOINT: "reservation"
183     - JAEGER_HOST: "localhost #todo"
184     - T2_CART_URL: "http://cart-cs/cart/"
185     - external_port: "80:8080"
186     operations: []
187     artifacts:
188     - docker_image:
189         name: "t2project/uibackend:main"
190         fileURI: "-"
191 - cart:
192     type: "cart-type"
193     description: null
194     properties:
195     - container_port: 8080
```

```

196     - T2_CART_TTL: "\"0\""
197     - JAEGER_HOST: "localhost #todo"
198     - MONGO_HOST: "mongo-cart-mongodb"
199     - T2_CART_TASKRATE: "\"0\""
200     - external_port: "80:8080"
201     operations: []
202     artifacts:
203     - docker_image:
204         name: "t2project/cart:main"
205         fileURI: "-"
206 - creditinstitute:
207     type: "creditinstitute-type"
208     description: null
209     properties:
210     - container_port: 8080
211     - http: "80:8080"
212     - external_port: "80:8080"
213     operations: []
214     artifacts:
215     - docker_image:
216         name: "t2project/creditinstitute:main"
217         fileURI: "-"
218 - ui:
219     type: "ui-type"
220     description: null
221     properties:
222     - container_port: 8080
223     - T2_UIBACKEND_URL: "http://uibackend-cs/"
224     - external_port: "80:8080"
225     operations: []
226     artifacts:
227     - docker_image:
228         name: "t2project/ui:main"
229         fileURI: "-"
230 - payment:
231     type: "payment-type"
232     description: null
233     properties:
234     - container_port: 8080
235     - T2_PAYMENT_PROVIDER_DUMMY_URL: "http://creditinstitute-cs/pay"
236     - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
237     - SPRING_DATASOURCE_PASSWORD: "eventuate"
238     - JAEGER_SERVICE_NAME: "payment"
239     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
240     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181"
241 "
242     - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
243     - JAEGER_ENABLE: "\"FALSE\""
244     - SPRING_DATASOURCE_USERNAME: "eventuate"
245     - JAEGER_HOST: "simplest-agent #todo"
246     - T2_PAYMENT_PROVIDER_TIMEOUT: "\"5\""
247     - external_port: "80:8080"

```

A Expected Technology-Agnostic Deployment Model

```
247     operations: []
248     artifacts:
249       - docker_image:
250         name: "t2project/payment:main"
251         fileURI: "-"
252   - orchestrator:
253     type: "orchestrator-type"
254     description: null
255     properties:
256       - container_port: 8080
257       - JAEGER_SERVICE_NAME: "orchestrator"
258       - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
259       - SPRING_DATASOURCE_PASSWORD: "eventuate"
260       - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
261       - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181"
262   "
263     - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
264     - JAEGER_ENABLE: "\"FALSE\""
265     - SPRING_DATASOURCE_USERNAME: "eventuate"
266     - JAEGER_HOST: "simplest-agent #todo"
267     - external_port: "80:8080"
268   operations: []
269   artifacts:
270     - docker_image:
271       name: "t2project/orchestrator:main"
272       fileURI: "-"
273   - postgres-orchestrator:
274     type: "postgres-db"
275     description: null
276     properties:
277       - container_port: 5432
278       - POSTGRES_PASSWORD: "eventuate"
279       - POSTGRES_USER: "eventuate"
280       - USE_DB_ID: "\"true\""
281       - external_port: "5432:5432"
282   operations: []
283   artifacts:
284     - docker_image:
285       name: "eventuateio/eventuate-tram-sagas-postgres:0.18.0.RELEASE"
286   "
287     fileURI: "-"
288   - postgres-inventory:
289     type: "postgres-db"
290     description: null
291     properties:
292       - container_port: 5432
293       - POSTGRES_DB: "inventory"
294       - POSTGRES_PASSWORD: "inventory"
295       - POSTGRES_USER: "inventory"
296       - external_port: "5432:5432"
297   operations: []
298   artifacts:
```

```

297     - docker_image:
298         name: "postgres:14.1"
299         fileURI: "-"
300     - order:
301         type: "order-type"
302         description: null
303         properties:
304             - container_port: 8080
305             - MONGO_HOST: "mongo-order-mongodb"
306             - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
307             - SPRING_DATASOURCE_PASSWORD: "eventuate"
308             - JAEGER_SERVICE_NAME: "order"
309             - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
310             - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181
"
311         - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
312         - JAEGER_ENABLE: "\"FALSE\""
313         - SPRING_DATASOURCE_USERNAME: "eventuate"
314         - JAEGER_HOST: "simplest-agent #todo"
315         - external_port: "80:8080"
316         operations: []
317         artifacts:
318             - docker_image:
319                 name: "t2project/order:main"
320                 fileURI: "-"
321     - inventory:
322         type: "inventory-type"
323         description: null
324         properties:
325             - container_port: 8080
326             - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
327             - INVENTORY_SIZE: "\"25\""
328             - JAEGER_HOST: "simplest-agent #todo"
329             - JAEGER_SERVICE_NAME: "inventory"
330             - SPRING_PROFILE_ACTIVE: "saga"
331             - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
332             - SPRING_DATASOURCE_PASSWORD: "inventory"
333             - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181
"
334         - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-inventory:5432
/inventory"
335         - JAEGER_ENABLE: "\"FALSE\""
336         - T2_INVENTORY_TTL: "\"0\""
337         - T2_INVENTORY_TASKRATE: "\"0\""
338         - SPRING_DATASOURCE_USERNAME: "inventory"
339         - external_port: "80:8080"
340         operations: []
341         artifacts:
342             - docker_image:
343                 name: "t2project/inventory:main"
344                 fileURI: "-"
345     relations:

```

A Expected Technology-Agnostic Deployment Model

```
346 - default-operating-system_HostedOn_default:
347   type: "HostedOn"
348   description: null
349   source: "default-operating-system"
350   target: "default"
351   properties: []
352   operations: []
353 - default-container-runtime_HostedOn_default-operating-system:
354   type: "HostedOn"
355   description: null
356   source: "default-container-runtime"
357   target: "default-operating-system"
358   properties: []
359   operations: []
360 - mongo-cart_HostedOn_default-container-runtime:
361   type: "HostedOn"
362   description: null
363   source: "mongo-cart"
364   target: "default-container-runtime"
365   properties: []
366   operations: []
367 - mongo-order_HostedOn_default-container-runtime:
368   type: "HostedOn"
369   description: null
370   source: "mongo-order"
371   target: "default-container-runtime"
372   properties: []
373   operations: []
374 - kafka-zookeeper_HostedOn_default-container-runtime:
375   type: "HostedOn"
376   description: null
377   source: "kafka-zookeeper"
378   target: "default-container-runtime"
379   properties: []
380   operations: []
381 - kafka_ConnectsTo_kafka-zookeeper:
382   type: "ConnectsTo"
383   description: null
384   source: "kafka"
385   target: "kafka-zookeeper"
386   properties: []
387   operations: []
388 - kafka_HostedOn_default-container-runtime:
389   type: "HostedOn"
390   description: null
391   source: "kafka"
392   target: "default-container-runtime"
393   properties: []
394   operations: []
395 - cdcservice_ConnectsTo_kafka:
396   type: "ConnectsTo"
397   description: null
398   source: "cdcservice"
```

```
399     target: "kafka"
400     properties: []
401     operations: []
402 - cdcservice_ConnectsTo_kafka-zookeeper:
403     type: "ConnectsTo"
404     description: null
405     source: "cdcservice"
406     target: "kafka-zookeeper"
407     properties: []
408     operations: []
409 - cdcservice_ConnectsTo_postgres-orchestrator:
410     type: "ConnectsTo"
411     description: null
412     source: "cdcservice"
413     target: "postgres-orchestrator"
414     properties: []
415     operations: []
416 - cdcservice_HostedOn_default-container-runtime:
417     type: "HostedOn"
418     description: null
419     source: "cdcservice"
420     target: "default-container-runtime"
421     properties: []
422     operations: []
423 - uibackend_ConnectsTo_orchestrator:
424     type: "ConnectsTo"
425     description: null
426     source: "uibackend"
427     target: "orchestrator"
428     properties: []
429     operations: []
430 - uibackend_ConnectsTo_inventory:
431     type: "ConnectsTo"
432     description: null
433     source: "uibackend"
434     target: "inventory"
435     properties: []
436     operations: []
437 - uibackend_ConnectsTo_cart:
438     type: "ConnectsTo"
439     description: null
440     source: "uibackend"
441     target: "cart"
442     properties: []
443     operations: []
444 - uibackend_HostedOn_default-container-runtime:
445     type: "HostedOn"
446     description: null
447     source: "uibackend"
448     target: "default-container-runtime"
449     properties: []
450     operations: []
451 - cart_ConnectsTo_mongo-cart:
```

A Expected Technology-Agnostic Deployment Model

```
452     type: "ConnectsTo"
453     description: null
454     source: "cart"
455     target: "mongo-cart"
456     properties: []
457     operations: []
458 - cart_HostedOn_default-container-runtime:
459     type: "HostedOn"
460     description: null
461     source: "cart"
462     target: "default-container-runtime"
463     properties: []
464     operations: []
465 - creditinstitute_HostedOn_default-container-runtime:
466     type: "HostedOn"
467     description: null
468     source: "creditinstitute"
469     target: "default-container-runtime"
470     properties: []
471     operations: []
472 - ui_ConnectsTo_uibackend:
473     type: "ConnectsTo"
474     description: null
475     source: "ui"
476     target: "uibackend"
477     properties: []
478     operations: []
479 - ui_HostedOn_default-container-runtime:
480     type: "HostedOn"
481     description: null
482     source: "ui"
483     target: "default-container-runtime"
484     properties: []
485     operations: []
486 - payment_ConnectsTo_creditinstitute:
487     type: "ConnectsTo"
488     description: null
489     source: "payment"
490     target: "creditinstitute"
491     properties: []
492     operations: []
493 - payment_ConnectsTo_kafka:
494     type: "ConnectsTo"
495     description: null
496     source: "payment"
497     target: "kafka"
498     properties: []
499     operations: []
500 - payment_ConnectsTo_kafka-zookeeper:
501     type: "ConnectsTo"
502     description: null
503     source: "payment"
504     target: "kafka-zookeeper"
```

