Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

# Concepts for Handling Heterogeneous Data Transformation Logic and their Integration with TraDE Middleware

Vladimir Yussupov

| | |
|---|---|
| **Course of Study:** | Computer Science |
| **Examiner:** | Prof. Dr. Dr. h.c. Frank Leymann |
| **Supervisor:** | Dipl.-Inf. Michael Hahn |
| **Commenced:** | 2017-07-18 |
| **Completed:** | 2017-12-18 |
| **CR-Classification:** | C.2.4, D.2.11, H.4.1, I.7.2 |

## Abstract

The concept of programming-in-the-Large became a substantial part of modern computer-based scientific research with an advent of web services and the concept of orchestration languages. While the notions of workflows and service choreographies help to reduce the complexity by providing means to support the communication between involved participants, the process still remains generally complex. The TraDE Middleware and underlying concepts were introduced in order to provide means for performing the modeled data exchange across choreography participants in a transparent and automated fashion. However, in order to achieve both transparency and automation, the TraDE Middleware must be capable of transforming the data along its path. The data transformation's transparency can be difficult to achieve due to various factors including the diversity of required execution environments and complicated configuration processes as well as the heterogeneity of data transformation software which results in tedious integration processes often involving the manual wrapping of software.

Having a method of handling data transformation applications in a standardized manner can help to simplify the process of modeling and executing scientific service choreographies with the TraDE concepts applied. In this master thesis we analyze various aspects of this problem and conceptualize an extensible framework for handling the data transformation applications. The resulting prototypical implementation of the presented framework provides means to address data transformation applications in a standardized manner.

## Acknowledgements

I want to thank my supervisor Michael Hahn at University of Stuttgart for constructive feedback, patience and continuous guidance during the progress of my master's thesis.

I would also like to thank Professor Frank Leymann for giving me the opportunity to do my master's thesis at the Institute of Architecture of Application Systems.

My heartfelt appreciation goes to my beloved wife Anna whose love and support made me more confident in chasing my dreams and made this thesis possible. I am immensely grateful to my parents Nail and Vera, and my sister Jane for their love, tremendous support and encouragement throughout my whole life. I am also very grateful to my other family members who have supported and encouraged me along the way.