```
505     properties: []
506     operations: []
507   - payment_ConnectsTo_postgres-orchestrator:
508     type: "ConnectsTo"
509     description: null
510     source: "payment"
511     target: "postgres-orchestrator"
512     properties: []
513     operations: []
514   - payment_HostedOn_default-container-runtime:
515     type: "HostedOn"
516     description: null
517     source: "payment"
518     target: "default-container-runtime"
519     properties: []
520     operations: []
521   - orchestrator_ConnectsTo_kafka:
522     type: "ConnectsTo"
523     description: null
524     source: "orchestrator"
525     target: "kafka"
526     properties: []
527     operations: []
528   - orchestrator_ConnectsTo_kafka-zookeeper:
529     type: "ConnectsTo"
530     description: null
531     source: "orchestrator"
532     target: "kafka-zookeeper"
533     properties: []
534     operations: []
535   - orchestrator_ConnectsTo_postgres-orchestrator:
536     type: "ConnectsTo"
537     description: null
538     source: "orchestrator"
539     target: "postgres-orchestrator"
540     properties: []
541     operations: []
542   - orchestrator_HostedOn_default-container-runtime:
543     type: "HostedOn"
544     description: null
545     source: "orchestrator"
546     target: "default-container-runtime"
547     properties: []
548     operations: []
549   - postgres-orchestrator_HostedOn_default-container-runtime:
550     type: "HostedOn"
551     description: null
552     source: "postgres-orchestrator"
553     target: "default-container-runtime"
554     properties: []
555     operations: []
556   - postgres-inventory_HostedOn_default-container-runtime:
557     type: "HostedOn"
```

A Expected Technology-Agnostic Deployment Model

```
558     description: null
559     source: "postgres-inventory"
560     target: "default-container-runtime"
561     properties: []
562     operations: []
563   - order_ConnectsTo_mongo-order:
564     type: "ConnectsTo"
565     description: null
566     source: "order"
567     target: "mongo-order"
568     properties: []
569     operations: []
570   - order_ConnectsTo_kafka:
571     type: "ConnectsTo"
572     description: null
573     source: "order"
574     target: "kafka"
575     properties: []
576     operations: []
577   - order_ConnectsTo_kafka-zookeeper:
578     type: "ConnectsTo"
579     description: null
580     source: "order"
581     target: "kafka-zookeeper"
582     properties: []
583     operations: []
584   - order_ConnectsTo_postgres-orchestrator:
585     type: "ConnectsTo"
586     description: null
587     source: "order"
588     target: "postgres-orchestrator"
589     properties: []
590     operations: []
591   - order_HostedOn_default-container-runtime:
592     type: "HostedOn"
593     description: null
594     source: "order"
595     target: "default-container-runtime"
596     properties: []
597     operations: []
598   - inventory_ConnectsTo_kafka:
599     type: "ConnectsTo"
600     description: null
601     source: "inventory"
602     target: "kafka"
603     properties: []
604     operations: []
605   - inventory_ConnectsTo_kafka-zookeeper:
606     type: "ConnectsTo"
607     description: null
608     source: "inventory"
609     target: "kafka-zookeeper"
610     properties: []
```

```

611     operations: []
612 - inventory_ConnectsTo_postgres-inventory:
613     type: "ConnectsTo"
614     description: null
615     source: "inventory"
616     target: "postgres-inventory"
617     properties: []
618     operations: []
619 - inventory_HostedOn_default-container-runtime:
620     type: "HostedOn"
621     description: null
622     source: "inventory"
623     target: "default-container-runtime"
624     properties: []
625     operations: []
626 component_types:
627 - BaseType:
628     extends: "-"
629     description: "This is the base type"
630     properties: []
631     operations: []
632 - physical_node:
633     extends: "-"
634     description: null
635     properties:
636     - cpu_count:
637         type: "INTEGER"
638         required: false
639     - ram_GiB:
640         type: "INTEGER"
641         required: false
642     - storage_GiB:
643         type: "INTEGER"
644         required: false
645     operations: []
646 - operating_system:
647     extends: "-"
648     description: null
649     properties:
650     - name:
651         type: "STRING"
652         required: false
653     - version:
654         type: "STRING"
655         required: false
656     - os_family:
657         type: "STRING"
658         required: false
659     operations: []
660 - container_runtime:
661     extends: "-"
662     description: null
663     properties:

```

A Expected Technology-Agnostic Deployment Model

```
664     - name:
665         type: "STRING"
666         required: false
667     - version:
668         type: "STRING"
669         required: false
670     operations: []
671 - mongodb:
672     extends: "-"
673     description: null
674     properties:
675         - container-port_mongodb:
676             type: "INTEGER"
677             required: false
678         - MONGODB_DISABLE_SYSTEM_LOG:
679             type: "STRING"
680             required: false
681         - ALLOW_EMPTY_PASSWORD:
682             type: "STRING"
683             required: false
684         - BITAMI_DEBUG:
685             type: "STRING"
686             required: false
687         - MONGODB_SYSTEM_LOG_VERBOSITY:
688             type: "STRING"
689             required: false
690         - MONGODB_ENABLE_IPV6:
691             type: "STRING"
692             required: false
693         - MONGODB_DISABLE_JAVASCRIPT:
694             type: "STRING"
695             required: false
696         - MONGODB_ENABLE_JOURNAL:
697             type: "STRING"
698             required: false
699         - MONGODB_ENABLE_DIRECTORY_PER_DB:
700             type: "STRING"
701             required: false
702         - external-port_mongodb:
703             type: "STRING"
704             required: false
705     operations: []
706 - kafka-zookeeper-type:
707     extends: "-"
708     description: null
709     properties:
710         - container-port_client:
711             type: "INTEGER"
712             required: false
713         - container-port_follower:
714             type: "INTEGER"
715             required: false
716         - container-port_election:
```

```
717         type: "INTEGER"
718         required: false
719     - ZOO_4LW_COMMANDS_WHITELIST:
720         type: "STRING"
721         required: false
722     - ALLOW_ANONYMOUS_LOGIN:
723         type: "STRING"
724         required: false
725     - ZOO_SYNC_LIMIT:
726         type: "STRING"
727         required: false
728     - ZOO_MAX_SESSION_TIMEOUT:
729         type: "STRING"
730         required: false
731     - ZOO_TICK_TIME:
732         type: "STRING"
733         required: false
734     - ZOO_PRE_ALLOC_SIZE:
735         type: "STRING"
736         required: false
737     - ZOO_SERVERS:
738         type: "STRING"
739         required: false
740     - ZOO_AUTOPURGE_RETAIN_COUNT:
741         type: "STRING"
742         required: false
743     - ZOO_SNAPCOUNT:
744         type: "STRING"
745         required: false
746     - ZOO_LOG_LEVEL:
747         type: "STRING"
748         required: false
749     - ZOO_MAX_CLIENT_CNXNS:
750         type: "STRING"
751         required: false
752     - ZOO_INIT_LIMIT:
753         type: "STRING"
754         required: false
755     - BITNAMI_DEBUG:
756         type: "STRING"
757         required: false
758     - ZOO_ENABLE_AUTH:
759         type: "STRING"
760         required: false
761     - ZOO_DATA_LOG_DIR:
762         type: "STRING"
763         required: false
764     - ZOO_LISTEN_ALLIPS_ENABLED:
765         type: "STRING"
766         required: false
767     - ZOO_PORT_NUMBER:
768         type: "STRING"
769         required: false
```

A Expected Technology-Agnostic Deployment Model

```
770     - ZOO_AUTOPURGE_INTERVAL:
771       type: "STRING"
772       required: false
773     - ZOO_HEAP_SIZE:
774       type: "STRING"
775       required: false
776     - POD_NAME:
777       type: "STRING"
778       required: false
779     - external-port_tcp-client:
780       type: "STRING"
781       required: false
782     - external-port_follower:
783       type: "STRING"
784       required: false
785     - external-port_tcp-election:
786       type: "STRING"
787       required: false
788     operations: []
789   - kafka-type:
790     extends: "-"
791     description: null
792     properties:
793       - external-port_kafka-client:
794         type: "INTEGER"
795         required: false
796       - external-port_kafka-internal:
797         type: "INTEGER"
798         required: false
799       - MY_POD_NAME:
800         type: "STRING"
801         required: false
802       - KAFKA_CFG_SOCKET_RECEIVE_BUFFER_BYTES:
803         type: "STRING"
804         required: false
805       - KAFKA_CFG_SOCKET_SEND_BUFFER_BYTES:
806         type: "STRING"
807         required: false
808       - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP:
809         type: "STRING"
810         required: false
811       - KAFKA_CFG_OFFSETS_TOPIC_REPLICATION_FACTOR:
812         type: "STRING"
813         required: false
814       - KAFKA_CFG_LOG_FLUSH_INTERVAL_MS:
815         type: "STRING"
816         required: false
817       - KAFKA_CFG_ZOOKEEPER_CONNECTION_TIMEOUT_MS:
818         type: "STRING"
819         required: false
820       - KAFKA_CFG_TRANSACTION_STATE_LOG_REPLICATION_FACTOR:
821         type: "STRING"
822         required: false
```

```
823     - KAFKA_CFG_LOG_SEGMENT_BYTES:
824       type: "STRING"
825       required: false
826     - KAFKA_CFG_DEFAULT_REPLICATION_FACTOR:
827       type: "STRING"
828       required: false
829     - KAFKA_CFG_SUPER_USERS:
830       type: "STRING"
831       required: false
832     - KAFKA_CFG_ADVERTISED_LISTENERS:
833       type: "STRING"
834       required: false
835     - KAFKA_CFG_LISTENERS:
836       type: "STRING"
837       required: false
838     - KAFKA_CFG_TRANSACTION_STATE_LOG_MIN_ISR:
839       type: "STRING"
840       required: false
841     - KAFKA_CFG_LOG_RETENTION_HOURS:
842       type: "STRING"
843       required: false
844     - KAFKA_CFG_LOG_FLUSH_INTERVAL_MESSAGES:
845       type: "STRING"
846       required: false
847     - BITNAMI_DEBUG:
848       type: "STRING"
849       required: false
850     - KAFKA_CFG_ZOOKEEPER_CONNECT:
851       type: "STRING"
852       required: false
853     - KAFKA_CFG_NUM_NETWORK_THREADS:
854       type: "STRING"
855       required: false
856     - KAFKA_CFG_ALLOW_EVERYONE_IF_NO_ACL_FOUND:
857       type: "STRING"
858       required: false
859     - KAFKA_CFG_NUM_RECOVERY_THREADS_PER_DATA_DIR:
860       type: "STRING"
861       required: false
862     - KAFKA_CFG_LOG_RETENTION_BYTES:
863       type: "STRING"
864       required: false
865     - KAFKA_CFG_SOCKET_REQUEST_MAX_BYTES:
866       type: "STRING"
867       required: false
868     - KAFKA_INTER_BROKER_LISTENER_NAME:
869       type: "STRING"
870       required: false
871     - KAFKA_CFG_NUM_IO_THREADS:
872       type: "STRING"
873       required: false
874     - ALLOW_PLAINTEXT_LISTENER:
875       type: "STRING"
```

A Expected Technology-Agnostic Deployment Model