# Contents

# List of Abbreviations

**ADL** Architecture Description Language. 32

**API** Application Programming Interface. 22

**BPEL** Business Process Execution Language. 11

**BPMN** Business Process Model and Notation. 14

**CBSE** Component-Based Software Engineering. 25

**CLI** Command Line Interface. 32

**CORBA** Common Object Request Broker Architecture. 25

**COTS** Commercial Off-The-Shelf. 23

**CSV** Comma-Separated Values. 56

**DSL** Domain-Specific Language. 23

**EJB** Enterprise JavaBeans. 25

**ESB** Enterprise Service Bus. 78

**FSA** Finite State Automaton. 35

**GUI** Graphical User Interface. 28

**HATEOAS** Hypertext As The Engine Of Application State. 92

**HTML** Hypertext Markup Language. 34

**HTTP** Hypertext Transfer Protocol. 34

**IaaS** Infrastructure as a Service. 42

**IDE** Integrated Development Environment. 31

**Java RMI** Java Remote Method Invocation. 86

**JSON** JavaScript Object Notation. 38

**KDD** Knowledge Discovery in Databases. 35

**KMC** Kinetic Monte Carlo. 16

**LIS** Legacy Information Systems. 28

**MIME** Multipurpose Internet Mail Extensions. 56

**OPAL** Ostwald ripening of Precipitates on an Atomic Lattice. 16

**OS** Operating System. 35

**PaaS** Platform as a Service. 38

**RBAC** Role Based Access Control. 79

**REST** Representational State Transfer. 22

**SaaS** Software as a Service. 38

**SOA** Service Oriented Architecture. 37

**SSH** Secure Shell. 48

**UI** User Interface. 22

**UML** Unified Modeling Language. 28

**URI** Uniform Resource Identifier. 36

**URL** Uniform Resource Locator. 34

**UTS** UNIX Timesharing System. 41

**VM** Virtual Machine. 40

**WFMS** Workflow Management System. 14

**WS-CDL** Web Service Choreography Description Language. 14

**WSDL** Web Service Description Language. 34

**XML** Extensible Markup Language. 28

**YAML** YAML Ain't Markup Language. 89

# 1 Introduction

The idea to distinguish between the development of atomic software modules and composing systems using such modules as building blocks is not novel and dates back at least to the middle of 70s when DeRemer et al. [DK76] coined the terms *programming-in-the-Small* and *programming-in-the-Large*. The former describes a process of developing individual software modules using a traditional set of programming languages. The latter refers to the process of composing a system using a large set of modules. DeRemer et al. describe these concepts as distinct types of intellectual activity requiring different programming languages. With an advent of web services and orchestration languages like Business Process Execution Language (BPEL) the concept of programming-in-the-Large became a substantial part of modern computer-based scientific research. Splitting a complex task into a set of dedicated tasks involving distinct software modules and orchestrating them makes overall process more controllable and understandable. Moreover, multiple sub-tasks are conducted by completely separate groups of people and the concept of orchestration can even be seen from the higher levels when parts of the experiment orchestrated by distinct scientists need to be combined as a whole. This process resembles a choreography class where an elegantly-executed dance is the result of a hard work required from any involved participant. The concept of service choreographies describes how multiple participants can communicate in a specified way in order to achieve a common goal. Applying this concept to the scientific setting, participants become parts of the computer-based experiment and the choreography becomes the model of how these parts need to interact.

However, the technical aspects of running the computer-based experiments “in-the-Large” add more complexity to the process. The modeling and orchestration of such experiments becomes a task requiring knowledge of specific tools and technologies. However, the involved scientists do not always have expertise in the field of service technology or orchestration languages. Moreover, the software which is used for the experiments is not always suitable for orchestrations and choreographies. The process of preparing the computer-based experiment for “in-the-Large” setting becomes a separate task and might become a serious difficulty on the way to running the experiment in such way. This thesis focuses on how a particular type of applications can be handled in the context of a choreographed interaction in the domain of e-Science.

In this chapter we focus on the high-level concepts underlying the topic of research. We discuss what is the motivation behind the topic and describe the problems needed to be solved. Based on a high-level view we define the main goal of this thesis and highlight the primary research questions. Finally, the general structure of this work is presented.

## 1.1 Informal Thoughts on Topic's Semantics

The wrong interpretation of the thesis' title might result in the improper direction of research. Prior to formulating the goal and the main research questions it can be helpful to start with an informal analysis of this work's title. The concept of text segmentation in the field of natural language processing is a helpful way to identify the important parts of the title. Luckily, the human brain is able to perform a sophisticated text segmentation on-the-fly. We are interested in identifying and analyzing the important text segments in the following topic: "*Concepts for Handling Heterogeneous Data Transformation Logic and their Integration with TraDE Middleware*". The most important part to highlight is *Data Transformation Logic*. In the context of computer science, this segment can be used to describe any form of software responsible for performing a task of *data transformation*, e.g. a source code, an executable binary, or a web service. The definition of data transformation task depends on the context in which it is applied. For instance, the data transformations using various functions in statistics differ from conversion between different data formats or structures in information integration. A more precise explanation of a data transformation task in the context of this thesis is needed.

The Data Transformation Logic text segment can serve as a center point and we can start linking it to the other words or text segments of the topic and analyzing their relations. This rather informal approach might help to highlight the potential problems which require additional attention.

**Heterogeneous Data Transformation Logic** The word *heterogeneous* adds an emphasis on diversity of software entities performing a data transformation task. Examples of such heterogeneity include a variety of factors such as the usage of different programming languages, various structures or formats of input and output data, or diverse invocation mechanisms.

**(Concepts for) Handling Data Transformation Logic** The segment *Concepts for* puts an emphasis on the plurality of ideas related to handling the data transformation logic. The term *handling* can be considered from the perspective of software reuse. An interesting question to answer is how multiple standalone data transformation applications can be handled in a standardized fashion. This problem, depending on reuse methodology, can refer to various research topics including *software re-engineering* and *legacy code wrapping*. Moreover, the word *handling* still sounds ambiguous at this point. Being synonymous to the word *management*, it has a broad spectrum of interpretations in computer science context. Assuming heterogeneity of data transformation software, the selection of storage and search mechanisms is one of the key questions to answer. Another important issue is the remote execution of software that might use diverse invocation mechanisms or have different and potentially contradicting dependencies. Numerous additional questions, for instance, monitoring of the execution or composition of data transformations might be addressed as well. Every stated interpretation leads to a different area of research, hence more precise definition for the spectrum of targeted problems is needed.

**Integration with TraDE Middleware** This text segment is connected to the concepts for handling the heterogeneous data transformation logic. The word *Integration* refers to a process of coordination and synthesis of multiple diverse program entities in order to achieve some global goal which requires these programs to communicate. Enterprise Application Integration is one of the research fields which deals with such topics. TraDE Middleware is the name of a target software with which the produced concepts need to be integrated. The TraDE Middleware has a broad spectrum of potential application domains. It operates with terms like *workflow, service choreographies, simulation*. The main idea is to support modeling and execution of multiple participants' workflows. One domain of application where TraDE Middleware might be particularly useful is the *e-Science* domain.

Combining everything together we can highlight the following important points as a summary for this informal analysis:

- we need to produce concepts for handling heterogeneous applications,
- every such application is only responsible for data transformation tasks,
- the meaning of data transformation task has to be defined more clearly in the context of this work,
- the term *handling an application* might be interpreted in various ways and we need to define its boundaries more precisely in the context of this work,
- derived concepts have to be integrated with the TraDE Middleware, an existing research prototype which can be used in multiple fields including e-Science.

The task of integration with the TraDE Middleware is a significant part of the research. Before proceeding to the problem statement and the description of the main research questions, we need to provide more details about the related concepts along with a brief motivational example.

## 1.2 TraDE: Background and Motivation

### 1.2.1 e-Science

Traditionally, scientific research operated with terms *in vivo, in vitro* or *in situ* in order to describe the way how experiments were performed. With ubiquitous usage of computer-based simulations researchers started to use the term *in silico* to proper describe this type of experiments [Joh01]. Drastic increase in amounts of data [HT03; NEO03] and growing computational complexity of experiments emerged interest in search for effective collaboration methods for scientists. Being coined in 1999 [Jan07], the term *e-Science* was meant to describe a large funding initiative in UK. Since then, the definition of this term remains flexible, although the general idea is about innovative ways of doing science. For instance, IEEE international e-Science conference's website provides two (short and

long) definitions of e-Science [eSc17] which both focus on the notion of collaborative and data- or computationally-intensive research on the global scale. With the advent of distributed computing paradigms like *grid* and *cloud* computing, performing experiments on distributed systems over data spread across multiple locations became possible in a more transparent and convenient way.

A typical chain of steps in scientific experiment is to formulate a hypothesis, design and run the experiment, obtain the results and evaluate their importance [SGG07]. Repetitive nature of experiments often makes the chain of steps to be cyclic. Typical example would be transferring the data to supercomputers for analysis or simulations, running the experiments, storing and evaluating the output, modifying the parameters and repeating the cycle again [Dee+09]. The application of traditional methodologies to the world of distributed computing required proper coordination mechanisms to be developed. One of the popular approaches helping to tackle this issue is the usage of *workflow technology* which is well-established in production management and business intelligence.

### 1.2.2 Workflows And Choreographies

The term *workflow* in its very generic sense describes a sequence of steps which defines a certain *process* [Dee+09]. *Workflow Management Systems (WFMS)* are specialized programming and run-time environments that provide means to *orchestrate* composed sequences of tasks with the help of service-based computing [Dee+09; SGG07]. This approach is particularly common for automation of production workflows [LR00] where the focus is on automatic execution of relatively fixed business processes across organizational boundaries. Well-established languages like BPEL [OAS06] and Business Process Model and Notation (BPMN) [Obj11] simplify execution and modeling of workflows [SGG07].

Compared to business workflows, scientific workflows are very dynamic in their nature and impose different requirements on workflow management systems. Every instance of a scientific workflow is a separate experiment with its own set of configuration data and preferences. Additionally, scientific workflows usually undergo rapid changes [SGG07]. Development of specialized software aimed to support scientific workflows utilizing the modern computing paradigms is an ongoing research area. Examples of e-Science workflow systems include Mayflower [SK13], Kepler [Alt+04], Triana [Tay+05] and many others.