```
876         required: false
877     - KAFKA_CFG_LOG_RETENTION_CHECK_INTERVALS_MS:
878         type: "STRING"
879         required: false
880     - KAFKA_CFG_LOG_DIR:
881         type: "STRING"
882         required: false
883     - KAFKA_CFG_NUM_PARTITIONS:
884         type: "STRING"
885         required: false
886     - KAFKA_CFG_AUTHORIZER_CLASS_NAME:
887         type: "STRING"
888         required: false
889     - KAFKA_CFG_DELETE_TOPIC_ENABLE:
890         type: "STRING"
891         required: false
892     - KAFKA_CFG_MESSAGE_MAX_BYTES:
893         type: "STRING"
894         required: false
895     - KAFKA_CFG_VOLUME_DIR:
896         type: "STRING"
897         required: false
898     - KAFKA_CFG_LOG_DIRS:
899         type: "STRING"
900         required: false
901     - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE:
902         type: "STRING"
903         required: false
904     - KAFKA_CFG_HEAP_OPTS:
905         type: "STRING"
906         required: false
907     - external-port_tcp-internal:
908         type: "STRING"
909         required: false
910     - external-port_tcp-client:
911         type: "STRING"
912         required: false
913     operations: []
914     - cdcservice-type:
915         extends: "-"
916         description: null
917         properties:
918             - container_port:
919                 type: "INTEGER"
920                 required: false
921             - EVENTUATELOCAL_CDC_READER_NAME:
922                 type: "STRING"
923                 required: false
924             - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
925                 type: "STRING"
926                 required: false
927             - SPRING_PROFILES_ACTIVE:
928                 type: "STRING"
```

```
929         required: false
930     - SPRING_DATASOURCE_PASSWORD:
931         type: "STRING"
932         required: false
933     - SPRING_DATASOURCE_TEST_ON_BORROW:
934         type: "STRING"
935         required: false
936     - SPRING_DATASOURCE_VALIDATION_QUERY:
937         type: "STRING"
938         required: false
939     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
940         type: "STRING"
941         required: false
942     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
943         type: "STRING"
944         required: false
945     - SPRING_DATASOURCE_URL:
946         type: "STRING"
947         required: false
948     - EVENTUATE_OUTBOX_ID:
949         type: "STRING"
950         required: false
951     - JAVA_OPTS:
952         type: "STRING"
953         required: false
954     - SPRING_DATASOURCE_USERNAME:
955         type: "STRING"
956         required: false
957     - "'8099'":
958         type: "STRING"
959         required: false
960     operations: []
961 - uibackend-type:
962     extends: "-"
963     description: null
964     properties:
965         - container_port:
966             type: "INTEGER"
967             required: false
968         - T2_ORCHESTRATOR_URL:
969             type: "STRING"
970             required: false
971         - T2_INVENTORY_URL:
972             type: "STRING"
973             required: false
974         - T2_RESERVATION_ENDPOINT:
975             type: "STRING"
976             required: false
977         - JAEGER_HOST:
978             type: "STRING"
979             required: false
980         - T2_CART_URL:
981             type: "STRING"
```

A Expected Technology-Agnostic Deployment Model

```
982         required: false
983     - external_port:
984         type: "STRING"
985         required: false
986     operations: []
987 - cart-type:
988     extends: "-"
989     description: null
990     properties:
991     - container_port:
992         type: "INTEGER"
993         required: false
994     - T2_CART_TTL:
995         type: "STRING"
996         required: false
997     - JAEGER_HOST:
998         type: "STRING"
999         required: false
1000     - MONGO_HOST:
1001         type: "STRING"
1002         required: false
1003     - T2_CART_TASKRATE:
1004         type: "STRING"
1005         required: false
1006     - external_port:
1007         type: "STRING"
1008         required: false
1009     operations: []
1010 - creditinstitute-type:
1011     extends: "-"
1012     description: null
1013     properties:
1014     - container_port:
1015         type: "INTEGER"
1016         required: false
1017     - http:
1018         type: "STRING"
1019         required: false
1020     - external_port:
1021         type: "STRING"
1022         required: false
1023     operations: []
1024 - ui-type:
1025     extends: "-"
1026     description: null
1027     properties:
1028     - container_port:
1029         type: "INTEGER"
1030         required: false
1031     - T2_UIBACKEND_URL:
1032         type: "STRING"
1033         required: false
1034     - external_port:
```

```
1035         type: "STRING"
1036         required: false
1037     operations: []
1038 - payment-type:
1039     extends: "-"
1040     description: null
1041     properties:
1042         - container_port:
1043             type: "INTEGER"
1044             required: false
1045         - T2_PAYMENT_PROVIDER_DUMMY_URL:
1046             type: "STRING"
1047             required: false
1048         - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1049             type: "STRING"
1050             required: false
1051         - SPRING_DATASOURCE_PASSWORD:
1052             type: "STRING"
1053             required: false
1054         - JAEGER_SERVICE_NAME:
1055             type: "STRING"
1056             required: false
1057         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1058             type: "STRING"
1059             required: false
1060         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1061             type: "STRING"
1062             required: false
1063         - SPRING_DATASOURCE_URL:
1064             type: "STRING"
1065             required: false
1066         - JAEGER_ENABLE:
1067             type: "STRING"
1068             required: false
1069         - SPRING_DATASOURCE_USERNAME:
1070             type: "STRING"
1071             required: false
1072         - JAEGER_HOST:
1073             type: "STRING"
1074             required: false
1075         - T2_PAYMENT_PROVIDER_TIMEOUT:
1076             type: "STRING"
1077             required: false
1078         - external_port:
1079             type: "STRING"
1080             required: false
1081     operations: []
1082 - orchestrator-type:
1083     extends: "-"
1084     description: null
1085     properties:
1086         - container_port:
1087             type: "INTEGER"
```

A Expected Technology-Agnostic Deployment Model

```
1088         required: false
1089     - JAEGER_SERVICE_NAME:
1090         type: "STRING"
1091         required: false
1092     - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1093         type: "STRING"
1094         required: false
1095     - SPRING_DATASOURCE_PASSWORD:
1096         type: "STRING"
1097         required: false
1098     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1099         type: "STRING"
1100         required: false
1101     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1102         type: "STRING"
1103         required: false
1104     - SPRING_DATASOURCE_URL:
1105         type: "STRING"
1106         required: false
1107     - JAEGER_ENABLE:
1108         type: "STRING"
1109         required: false
1110     - SPRING_DATASOURCE_USERNAME:
1111         type: "STRING"
1112         required: false
1113     - JAEGER_HOST:
1114         type: "STRING"
1115         required: false
1116     - external_port:
1117         type: "STRING"
1118         required: false
1119     operations: []
1120 - postgres-db:
1121     extends: "-"
1122     description: null
1123     properties:
1124     - container_port:
1125         type: "INTEGER"
1126         required: false
1127     - POSTGRES_DB:
1128         type: "STRING"
1129         required: false
1130     - POSTGRES_PASSWORD:
1131         type: "STRING"
1132         required: false
1133     - POSTGRES_USER:
1134         type: "STRING"
1135         required: false
1136     - USE_DB_ID:
1137         type: "STRING"
1138         required: false
1139     - external_port:
1140         type: "STRING"
```

```

1141         required: false
1142     operations: []
1143 - order-type:
1144     extends: "-"
1145     description: null
1146     properties:
1147         - container_port:
1148             type: "INTEGER"
1149             required: false
1150         - MONGO_HOST:
1151             type: "STRING"
1152             required: false
1153         - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1154             type: "STRING"
1155             required: false
1156         - SPRING_DATASOURCE_PASSWORD:
1157             type: "STRING"
1158             required: false
1159         - JAEGER_SERVICE_NAME:
1160             type: "STRING"
1161             required: false
1162         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1163             type: "STRING"
1164             required: false
1165         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1166             type: "STRING"
1167             required: false
1168         - SPRING_DATASOURCE_URL:
1169             type: "STRING"
1170             required: false
1171         - JAEGER_ENABLE:
1172             type: "STRING"
1173             required: false
1174         - SPRING_DATASOURCE_USERNAME:
1175             type: "STRING"
1176             required: false
1177         - JAEGER_HOST:
1178             type: "STRING"
1179             required: false
1180         - external_port:
1181             type: "STRING"
1182             required: false
1183     operations: []
1184 - inventory-type:
1185     extends: "-"
1186     description: null
1187     properties:
1188         - container_port:
1189             type: "INTEGER"
1190             required: false
1191         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1192             type: "STRING"
1193             required: false

```

A Expected Technology-Agnostic Deployment Model

```
1194     - INVENTORY_SIZE:
1195       type: "STRING"
1196       required: false
1197     - JAEGER_HOST:
1198       type: "STRING"
1199       required: false
1200     - JAEGER_SERVICE_NAME:
1201       type: "STRING"
1202       required: false
1203     - SPRING_PROFILE_ACTIVE:
1204       type: "STRING"
1205       required: false
1206     - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1207       type: "STRING"
1208       required: false
1209     - SPRING_DATASOURCE_PASSWORD:
1210       type: "STRING"
1211       required: false
1212     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1213       type: "STRING"
1214       required: false
1215     - SPRING_DATASOURCE_URL:
1216       type: "STRING"
1217       required: false
1218     - JAEGER_ENABLE:
1219       type: "STRING"
1220       required: false
1221     - T2_INVENTORY_TTL:
1222       type: "STRING"
1223       required: false
1224     - T2_INVENTORY_TASKRATE:
1225       type: "STRING"
1226       required: false
1227     - SPRING_DATASOURCE_USERNAME:
1228       type: "STRING"
1229       required: false
1230     - external_port:
1231       type: "STRING"
1232       required: false
1233     operations: []
1234   relation_types:
1235     - DependsOn:
1236       extends: "-"
1237       description: "generic relation type"
1238       properties: []
1239       operations: []
1240     - HostedOn:
1241       extends: "DependsOn"
1242       description: "hosted on relation"
1243       properties: []
1244       operations: []
1245     - ConnectsTo:
1246       extends: "DependsOn"
```