The term *choreography* describes a global view on conversations among multiple workflow orchestrations. Choreographies behave as specifications allowing to utilize existing workflows as well as implement new ones [DKB08]. Collaborations could include multiple diverse partners with their own well-organized workflows willing to interact with each other. Modeling a choreography and looking at the overall process from the higher level might be very beneficial for the specification and understanding of an overall complex e-Science experiment composed of multiple scientific workflows. Various choreography modeling languages are available [DKB08] including BPMN, BPEL4Chor [Dec+07], or Web Service Choreography Description Language (WS-CDL) [Wor05].

### 1.2.3 SimTech Cluster of Excellence

The SimTech Cluster of Excellence is "an interdisciplinary research association in the field of simulation sciences" [Uni17]. It combines more than 200 researchers from various faculties of the University of Stuttgart. The SimTech Cluster of Excellence focuses on six areas of research including molecular dynamics, numerical mathematics and high performance computing. The Institute of Architecture of Application Systems (IAAS) has developed a workflow management system shaped specifically for scientific simulation workflows. At its initial state, the system allowed to orchestrate experiments as one workflow incorporating all different simulation components. In order to simplify the processes involving communication of multiple participants operating on different scales the notion of choreographies was introduced to the system [Wei+17]. Choreography modeling is performed using the BPEL4Chor [Dec+07] language and can later be transformed into executable BPEL workflows for every involved participant.

### 1.2.4 TraDE Middleware

Collaboration, being one of the key reasons for emergence of e-Science, at the same time is an additional factor which adds an implementation complexity. Choreography models which consist of multiple participants working with different formats and settings all have a specific *data flow* related to them. *Transparent Data Exchange* methods were introduced by Hahn et al. [HKL16] to support *data-awareness* during the choreography's lifecycle. Data flow modeling capabilities were introduced on the level of the choreographies. This allows separating the actual control flow logic from the data flow logic between participants as a prerequisite to enable transparent data exchange. We provide more details regarding these concepts in Section 2.1.

### 1.2.5 A High Level View On Data Transformation Tasks

The term *data transformation* is not new and has been used since at least 1947 [Bar47]. Commonly, it describes numeric data transformation in the context of statistics and exploratory data analysis [Fin09]. Transformation of the data might pursue various goals, e.g. meeting some assumptions for statistical inference or improving the overall readability of the resulting diagram. Some examples of transformation techniques include linear, root square, and Box-Cox transformations [GLH15]. Information integration is another field where this term is also used to describe a data preprocessing technique [RD00]. However, in this context transformation often refers to combining multiple heterogeneous data sources using schema and instance-level transformations. For example, a format conversion might be needed if data sources have different representations.

In the context of service choreographies and TraDE Middleware, transparent data flow is only achievable by means of data transformation. The modeling of data flow needs to capture data transformations in often complex cross-participant interactions relying on

different data formats and representations. While the difference between the modeling of choreographies and data flow is not in the scope of this thesis, it is worth mentioning that they might introduce some ambiguity. More specifically, a modeler has to decide whether a particular constituent is a part of choreography or it needs to be modeled as a part of a data flow. In this thesis we are only interested in the data transformation as means to support a transparent data exchange in cross-participant interaction. Ignoring the content of the transformation allows considering it as a function which transforms some vector of input parameters into an output parameters vector.

### 1.2.6 OPAL Simulation: A Motivational Example

To provide the reader with a better view on the topic we briefly discuss how the concepts of service choreographies and transparent data exchange can be applied to a Kinetic Monte Carlo (KMC) simulation which uses a special software called OPAL [BS03; Hah+17a]. Its name is an acronym which stands for Ostwald ripening of Precipitates on an Atomic Lattice (OPAL). This software allows simulating how copper precipitates are generated. More precisely, OPAL simulates a formation of clusters of atoms in a lattice due to thermal aging. Figure 1.1 illustrates a BPMN diagram of a simulation choreography using OPAL software with the TraDE concepts applied. One important aspect is that OPAL software is developed in Fortran and uses files as a unit of interchange. The software is provided as a set of executables representing different modules of OPAL which have to be wrapped as services in order to run as parts of the choreography. In Figure 1.1 the participants of the choreography are named with the *Opal* string in front whereas the data flow is represented by the grey cross-partner data objects [Hah+17a] connected with the dotted arrows. Such separation between all the data which is relevant for a choreography and its participants improves the overall readability of the choreography as well as optimizes the graphical models by reducing the number of data objects and inter-participant data flow.

Before the simulation process begins, an initial request containing parameters such as the total number of snapshots to take, initial energy configuration and a lattice have to be received by the participant named *OpalPrep*. Already at this point the data needs to be transformed. This data transformation is not a part of the simulation itself, but is required in order for the KMC simulation to start as the parameters need to be transformed into a supported input format. In *sim_input* data object shown in Figure 1.1, *energy* and *params* are transformed into *opal_in* by the service called “Prepare Input Files”. The KMC simulation is triggered by the *OpalMC* participant. Based on the passed parameters, the service responsible for the simulation creates a particular number of snapshots of the atom lattice’s state at certain points in time and produces the saturation data. Afterwards, this data is analyzed and visualized at the same time by responsible participants of the choreography. Without going into details, each snapshot is first checked for the presence of clusters by *OpalCLUS* participant and resulting cluster data is then processed by *OpalXYZR* participant in order to identify the position and size of the cluster. The resulting data for every snapshot is then grouped together. At the same time when the snapshots are being analyzed, *OpalVisual* participant is responsible for visualizing the snapshot and saturation

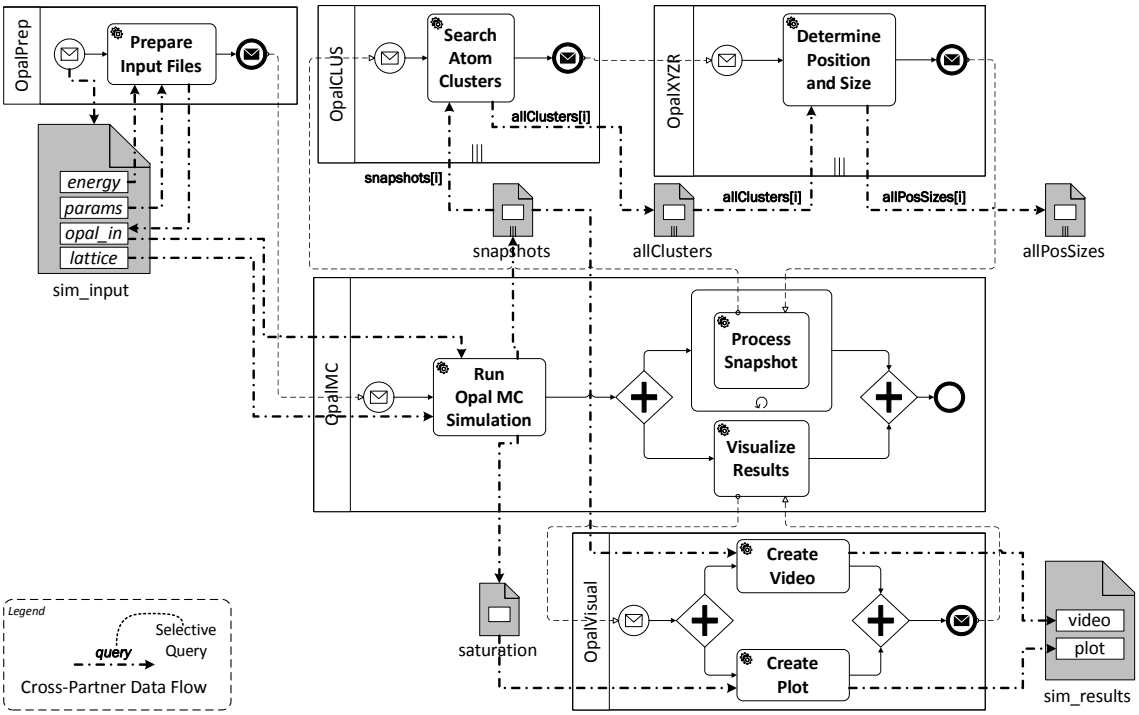


**Figure 1.1:** A BPMN diagram of OPAL simulation choreography with TraDE concepts applied [Hah+17a]

data. The former is visualized as an animated video of 3D scatter plots and the latter is used for visualizing a saturation function of the precipitation process as a 2D plot. Finally, the output media is passed to *OpalMC* and the simulation process finishes [Hah+17a].