```
1247     description: "connects to relation"  
1248     properties: []  
1249     operations: []
```

Listing A.1: The expected technology-agnostic deployment model. It is the result that we expect from the transformation framework when it transforms the exemplary technology-specific deployment model of the T2 Project.

B Evaluation Results

In the following we present the results of the evaluation from Chapter 5. Each result refers to a specific metric of the GQM model presented in Table 5.1. Some files of the evaluation result are too big to include here, therefore we provide the complete result in a public repository on Zenodo¹.

B.1 M1: Application Logs Summary

```
1 Registers plugin bash
2 Registers plugin terraform
3 Registers plugin helm
4 Registers plugin kubernetes
5
6 Receives transform command from user
7 Sends request to models service for initialiing internal deployment models
8
9 Sends AnalysisTask to plugin bash (azure-start.sh)
10 Receives EmbeddedDeploymentModelAnalysisRequest: for terraform (terraform/)
11 Receives EmbeddedDeploymentModelAnalysisRequest: for helm (mongo-cart)
12 Receives EmbeddedDeploymentModelAnalysisRequest: for helm (mongo-order)
13 Receives EmbeddedDeploymentModelAnalysisRequest: for helm (kafka)
14 Receives EmbeddedDeploymentModelAnalysisRequest: for kubernetes (k8/)
15 Receives successful AnalysisTaskResponse from plugin bash
16 Unable to find dynamic plugin for bash
17
18 Sends AnalysisTask to plugin terraform (terraform/)
19 Receives successful AnalysisTaskResponse from plugin terraform
20 Unable to find dynamic plugin for terraform
21
22 Sends AnalysisTask to plugin helm (mongo-cart)
23 Receives EmbeddedDeploymentModelAnalysisRequest: for kubernetes (mongo-cart
)
24 Receives successful AnalysisTaskResponse from plugin helm
25 Unable to find dynamic plugin for helm
26
27 Sends AnalysisTask to plugin kubernetes (mongo-cart)
28 Receives successful AnalysisTaskResponse from plugin kubernetes
29 Unable to find dynamic plugin for kubernetes
30
31 Sends AnalysisTask to plugin helm (mongo-order)
```

¹Public repository with complete evaluation results: <https://doi.org/10.5281/zenodo.6503667>

B Evaluation Results

```
32  Receives EmbeddedDeploymentModelAnalysisRequest: for kubernetes (mongo-
    order)
33  Receives successful AnalysisTaskResponse from plugin helm
34  Unable to find dynamic plugin for helm
35
36  Sends AnalysisTask to plugin kubernetes (mongo-order)
37  Receives successful AnalysisTaskResponse from plugin kubernetes
38  Unable to find dynamic plugin for kubernetes
39
40  Sends AnalysisTask to plugin helm (kafka)
41  Receives EmbeddedDeploymentModelAnalysisRequest: for kubernetes (kafka)
42  Receives successful AnalysisTaskResponse from plugin helm
43  Unable to find dynamic plugin for helm
44
45  Sends AnalysisTask to plugin kubernetes (kafka)
46  Receives successful AnalysisTaskResponse from plugin kubernetes
47  Unable to find dynamic plugin for kubernetes
48
49  Sends AnalysisTask to plugin helm (k8/)
50  Receives successful AnalysisTaskResponse from plugin kubernetes
51  Unable to find dynamic plugin for kubernetes
52
53  Requests result from models service
```

Listing B.1: Summary of the events in the application logs from the viewpoint of the analysis manager.

B.2 M2: Registered Plugins

```
1  [
2    {
3      "id": "f142e723-e658-4bc8-929a-5ee9e45b972d",
4      "technology": "bash",
5      "analysisType": "STATIC",
6      "queueName": "bashSTATIC"
7    },
8    {
9      "id": "8df9a6c6-92d1-42b5-a23f-f8e4a12e3de4",
10     "technology": "terraform",
11     "analysisType": "STATIC",
12     "queueName": "terraformSTATIC"
13   },
14   {
15     "id": "04f6c11d-4f25-48a1-a74f-4ab0293b0682",
16     "technology": "helm",
17     "analysisType": "STATIC",
18     "queueName": "helmSTATIC"
19   },
20   {
21     "id": "227924af-8e39-4e9d-b917-146d03a6658a",
22     "technology": "kubernetes",
23     "analysisType": "STATIC",
24     "queueName": "kubernetesSTATIC"
25   }
26 ]
```

Listing B.2: Registered plugins in the configurations database.

B.3 M3: Message Broker Definitions

```
1  {
2    "rabbit_version": "3.9.13",
3    "rabbitmq_version": "3.9.13",
4    "product_name": "RabbitMQ",
5    "product_version": "3.9.13",
6    "users": [
7      {
8        "name": "guest",
9        "password_hash": "6
10       J8azACJ2ifxWax9flllyI0e1XXyLKy6XNeuUWW629dARvvNZ",
11        "hashing_algorithm": "rabbit_password_hashing_sha256",
12        "tags": [
13          "administrator"
14        ],
15        "limits": {}
16      },
17    ],
18    "vhosts": [
19      {
20        "name": "/"
21      },
22    ],
23    "permissions": [
24      {
25        "user": "guest",
26        "vhost": "/",
27        "configure": ".*",
28        "write": ".*",
29        "read": ".*"
30      },
31    ],
32    "topic_permissions": [],
33    "parameters": [],
34    "global_parameters": [
35      {
36        "name": "internal_cluster_id",
37        "value": "rabbitmq-cluster-id-RBpa9WVC4sdq-HJjRnztKQ"
38      },
39    ],
40    "policies": [],
41    "queues": [
42      {
43        "name": "helmSTATIC",
44        "vhost": "/",
45        "durable": true,
46        "auto_delete": false,
47        "arguments": {}
48      },
49      {
50        "name": "bashSTATIC",
```



```
50     "vhost": "/",
51     "durable": true,
52     "auto_delete": false,
53     "arguments": {}
54   },
55   {
56     "name": "AnalysisTaskResponseQueue",
57     "vhost": "/",
58     "durable": true,
59     "auto_delete": false,
60     "arguments": {}
61   },
62   {
63     "name": "kubernetesSTATIC",
64     "vhost": "/",
65     "durable": true,
66     "auto_delete": false,
67     "arguments": {}
68   },
69   {
70     "name": "terraformSTATIC",
71     "vhost": "/",
72     "durable": true,
73     "auto_delete": false,
74     "arguments": {}
75   }
76 ],
77 "exchanges": [
78   {
79     "name": "AnalysisTaskResponseExchange",
80     "vhost": "/",
81     "type": "fanout",
82     "durable": true,
83     "auto_delete": false,
84     "internal": false,
85     "arguments": {}
86   },
87   {
88     "name": "AnalysisTaskRequestExchange",
89     "vhost": "/",
90     "type": "headers",
91     "durable": true,
92     "auto_delete": false,
93     "internal": false,
94     "arguments": {}
95   }
96 ],
97 "bindings": [
98   {
99     "source": "AnalysisTaskRequestExchange",
100    "vhost": "/",
101    "destination": "bashSTATIC",
102    "destination_type": "queue",
```

B Evaluation Results

```
103     "routing_key": "",
104     "arguments": {
105         "analysisType": "STATIC",
106         "technology": "bash",
107         "x-match": "all"
108     }
109 },
110 {
111     "source": "AnalysisTaskRequestExchange",
112     "vhost": "/",
113     "destination": "helmSTATIC",
114     "destination_type": "queue",
115     "routing_key": "",
116     "arguments": {
117         "analysisType": "STATIC",
118         "technology": "helm",
119         "x-match": "all"
120     }
121 },
122 {
123     "source": "AnalysisTaskRequestExchange",
124     "vhost": "/",
125     "destination": "kubernetesSTATIC",
126     "destination_type": "queue",
127     "routing_key": "",
128     "arguments": {
129         "analysisType": "STATIC",
130         "technology": "kubernetes",
131         "x-match": "all"
132     }
133 },
134 {
135     "source": "AnalysisTaskRequestExchange",
136     "vhost": "/",
137     "destination": "terraformSTATIC",
138     "destination_type": "queue",
139     "routing_key": "",
140     "arguments": {
141         "analysisType": "STATIC",
142         "technology": "terraform",
143         "x-match": "all"
144     }
145 },
146 {
147     "source": "AnalysisTaskResponseExchange",
148     "vhost": "/",
149     "destination": "AnalysisTaskResponseQueue",
150     "destination_type": "queue",
151     "routing_key": "",
152     "arguments": {}
153 }
154 ]
```

155 }

Listing B.3: Created AMQP entities on the RabbitMQ message broker.

B.4 M5: Actual Technology-Agnostic Deployment Model

```
1 ---
2 properties: []
3 components:
4   - default:
5     type: "physical_node"
6     description: null
7     properties:
8       - cpu_count: 4
9       - ram_GiB: 16
10      - storage_GiB: 32
11     operations: []
12     artifacts: []
13   - default-operating-system:
14     type: "operating_system"
15     description: null
16     properties:
17       - name: "Ubuntu"
18       - version: "18.04"
19       - os_family: "Linux"
20     operations: []
21     artifacts: []
22   - default-container-runtime:
23     type: "container_runtime"
24     description: null
25     properties:
26       - name: "containerd"
27     operations: []
28     artifacts: []
29   - mongo-cart-mongodb:
30     type: "mongo-cart-mongodb-type"
31     description: null
32     properties:
33       - mongodb: 27017
34       - MONGODB_DISABLE_SYSTEM_LOG: "\"no\""
35       - ALLOW_EMPTY_PASSWORD: "\"yes\""
36       - BITNAMI_DEBUG: "\"false\""
37       - MONGODB_SYSTEM_LOG_VERBOSITY: "\"0\""
38       - MONGODB_ENABLE_IPV6: "\"no\""
39       - MONGODB_DISABLE_JAVASCRIPT: "\"no\""
40       - MONGODB_ENABLE_JOURNAL: "\"yes\""
41       - MONGODB_ENABLE_DIRECTORY_PER_DB: "\"no\""
42       - mongodb: "27017:mongodb"
43     operations: []
44     artifacts:
45       - docker_image:
46         name: "docker.io/bitnami/mongodb:4.4.10-debian-10-r44"
47         fileURI: "-"
48   - mongo-order-mongodb:
49     type: "mongo-order-mongodb-type"
50     description: null
```

```

51   properties:
52     - mongodb: 27017
53     - MONGODB_DISABLE_SYSTEM_LOG: "\"no\""
54     - ALLOW_EMPTY_PASSWORD: "\"yes\""
55     - BITNAMI_DEBUG: "\"false\""
56     - MONGODB_SYSTEM_LOG_VERBOSITY: "\"0\""
57     - MONGODB_ENABLE_IPV6: "\"no\""
58     - MONGODB_DISABLE_JAVASCRIPT: "\"no\""
59     - MONGODB_ENABLE_JOURNAL: "\"yes\""
60     - MONGODB_ENABLE_DIRECTORY_PER_DB: "\"no\""
61     - mongodb: "27017:mongodb"
62   operations: []
63   artifacts:
64     - docker_image:
65       name: "docker.io/bitnami/mongodb:4.4.10-debian-10-r44"
66       fileURI: "-"
67   - kafka-zookeeper:
68     type: "kafka-zookeeper-type"
69     description: null
70     properties:
71       - client: 2181
72       - election: 3888
73       - follower: 2888
74       - ZOO_4LW_COMMANDS_WHITELIST: "\"srvr, mntr, ruok\""
75       - ALLOW_ANONYMOUS_LOGIN: "\"yes\""
76       - ZOO_SYNC_LIMIT: "\"5\""
77       - ZOO_MAX_SESSION_TIMEOUT: "\"40000\""
78       - ZOO_TICK_TIME: "\"2000\""
79       - ZOO_PRE_ALLOC_SIZE: "\"65536\""
80       - ZOO_SERVERS: "kafka-zookeeper-0.kafka-zookeeper-headless.default.
svc.cluster.local:2888:3888:1"
81       - ZOO_AUTOPURGE_RETAIN_COUNT: "\"3\""
82       - ZOO_SNAPCOUNT: "\"100000\""
83       - ZOO_LOG_LEVEL: "\"ERROR\""
84       - ZOO_MAX_CLIENT_CNXNS: "\"60\""
85       - ZOO_INIT_LIMIT: "\"10\""
86       - BITNAMI_DEBUG: "\"false\""
87       - ZOO_ENABLE_AUTH: "\"no\""
88       - ZOO_DATA_LOG_DIR: "\"\""
89       - ZOO_LISTEN_ALL_IPS_ENABLED: "\"no\""
90       - ZOO_PORT_NUMBER: "\"2181\""
91       - ZOO_AUTOPURGE_INTERVAL: "\"0\""
92       - ZOO_HEAP_SIZE: "\"1024\""
93     operations: []
94     artifacts:
95       - docker_image:
96         name: "docker.io/bitnami/zookeeper:3.7.0-debian-10-r188"
97         fileURI: "-"
98   - kafka:
99     type: "kafka-type"
100    description: null
101    properties:
102      - kafka-client: 9092

```

B Evaluation Results

```
103     - kafka-internal: 9093
104     - KAFKA_CFG_SOCKET_RECEIVE_BUFFER_BYTES: "\"102400\""
105     - KAFKA_CFG_SOCKET_SEND_BUFFER_BYTES: "\"102400\""
106     - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP: "\"INTERNAL:PLAINTEXT,
CLIENT:PLAINTEXT\""
107     - KAFKA_CFG_OFFSETS_TOPIC_REPLICATION_FACTOR: "\"1\""
108     - KAFKA_CFG_LOG_FLUSH_INTERVAL_MS: "\"1000\""
109     - KAFKA_CFG_ZOOKEEPER_CONNECTION_TIMEOUT_MS: "\"6000\""
110     - KAFKA_CFG_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: "\"1\""
111     - KAFKA_CFG_LOG_SEGMENT_BYTES: "\"1073741824\""
112     - KAFKA_CFG_DEFAULT_REPLICATION_FACTOR: "\"1\""
113     - KAFKA_CFG_SUPER_USERS: "\"User:admin\""
114     - KAFKA_CFG_ADVERTISED_LISTENERS: "\"INTERNAL://$(MY_POD_NAME).
kafka-headless.default.svc.cluster.local:9093,CLIENT://$(MY_POD_NAME).kafka
-headless.default.svc.cluster.local:9092\""
115     - KAFKA_CFG_LISTENERS: "\"INTERNAL://:9093,CLIENT://:9092\""
116     - KAFKA_CFG_TRANSACTION_STATE_LOG_MIN_ISR: "\"1\""
117     - KAFKA_CFG_LOG_RETENTION_HOURS: "\"168\""
118     - KAFKA_CFG_LOG_FLUSH_INTERVAL_MESSAGES: "\"10000\""
119     - BITNAMI_DEBUG: "\"false\""
120     - KAFKA_CFG_ZOOKEEPER_CONNECT: "\"kafka-zookeeper\""
121     - KAFKA_CFG_NUM_NETWORK_THREADS: "\"3\""
122     - KAFKA_CFG_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "\"true\""
123     - KAFKA_CFG_NUM_RECOVERY_THREADS_PER_DATA_DIR: "\"1\""
124     - KAFKA_CFG_LOG_RETENTION_BYTES: "\"1073741824\""
125     - KAFKA_CFG_SOCKET_REQUEST_MAX_BYTES: "\"104857600\""
126     - KAFKA_INTER_BROKER_LISTENER_NAME: "\"INTERNAL\""
127     - KAFKA_CFG_NUM_IO_THREADS: "\"8\""
128     - ALLOW_PLAINTEXT_LISTENER: "\"yes\""
129     - KAFKA_CFG_LOG_RETENTION_CHECK_INTERVALS_MS: "\"300000\""
130     - KAFKA_LOG_DIR: "\"/opt/bitnami/kafka/logs\""
131     - KAFKA_CFG_NUM_PARTITIONS: "\"1\""
132     - KAFKA_CFG_AUTHORIZER_CLASS_NAME: "\"\""
133     - KAFKA_CFG_DELETE_TOPIC_ENABLE: "\"false\""
134     - KAFKA_CFG_MESSAGE_MAX_BYTES: "\"1000012\""
135     - KAFKA_VOLUME_DIR: "\"/bitnami/kafka\""
136     - KAFKA_CFG_LOG_DIRS: "\"/bitnami/kafka/data\""
137     - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE: "\"true\""
138     - KAFKA_HEAP_OPTS: "\"-Xmx1024m -Xms1024m\""
139     - tcp-internal: "9093:kafka-internal"
140     - tcp-client: "9092:kafka-client"
141     operations: []
142     artifacts:
143     - docker_image:
144         name: "docker.io/bitnami/kafka:2.8.1-debian-10-r57"
145         fileURI: "-"
146     - cdcservice:
147         type: "cdcservice-type"
148         description: null
149         properties:
150         - container_port: 8080
151         - EVENTUATELOCAL_CDC_READER_NAME: "PostgresPollingReader"
152         - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
```

```

153     - SPRING_PROFILES_ACTIVE: "EventuatePolling"
154     - SPRING_DATASOURCE_PASSWORD: "eventuate"
155     - SPRING_DATASOURCE_TEST_ON_BORROW: "\"true\""
156     - SPRING_DATASOURCE_VALIDATION_QUERY: "SELECT 1"
157     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
158     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181"
159
160     - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
161     - EVENTUATE_OUTBOX_ID: "\"1\""
162     - JAVA_OPTS: "-Xmx64m"
163     - SPRING_DATASOURCE_USERNAME: "eventuate"
164     - "'8099)': "8099:8080"
165     operations: []
166     artifacts:
167     - docker_image:
168       name: "eventuateio/eventuate-cdc-service:0.12.0.RELEASE"
169       fileURI: "-"
170   - uibackend:
171     type: "uibackend-type"
172     description: null
173     properties:
174     - container_port: 8080
175     - T2_ORCHESTRATOR_URL: "http://orchestrator-cs/order/"
176     - T2_INVENTORY_URL: "http://inventory-cs/inventory/"
177     - T2_RESERVATION_ENDPOINT: "reservation"
178     - JAEGER_HOST: "localhost #todo"
179     - T2_CART_URL: "http://cart-cs/cart/"
180     - external_port: "80:8080"
181     operations: []
182     artifacts:
183     - docker_image:
184       name: "t2project/uibackend:main"
185       fileURI: "-"
186   - cart:
187     type: "cart-type"
188     description: null
189     properties:
190     - container_port: 8080
191     - T2_CART_TTL: "\"0\""
192     - JAEGER_HOST: "localhost #todo"
193     - MONGO_HOST: "mongo-cart-mongodb"
194     - T2_CART_TASKRATE: "\"0\""
195     - external_port: "80:8080"
196     operations: []
197     artifacts:
198     - docker_image:
199       name: "t2project/cart:main"
200       fileURI: "-"
201   - creditinstitute:
202     type: "creditinstitute-type"
203     description: null
204     properties:

```

B Evaluation Results

```
204     - container_port: 8080
205     - http: "80:8080"
206     - external_port: "80:8080"
207     operations: []
208     artifacts:
209     - docker_image:
210         name: "t2project/creditinstitute:main"
211         fileURI: "-"
212 - ui:
213     type: "ui-type"
214     description: null
215     properties:
216     - container_port: 8080
217     - T2_UIBACKEND_URL: "http://uibackend-cs/"
218     - external_port: "80:8080"
219     operations: []
220     artifacts:
221     - docker_image:
222         name: "t2project/ui:main"
223         fileURI: "-"
224 - payment:
225     type: "payment-type"
226     description: null
227     properties:
228     - container_port: 8080
229     - T2_PAYMENT_PROVIDER_DUMMY_URL: "http://creditinstitute-cs/pay"
230     - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
231     - SPRING_DATASOURCE_PASSWORD: "eventuate"
232     - JAEGER_SERVICE_NAME: "payment"
233     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
234     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181"
235     - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
236     - JAEGER_ENABLE: "\"FALSE\""
237     - SPRING_DATASOURCE_USERNAME: "eventuate"
238     - JAEGER_HOST: "simplest-agent #todo"
239     - T2_PAYMENT_PROVIDER_TIMEOUT: "\"5\""
240     - external_port: "80:8080"
241     operations: []
242     artifacts:
243     - docker_image:
244         name: "t2project/payment:main"
245         fileURI: "-"
246 - orchestrator:
247     type: "orchestrator-type"
248     description: null
249     properties:
250     - container_port: 8080
251     - JAEGER_SERVICE_NAME: "orchestrator"
252     - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
253     - SPRING_DATASOURCE_PASSWORD: "eventuate"
254     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
```