During the execution of a simulation process one can notice several data transformations required for achieving a data flow's transparency. Moreover, it is also possible to say that in some cases the data transformation is needed to produce new information, e.g. deriving the clusters, while in other cases transformation serves more like a technical step. At this point, a modeler needs to decide which parts of the simulation process have to be modeled explicitly, i.e. as participants of the choreography. For instance, modeling the analysis of snapshots as participants in the choreography simplifies an overall readability of the simulation process. On the other hand, some parts can be modeled rather implicitly, i.e. as a part of a data flow. For example, generating a video and a plot can be seen as technicalities not directly related to the simulation process. A modeler, in order to simplify the choreography can potentially hide *OpalVisual* participant by means of modeling the involved transformations as parts of the data flow. Figure 1.2 illustrates the modeling of OPAL simulation in the actual software supporting the discussed concepts using BPEL4Chor [Dec+09] as underlying choreography modeling notation. The participants of the choreography are represented by white boxes with the lists of activities inside. The cross-partner data objects are modeled as grey boxes with blue headers and the data flow is represented by dotted arrows connecting the data objects and participants' activites. Even this relatively small model of the simulation looks quite complex especially for a person not

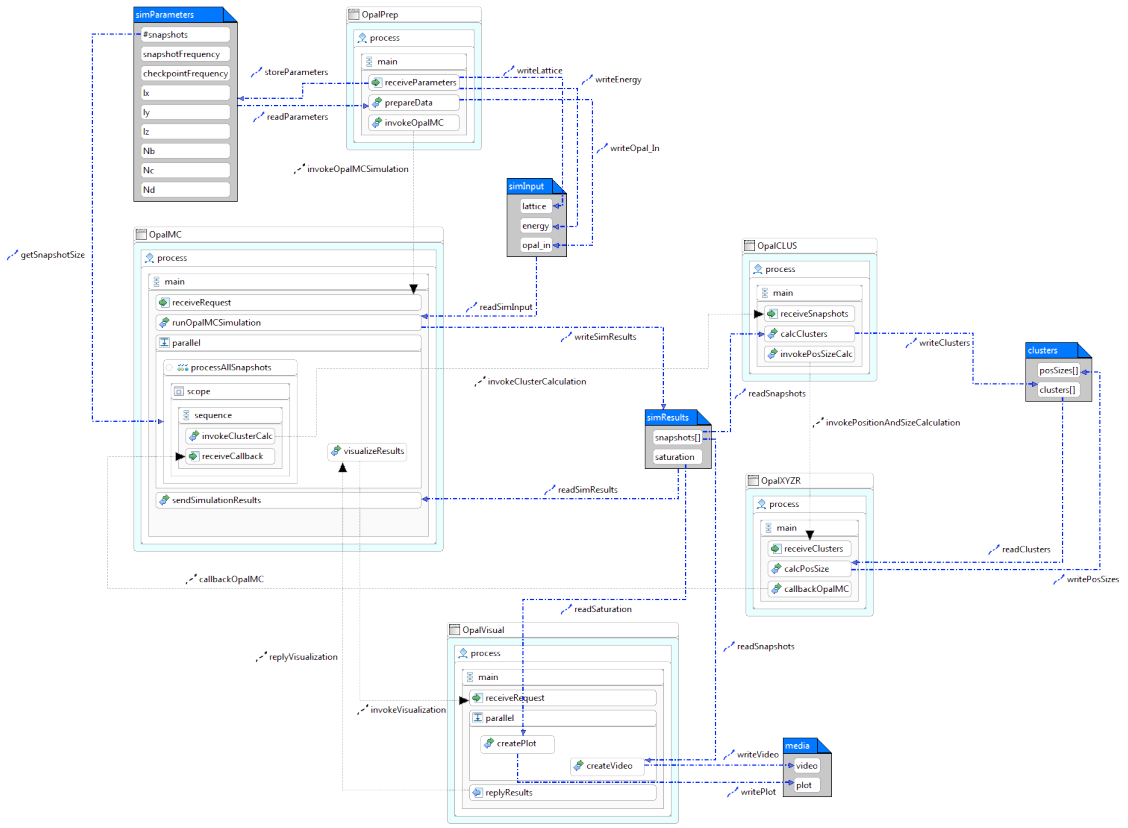


**Figure 1.2:** A modeling of OPAL simulation [Hah+17a]

familiar with the software. In this thesis we examine the ways of how handling the data transformations in service choreographies in the domain of e-Science can be simplified and combined with the TraDE concepts.