```

255         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181
"
256         - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
257         - JAEGER_ENABLE: "\"FALSE\""
258         - SPRING_DATASOURCE_USERNAME: "eventuate"
259         - JAEGER_HOST: "simplest-agent #todo"
260         - external_port: "80:8080"
261         operations: []
262         artifacts:
263         - docker_image:
264             name: "t2project/orchestrator:main"
265             fileURI: "-"
266     - postgres-orchestrator:
267         type: "postgres-orchestrator-type"
268         description: null
269         properties:
270         - container_port: 5432
271         - POSTGRES_PASSWORD: "eventuate"
272         - POSTGRES_USER: "eventuate"
273         - USE_DB_ID: "\"true\""
274         - external_port: "5432:5432"
275         operations: []
276         artifacts:
277         - docker_image:
278             name: "eventuateio/eventuate-tram-sagas-postgres:0.18.0.RELEASE
"
279             fileURI: "-"
280     - postgres-inventory:
281         type: "postgres-inventory-type"
282         description: null
283         properties:
284         - container_port: 5432
285         - POSTGRES_DB: "inventory"
286         - POSTGRES_PASSWORD: "inventory"
287         - POSTGRES_USER: "inventory"
288         - external_port: "5432:5432"
289         operations: []
290         artifacts:
291         - docker_image:
292             name: "postgres:14.1"
293             fileURI: "-"
294     - order:
295         type: "order-type"
296         description: null
297         properties:
298         - container_port: 8080
299         - MONGO_HOST: "mongo-order-mongodb"
300         - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
301         - SPRING_DATASOURCE_PASSWORD: "eventuate"
302         - JAEGER_SERVICE_NAME: "order"
303         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"

```

B Evaluation Results

```
304         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181
"
305         - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-orchestrator/
eventuate"
306         - JAEGER_ENABLE: "\"FALSE\""
307         - SPRING_DATASOURCE_USERNAME: "eventuate"
308         - JAEGER_HOST: "simplest-agent #todo"
309         - external_port: "80:8080"
310         operations: []
311         artifacts:
312         - docker_image:
313             name: "t2project/order:main"
314             fileURI: "-"
315     - inventory:
316         type: "inventory-type"
317         description: null
318         properties:
319         - container_port: 8080
320         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS: "kafka:9092"
321         - INVENTORY_SIZE: "\"25\""
322         - JAEGER_HOST: "simplest-agent #todo"
323         - JAEGER_SERVICE_NAME: "inventory"
324         - SPRING_PROFILE_ACTIVE: "saga"
325         - SPRING_DATASOURCE_DRIVER_CLASS_NAME: "org.postgresql.Driver"
326         - SPRING_DATASOURCE_PASSWORD: "inventory"
327         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING: "kafka-zookeeper:2181
"
328         - SPRING_DATASOURCE_URL: "jdbc:postgresql://postgres-inventory:5432
/inventory"
329         - JAEGER_ENABLE: "\"FALSE\""
330         - T2_INVENTORY_TTL: "\"0\""
331         - T2_INVENTORY_TASKRATE: "\"0\""
332         - SPRING_DATASOURCE_USERNAME: "inventory"
333         - external_port: "80:8080"
334         operations: []
335         artifacts:
336         - docker_image:
337             name: "t2project/inventory:main"
338             fileURI: "-"
339     relations:
340     - default-operating-system_HostedOn_default:
341         type: "HostedOn"
342         description: null
343         source: "default-operating-system"
344         target: "default"
345         properties: []
346         operations: []
347     - default-container-runtime_HostedOn_default-operating-system:
348         type: "HostedOn"
349         description: null
350         source: "default-container-runtime"
351         target: "default-operating-system"
352         properties: []
```

```
353     operations: []
354 - mongo-cart-mongodb_HostedOn_default-container-runtime:
355   type: "HostedOn"
356   description: null
357   source: "mongo-cart-mongodb"
358   target: "default-container-runtime"
359   properties: []
360   operations: []
361 - mongo-order-mongodb_HostedOn_default-container-runtime:
362   type: "HostedOn"
363   description: null
364   source: "mongo-order-mongodb"
365   target: "default-container-runtime"
366   properties: []
367   operations: []
368 - kafka-zookeeper_HostedOn_default-container-runtime:
369   type: "HostedOn"
370   description: null
371   source: "kafka-zookeeper"
372   target: "default-container-runtime"
373   properties: []
374   operations: []
375 - kafka_HostedOn_default-container-runtime:
376   type: "HostedOn"
377   description: null
378   source: "kafka"
379   target: "default-container-runtime"
380   properties: []
381   operations: []
382 - cdcservice_ConnectsTo_kafka:
383   type: "ConnectsTo"
384   description: null
385   source: "cdcservice"
386   target: "kafka"
387   properties: []
388   operations: []
389 - cdcservice_ConnectsTo_kafka-zookeeper:
390   type: "ConnectsTo"
391   description: null
392   source: "cdcservice"
393   target: "kafka-zookeeper"
394   properties: []
395   operations: []
396 - cdcservice_ConnectsTo_postgres-orchestrator:
397   type: "ConnectsTo"
398   description: null
399   source: "cdcservice"
400   target: "postgres-orchestrator"
401   properties: []
402   operations: []
403 - cdcservice_HostedOn_default-container-runtime:
404   type: "HostedOn"
405   description: null
```

B Evaluation Results

```
406     source: "cdcservice"
407     target: "default-container-runtime"
408     properties: []
409     operations: []
410 - uibackend_ConnectsTo_orchestrator:
411     type: "ConnectsTo"
412     description: null
413     source: "uibackend"
414     target: "orchestrator"
415     properties: []
416     operations: []
417 - uibackend_ConnectsTo_inventory:
418     type: "ConnectsTo"
419     description: null
420     source: "uibackend"
421     target: "inventory"
422     properties: []
423     operations: []
424 - uibackend_ConnectsTo_cart:
425     type: "ConnectsTo"
426     description: null
427     source: "uibackend"
428     target: "cart"
429     properties: []
430     operations: []
431 - uibackend_HostedOn_default-container-runtime:
432     type: "HostedOn"
433     description: null
434     source: "uibackend"
435     target: "default-container-runtime"
436     properties: []
437     operations: []
438 - cart_ConnectsTo_mongo-cart-mongodb:
439     type: "ConnectsTo"
440     description: null
441     source: "cart"
442     target: "mongo-cart-mongodb"
443     properties: []
444     operations: []
445 - cart_HostedOn_default-container-runtime:
446     type: "HostedOn"
447     description: null
448     source: "cart"
449     target: "default-container-runtime"
450     properties: []
451     operations: []
452 - creditinstitute_HostedOn_default-container-runtime:
453     type: "HostedOn"
454     description: null
455     source: "creditinstitute"
456     target: "default-container-runtime"
457     properties: []
458     operations: []
```

```
459 - ui_ConnectsTo_uibackend:
460   type: "ConnectsTo"
461   description: null
462   source: "ui"
463   target: "uibackend"
464   properties: []
465   operations: []
466 - ui_HostedOn_default-container-runtime:
467   type: "HostedOn"
468   description: null
469   source: "ui"
470   target: "default-container-runtime"
471   properties: []
472   operations: []
473 - payment_ConnectsTo_creditinstitute:
474   type: "ConnectsTo"
475   description: null
476   source: "payment"
477   target: "creditinstitute"
478   properties: []
479   operations: []
480 - payment_ConnectsTo_kafka:
481   type: "ConnectsTo"
482   description: null
483   source: "payment"
484   target: "kafka"
485   properties: []
486   operations: []
487 - payment_ConnectsTo_kafka-zookeeper:
488   type: "ConnectsTo"
489   description: null
490   source: "payment"
491   target: "kafka-zookeeper"
492   properties: []
493   operations: []
494 - payment_ConnectsTo_postgres-orchestrator:
495   type: "ConnectsTo"
496   description: null
497   source: "payment"
498   target: "postgres-orchestrator"
499   properties: []
500   operations: []
501 - payment_HostedOn_default-container-runtime:
502   type: "HostedOn"
503   description: null
504   source: "payment"
505   target: "default-container-runtime"
506   properties: []
507   operations: []
508 - orchestrator_ConnectsTo_kafka:
509   type: "ConnectsTo"
510   description: null
511   source: "orchestrator"
```

B Evaluation Results

```
512     target: "kafka"
513     properties: []
514     operations: []
515 - orchestrator_ConnectsTo_kafka-zookeeper:
516     type: "ConnectsTo"
517     description: null
518     source: "orchestrator"
519     target: "kafka-zookeeper"
520     properties: []
521     operations: []
522 - orchestrator_ConnectsTo_postgres-orchestrator:
523     type: "ConnectsTo"
524     description: null
525     source: "orchestrator"
526     target: "postgres-orchestrator"
527     properties: []
528     operations: []
529 - orchestrator_HostedOn_default-container-runtime:
530     type: "HostedOn"
531     description: null
532     source: "orchestrator"
533     target: "default-container-runtime"
534     properties: []
535     operations: []
536 - postgres-orchestrator_HostedOn_default-container-runtime:
537     type: "HostedOn"
538     description: null
539     source: "postgres-orchestrator"
540     target: "default-container-runtime"
541     properties: []
542     operations: []
543 - postgres-inventory_HostedOn_default-container-runtime:
544     type: "HostedOn"
545     description: null
546     source: "postgres-inventory"
547     target: "default-container-runtime"
548     properties: []
549     operations: []
550 - order_ConnectsTo_mongo-order-mongodb:
551     type: "ConnectsTo"
552     description: null
553     source: "order"
554     target: "mongo-order-mongodb"
555     properties: []
556     operations: []
557 - order_ConnectsTo_kafka:
558     type: "ConnectsTo"
559     description: null
560     source: "order"
561     target: "kafka"
562     properties: []
563     operations: []
564 - order_ConnectsTo_kafka-zookeeper:
```