## 1.3 Problem statement

After an informal discussion of the topic's semantics and motivation behind it, we can formulate the problems which need to be solved:

***How heterogeneous data transformation logic can be uniformly addressed?***
Data transformation logic could be provided in multiple ways. For instance, it could be supplied directly by a participant of an experiment or a trusted third party, e.g., a publicly-available service can be reused. In any case, this logic represents an atomic application supporting one or more data transformation types. One of the problems is to identify the optimal solution among multiple available options of software reuse. Another part of this problem is to find a uniform way of communication with heterogeneous reusable applications.

***What are the most important aspects of handling the data transformation application?***
Analysis of the ways data transformation logic can be invoked is one of the most important parts of the work. However, there are multiple other issues to tackle. For instance, the following tasks might be considered: storage and search options, an analysis of the data transformation code suitability for specific data transformation task, error handling, mining data transformations in order to derive new insight, searching for composite data transformations.

***What architectural decision could unify the resulting concepts?***
Apart from identifying and prioritizing the concepts we need to understand which interaction models are suitable for combining various aspects together and what zoning should be applied to the set of concepts in order to fit them into the overall picture.

***How to integrate the resulting set of concepts with the TraDE Middleware?***
The most suitable option has to be chosen from the possible integration styles. Additionally, the expected communication behavior in the context of the TraDE Middleware has to be defined.

## 1.4 Main goal

The goal of this research work is to develop a framework for handling the data transformation applications used in service choreographies with the focus on the domain of e-Science. The resulting set of assumptions, principles and practices should be able to solve the previously described problems in a technology- and domain-agnostic way. Moreover, the resulting framework should support the integration with the TraDE Middleware. Finally, we will validate our concepts by developing a prototypical implementation of the proposed framework and by applying the results in a case study to the previously discussed motivational example as a proof of concepts.

## 1.5 Structure

The thesis is structured as follows:

**Chapter 2 – Background and Related Work** describes the background concepts underlying the research and discuss the related scientific work relevant to the topic of research.

**Chapter 3 – Concepts and Design** introduces the concepts of handling the data transformation logic and describes the design of the framework.

**Chapter 4 – Implementation** elaborates on the details of the prototypical implementation of the introduced framework.

**Chapter 5 – Case Study** presents the results of the case study showing how the introduced concepts can be applied to a real-world example.

**Chapter 6 – Conclusion and Future Work** summarizes the work and discusses the potential directions for future work.

# 2 Background and Related Work

In this chapter we focus on the theory underlying our research work. We discuss the basic concepts and review the relevant scientific work from various fields of research. The chapter consists of five sections each related to a certain research topic relevant for our work. Section 2.1 discusses the concepts behind the TraDE Midlleware and briefly outlines its architecture. Section 2.2 describes the notion of software reuse and its types as well as reviews the search methods for software components retrieval. Section 2.3 dives deeper into the topic of legacy code wrapping as one of the methods of software reuse and describes how this problem can be solved in various scenarios. Section 2.4 compares different virtualization techniques and provides a performance comparison information. In addition, this section describes how container-based virtualization can be used in e-Science world. Finally, Section 2.5 lists several container-based architectures in e-Science which focus on the encapsulation of scientific software.

## 2.1 TraDE Concepts

The introduction and motivational example briefly touches the concept of transparent data exchange. In this section we provide more details on the underlying theory. Consider a simple choreography example shown in Figure 2.1. It is modeled using BPMN and consists of three participants interacting with each other using messages [Hah+17b].