```

565     type: "ConnectsTo"
566     description: null
567     source: "order"
568     target: "kafka-zookeeper"
569     properties: []
570     operations: []
571 - order_ConnectsTo_postgres-orchestrator:
572     type: "ConnectsTo"
573     description: null
574     source: "order"
575     target: "postgres-orchestrator"
576     properties: []
577     operations: []
578 - order_HostedOn_default-container-runtime:
579     type: "HostedOn"
580     description: null
581     source: "order"
582     target: "default-container-runtime"
583     properties: []
584     operations: []
585 - inventory_ConnectsTo_kafka:
586     type: "ConnectsTo"
587     description: null
588     source: "inventory"
589     target: "kafka"
590     properties: []
591     operations: []
592 - inventory_ConnectsTo_kafka-zookeeper:
593     type: "ConnectsTo"
594     description: null
595     source: "inventory"
596     target: "kafka-zookeeper"
597     properties: []
598     operations: []
599 - inventory_ConnectsTo_postgres-inventory:
600     type: "ConnectsTo"
601     description: null
602     source: "inventory"
603     target: "postgres-inventory"
604     properties: []
605     operations: []
606 - inventory_HostedOn_default-container-runtime:
607     type: "HostedOn"
608     description: null
609     source: "inventory"
610     target: "default-container-runtime"
611     properties: []
612     operations: []
613 component_types:
614 - BaseType:
615     extends: "-"
616     description: "This is the base type"
617     properties: []

```

B Evaluation Results

```
618     operations: []
619 - physical_node:
620     extends: "-"
621     description: null
622     properties:
623       - cpu_count:
624         type: "INTEGER"
625         required: false
626       - ram_GiB:
627         type: "INTEGER"
628         required: false
629       - storage_GiB:
630         type: "INTEGER"
631         required: false
632     operations: []
633 - operating_system:
634     extends: "-"
635     description: null
636     properties:
637       - name:
638         type: "STRING"
639         required: false
640       - version:
641         type: "STRING"
642         required: false
643       - os_family:
644         type: "STRING"
645         required: false
646     operations: []
647 - container_runtime:
648     extends: "-"
649     description: null
650     properties:
651       - name:
652         type: "STRING"
653         required: false
654       - version:
655         type: "STRING"
656         required: false
657     operations: []
658 - mongo-cart-mongodb-type:
659     extends: "-"
660     description: null
661     properties:
662       - mongodb:
663         type: "INTEGER"
664         required: false
665       - MONGODB_DISABLE_SYSTEM_LOG:
666         type: "STRING"
667         required: false
668       - ALLOW_EMPTY_PASSWORD:
669         type: "STRING"
670         required: false
```



```
671     - BITNAMI_DEBUG:
672       type: "STRING"
673       required: false
674     - MONGODB_SYSTEM_LOG_VERBOSITY:
675       type: "STRING"
676       required: false
677     - MONGODB_ENABLE_IPV6:
678       type: "STRING"
679       required: false
680     - MONGODB_DISABLE_JAVASCRIPT:
681       type: "STRING"
682       required: false
683     - MONGODB_ENABLE_JOURNAL:
684       type: "STRING"
685       required: false
686     - MONGODB_ENABLE_DIRECTORY_PER_DB:
687       type: "STRING"
688       required: false
689     - mongodb:
690       type: "STRING"
691       required: false
692     operations: []
693   - mongo-order-mongodb-type:
694     extends: "-"
695     description: null
696     properties:
697       - mongodb:
698         type: "INTEGER"
699         required: false
700       - MONGODB_DISABLE_SYSTEM_LOG:
701         type: "STRING"
702         required: false
703       - ALLOW_EMPTY_PASSWORD:
704         type: "STRING"
705         required: false
706       - BITNAMI_DEBUG:
707         type: "STRING"
708         required: false
709       - MONGODB_SYSTEM_LOG_VERBOSITY:
710         type: "STRING"
711         required: false
712       - MONGODB_ENABLE_IPV6:
713         type: "STRING"
714         required: false
715       - MONGODB_DISABLE_JAVASCRIPT:
716         type: "STRING"
717         required: false
718       - MONGODB_ENABLE_JOURNAL:
719         type: "STRING"
720         required: false
721       - MONGODB_ENABLE_DIRECTORY_PER_DB:
722         type: "STRING"
723         required: false
```

B Evaluation Results

```
724     - mongodb:
725         type: "STRING"
726         required: false
727     operations: []
728 - kafka-zookeeper-type:
729     extends: "-"
730     description: null
731     properties:
732     - client:
733         type: "INTEGER"
734         required: false
735     - election:
736         type: "INTEGER"
737         required: false
738     - follower:
739         type: "INTEGER"
740         required: false
741     - ZOO_4LW_COMMANDS_WHITELIST:
742         type: "STRING"
743         required: false
744     - ALLOW_ANONYMOUS_LOGIN:
745         type: "STRING"
746         required: false
747     - ZOO_SYNC_LIMIT:
748         type: "STRING"
749         required: false
750     - ZOO_MAX_SESSION_TIMEOUT:
751         type: "STRING"
752         required: false
753     - ZOO_TICK_TIME:
754         type: "STRING"
755         required: false
756     - ZOO_PRE_ALLOC_SIZE:
757         type: "STRING"
758         required: false
759     - ZOO_SERVERS:
760         type: "STRING"
761         required: false
762     - ZOO_AUTOPURGE_RETAIN_COUNT:
763         type: "STRING"
764         required: false
765     - ZOO_SNAPCOUNT:
766         type: "STRING"
767         required: false
768     - ZOO_LOG_LEVEL:
769         type: "STRING"
770         required: false
771     - ZOO_MAX_CLIENT_CNXNS:
772         type: "STRING"
773         required: false
774     - ZOO_INIT_LIMIT:
775         type: "STRING"
776         required: false
```

```
777     - BITNAMI_DEBUG:
778       type: "STRING"
779       required: false
780     - ZOO_ENABLE_AUTH:
781       type: "STRING"
782       required: false
783     - ZOO_DATA_LOG_DIR:
784       type: "STRING"
785       required: false
786     - ZOO_LISTEN_ALLIPES_ENABLED:
787       type: "STRING"
788       required: false
789     - ZOO_PORT_NUMBER:
790       type: "STRING"
791       required: false
792     - ZOO_AUTOPURGE_INTERVAL:
793       type: "STRING"
794       required: false
795     - ZOO_HEAP_SIZE:
796       type: "STRING"
797       required: false
798     operations: []
799   - kafka-type:
800     extends: "-"
801     description: null
802     properties:
803       - kafka-client:
804         type: "INTEGER"
805         required: false
806       - kafka-internal:
807         type: "INTEGER"
808         required: false
809       - KAFKA_CFG_SOCKET_RECEIVE_BUFFER_BYTES:
810         type: "STRING"
811         required: false
812       - KAFKA_CFG_SOCKET_SEND_BUFFER_BYTES:
813         type: "STRING"
814         required: false
815       - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP:
816         type: "STRING"
817         required: false
818       - KAFKA_CFG_OFFSETS_TOPIC_REPLICATION_FACTOR:
819         type: "STRING"
820         required: false
821       - KAFKA_CFG_LOG_FLUSH_INTERVAL_MS:
822         type: "STRING"
823         required: false
824       - KAFKA_CFG_ZOOKEEPER_CONNECTION_TIMEOUT_MS:
825         type: "STRING"
826         required: false
827       - KAFKA_CFG_TRANSACTION_STATE_LOG_REPLICATION_FACTOR:
828         type: "STRING"
829         required: false
```

B Evaluation Results

```
830     - KAFKA_CFG_LOG_SEGMENT_BYTES:
831       type: "STRING"
832       required: false
833     - KAFKA_CFG_DEFAULT_REPLICATION_FACTOR:
834       type: "STRING"
835       required: false
836     - KAFKA_CFG_SUPER_USERS:
837       type: "STRING"
838       required: false
839     - KAFKA_CFG_ADVERTISED_LISTENERS:
840       type: "STRING"
841       required: false
842     - KAFKA_CFG_LISTENERS:
843       type: "STRING"
844       required: false
845     - KAFKA_CFG_TRANSACTION_STATE_LOG_MIN_ISR:
846       type: "STRING"
847       required: false
848     - KAFKA_CFG_LOG_RETENTION_HOURS:
849       type: "STRING"
850       required: false
851     - KAFKA_CFG_LOG_FLUSH_INTERVAL_MESSAGES:
852       type: "STRING"
853       required: false
854     - BITNAMI_DEBUG:
855       type: "STRING"
856       required: false
857     - KAFKA_CFG_ZOOKEEPER_CONNECT:
858       type: "STRING"
859       required: false
860     - KAFKA_CFG_NUM_NETWORK_THREADS:
861       type: "STRING"
862       required: false
863     - KAFKA_CFG_ALLOW_EVERYONE_IF_NO_ACL_FOUND:
864       type: "STRING"
865       required: false
866     - KAFKA_CFG_NUM_RECOVERY_THREADS_PER_DATA_DIR:
867       type: "STRING"
868       required: false
869     - KAFKA_CFG_LOG_RETENTION_BYTES:
870       type: "STRING"
871       required: false
872     - KAFKA_CFG_SOCKET_REQUEST_MAX_BYTES:
873       type: "STRING"
874       required: false
875     - KAFKA_INTER_BROKER_LISTENER_NAME:
876       type: "STRING"
877       required: false
878     - KAFKA_CFG_NUM_IO_THREADS:
879       type: "STRING"
880       required: false
881     - ALLOW_PLAINTEXT_LISTENER:
882       type: "STRING"
```

```

883         required: false
884     - KAFKA_CFG_LOG_RETENTION_CHECK_INTERVALS_MS:
885         type: "STRING"
886         required: false
887     - KAFKA_CFG_LOG_DIR:
888         type: "STRING"
889         required: false
890     - KAFKA_CFG_NUM_PARTITIONS:
891         type: "STRING"
892         required: false
893     - KAFKA_CFG_AUTHORIZER_CLASS_NAME:
894         type: "STRING"
895         required: false
896     - KAFKA_CFG_DELETE_TOPIC_ENABLE:
897         type: "STRING"
898         required: false
899     - KAFKA_CFG_MESSAGE_MAX_BYTES:
900         type: "STRING"
901         required: false
902     - KAFKA_CFG_VOLUME_DIR:
903         type: "STRING"
904         required: false
905     - KAFKA_CFG_LOG_DIRS:
906         type: "STRING"
907         required: false
908     - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE:
909         type: "STRING"
910         required: false
911     - KAFKA_CFG_HEAP_OPTS:
912         type: "STRING"
913         required: false
914     - tcp-internal:
915         type: "STRING"
916         required: false
917     - tcp-client:
918         type: "STRING"
919         required: false
920     operations: []
921 - cdc-service-type:
922     extends: "-"
923     description: null
924     properties:
925         - container_port:
926             type: "INTEGER"
927             required: false
928         - EVENTUATELOCAL_CDC_READER_NAME:
929             type: "STRING"
930             required: false
931         - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
932             type: "STRING"
933             required: false
934         - SPRING_PROFILES_ACTIVE:
935             type: "STRING"

```

B Evaluation Results

```
936         required: false
937     - SPRING_DATASOURCE_PASSWORD:
938         type: "STRING"
939         required: false
940     - SPRING_DATASOURCE_TEST_ON_BORROW:
941         type: "STRING"
942         required: false
943     - SPRING_DATASOURCE_VALIDATION_QUERY:
944         type: "STRING"
945         required: false
946     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
947         type: "STRING"
948         required: false
949     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
950         type: "STRING"
951         required: false
952     - SPRING_DATASOURCE_URL:
953         type: "STRING"
954         required: false
955     - EVENTUATE_OUTBOX_ID:
956         type: "STRING"
957         required: false
958     - JAVA_OPTS:
959         type: "STRING"
960         required: false
961     - SPRING_DATASOURCE_USERNAME:
962         type: "STRING"
963         required: false
964     - "8099":
965         type: "STRING"
966         required: false
967     operations: []
968 - uibackend-type:
969     extends: "-"
970     description: null
971     properties:
972     - container_port:
973         type: "INTEGER"
974         required: false
975     - T2_ORCHESTRATOR_URL:
976         type: "STRING"
977         required: false
978     - T2_INVENTORY_URL:
979         type: "STRING"
980         required: false
981     - T2_RESERVATION_ENDPOINT:
982         type: "STRING"
983         required: false
984     - JAEGER_HOST:
985         type: "STRING"
986         required: false
987     - T2_CART_URL:
988         type: "STRING"
```

```
989         required: false
990     - external_port:
991         type: "STRING"
992         required: false
993     operations: []
994 - cart-type:
995     extends: "-"
996     description: null
997     properties:
998     - container_port:
999         type: "INTEGER"
1000         required: false
1001     - T2_CART_TTL:
1002         type: "STRING"
1003         required: false
1004     - JAEGER_HOST:
1005         type: "STRING"
1006         required: false
1007     - MONGO_HOST:
1008         type: "STRING"
1009         required: false
1010     - T2_CART_TASKRATE:
1011         type: "STRING"
1012         required: false
1013     - external_port:
1014         type: "STRING"
1015         required: false
1016     operations: []
1017 - creditinstitute-type:
1018     extends: "-"
1019     description: null
1020     properties:
1021     - container_port:
1022         type: "INTEGER"
1023         required: false
1024     - http:
1025         type: "STRING"
1026         required: false
1027     - external_port:
1028         type: "STRING"
1029         required: false
1030     operations: []
1031 - ui-type:
1032     extends: "-"
1033     description: null
1034     properties:
1035     - container_port:
1036         type: "INTEGER"
1037         required: false
1038     - T2_UIBACKEND_URL:
1039         type: "STRING"
1040         required: false
1041     - external_port:
```

B Evaluation Results

```
1042         type: "STRING"
1043         required: false
1044     operations: []
1045 - payment-type:
1046     extends: "-"
1047     description: null
1048     properties:
1049         - container_port:
1050             type: "INTEGER"
1051             required: false
1052         - T2_PAYMENT_PROVIDER_DUMMY_URL:
1053             type: "STRING"
1054             required: false
1055         - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1056             type: "STRING"
1057             required: false
1058         - SPRING_DATASOURCE_PASSWORD:
1059             type: "STRING"
1060             required: false
1061         - JAEGER_SERVICE_NAME:
1062             type: "STRING"
1063             required: false
1064         - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1065             type: "STRING"
1066             required: false
1067         - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1068             type: "STRING"
1069             required: false
1070         - SPRING_DATASOURCE_URL:
1071             type: "STRING"
1072             required: false
1073         - JAEGER_ENABLE:
1074             type: "STRING"
1075             required: false
1076         - SPRING_DATASOURCE_USERNAME:
1077             type: "STRING"
1078             required: false
1079         - JAEGER_HOST:
1080             type: "STRING"
1081             required: false
1082         - T2_PAYMENT_PROVIDER_TIMEOUT:
1083             type: "STRING"
1084             required: false
1085         - external_port:
1086             type: "STRING"
1087             required: false
1088     operations: []
1089 - orchestrator-type:
1090     extends: "-"
1091     description: null
1092     properties:
1093         - container_port:
1094             type: "INTEGER"
```



```

1095         required: false
1096     - JAEGER_SERVICE_NAME:
1097         type: "STRING"
1098         required: false
1099     - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1100         type: "STRING"
1101         required: false
1102     - SPRING_DATASOURCE_PASSWORD:
1103         type: "STRING"
1104         required: false
1105     - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1106         type: "STRING"
1107         required: false
1108     - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1109         type: "STRING"
1110         required: false
1111     - SPRING_DATASOURCE_URL:
1112         type: "STRING"
1113         required: false
1114     - JAEGER_ENABLE:
1115         type: "STRING"
1116         required: false
1117     - SPRING_DATASOURCE_USERNAME:
1118         type: "STRING"
1119         required: false
1120     - JAEGER_HOST:
1121         type: "STRING"
1122         required: false
1123     - external_port:
1124         type: "STRING"
1125         required: false
1126     operations: []
1127 - postgres-orchestrator-type:
1128     extends: "-"
1129     description: null
1130     properties:
1131     - container_port:
1132         type: "INTEGER"
1133         required: false
1134     - POSTGRES_PASSWORD:
1135         type: "STRING"
1136         required: false
1137     - POSTGRES_USER:
1138         type: "STRING"
1139         required: false
1140     - USE_DB_ID:
1141         type: "STRING"
1142         required: false
1143     - external_port:
1144         type: "STRING"
1145         required: false
1146     operations: []
1147 - postgres-inventory-type:

```

B Evaluation Results

```
1148     extends: "-"
1149     description: null
1150     properties:
1151       - container_port:
1152         type: "INTEGER"
1153         required: false
1154       - POSTGRES_DB:
1155         type: "STRING"
1156         required: false
1157       - POSTGRES_PASSWORD:
1158         type: "STRING"
1159         required: false
1160       - POSTGRES_USER:
1161         type: "STRING"
1162         required: false
1163       - external_port:
1164         type: "STRING"
1165         required: false
1166     operations: []
1167   - order-type:
1168     extends: "-"
1169     description: null
1170     properties:
1171       - container_port:
1172         type: "INTEGER"
1173         required: false
1174       - MONGO_HOST:
1175         type: "STRING"
1176         required: false
1177       - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1178         type: "STRING"
1179         required: false
1180       - SPRING_DATASOURCE_PASSWORD:
1181         type: "STRING"
1182         required: false
1183       - JAEGER_SERVICE_NAME:
1184         type: "STRING"
1185         required: false
1186       - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1187         type: "STRING"
1188         required: false
1189       - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1190         type: "STRING"
1191         required: false
1192       - SPRING_DATASOURCE_URL:
1193         type: "STRING"
1194         required: false
1195       - JAEGER_ENABLE:
1196         type: "STRING"
1197         required: false
1198       - SPRING_DATASOURCE_USERNAME:
1199         type: "STRING"
1200         required: false
```

```
1201     - JAEGER_HOST:
1202       type: "STRING"
1203       required: false
1204     - external_port:
1205       type: "STRING"
1206       required: false
1207     operations: []
1208   - inventory-type:
1209     extends: "-"
1210     description: null
1211     properties:
1212       - container_port:
1213         type: "INTEGER"
1214         required: false
1215       - EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS:
1216         type: "STRING"
1217         required: false
1218       - INVENTORY_SIZE:
1219         type: "STRING"
1220         required: false
1221       - JAEGER_HOST:
1222         type: "STRING"
1223         required: false
1224       - JAEGER_SERVICE_NAME:
1225         type: "STRING"
1226         required: false
1227       - SPRING_PROFILE_ACTIVE:
1228         type: "STRING"
1229         required: false
1230       - SPRING_DATASOURCE_DRIVER_CLASS_NAME:
1231         type: "STRING"
1232         required: false
1233       - SPRING_DATASOURCE_PASSWORD:
1234         type: "STRING"
1235         required: false
1236       - EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING:
1237         type: "STRING"
1238         required: false
1239       - SPRING_DATASOURCE_URL:
1240         type: "STRING"
1241         required: false
1242       - JAEGER_ENABLE:
1243         type: "STRING"
1244         required: false
1245       - T2_INVENTORY_TTL:
1246         type: "STRING"
1247         required: false
1248       - T2_INVENTORY_TASKRATE:
1249         type: "STRING"
1250         required: false
1251       - SPRING_DATASOURCE_USERNAME:
1252         type: "STRING"
1253         required: false
```

```
1254     - external_port:
1255         type: "STRING"
1256         required: false
1257     operations: []
1258 relation_types:
1259   - DependsOn:
1260       extends: "-"
1261       description: "generic relation type"
1262       properties: []
1263       operations: []
1264   - HostedOn:
1265       extends: "DependsOn"
1266       description: "hosted on relation"
1267       properties: []
1268       operations: []
1269   - ConnectsTo:
1270       extends: "DependsOn"
1271       description: "connects to relation"
1272       properties: []
1273       operations: []
```

Listing B.4: The actual technology-agnostic deployment model that the transformation framework created from the transformation of the exemplary technology-specific deployment model.

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.

Winnenden, 29.04.2022, M. Welzel

place, date, signature