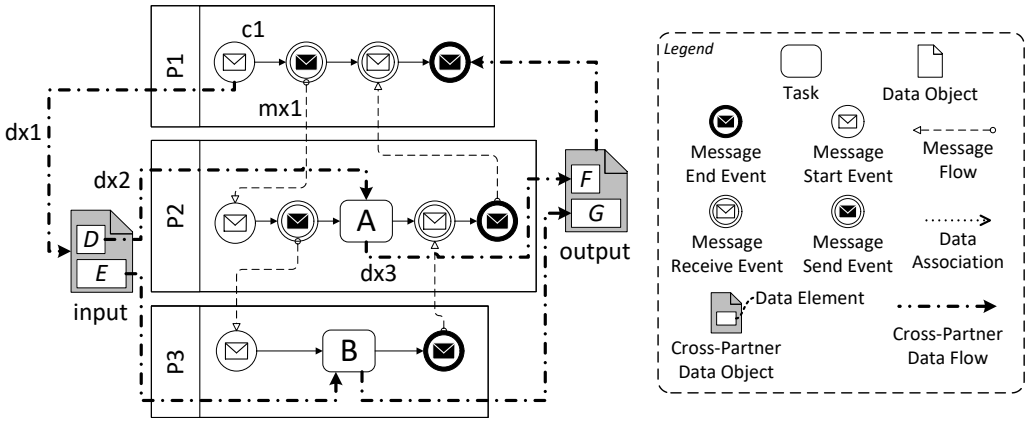


**Figure 2.1:** A BPMN model of a choreography with TraDE concepts applied [Hah+17b]

The cross-partner data objects and cross-partner data flows modeled as separate constructs in the diagram describe how the data is exchanged independently from the message

flow. Such separation of concerns allows modeler to focus on the common data model of a choreography without need to include the data objects into the message flow using, e.g. standard BPMN modeling constructs. A cross-partner data object consists of one or more data elements describing the actual data. A cross-partner data flow is the path of data interchange modeled independently from the message flow. It can use the cross-partner data objects as well as certain data elements as units of interchange. In classical choreography modeling languages such as BPMN the resulting models are often not directly executable because of missing technical details, e.g., transport protocols to use or the information required by a concrete process engine. The common approach is to transform the choreography into a set of private processes and refine them so that the resulting processes are possible to execute. In order to use the new TraDE modeling constructs the concept of choreography transformation was enhanced. As a result, at the transformation stage cross-partner data objects and cross-partner data flows are transformed into standard BPEL or BPMN constructs enriched with corresponding TraDE annotations. Linking the elements of cross-partner data objects with the standard data containers in private process models allows influencing the data interchange rules of respective process engines. For instance, if during the execution a private process needs to store the data in a data container which is linked to the element of a cross-partner data object then instead of storing this data internally, the process engine uploads it to the corresponding data element of the cross-partner data object at the TraDE Middleware [Hah+17b].

The final goal of modeling a data-aware choreography is being able to execute it. Every participant's process is executed using possibly a diverse process engine, i.e. a WFMS, which have to interact with other participants' processes based on the newly introduced cross-partner data exchange concepts. For this reason the special software layer called TraDE Middleware is implemented. It serves as a hub for inter-participant data exchange. The

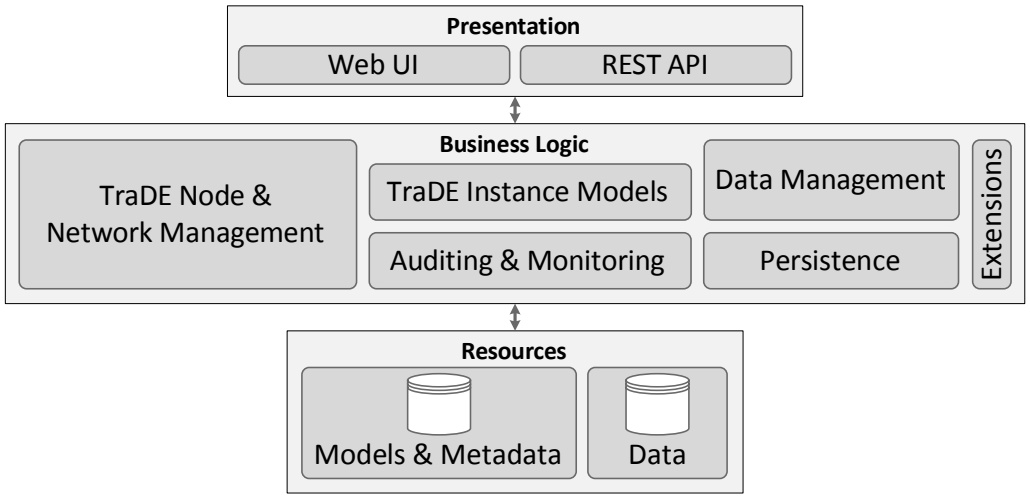


**Figure 2.2:** Architecture of TraDE Middleware [Hah+17b]

TraDE Middleware relies on its own choreography modeling language-agnostic metamodel and is not bound to a particular process engine. Figure 2.2 demonstrates the three-layered architecture of TraDE Middleware consisting of Presentation, Business Logic, and Resources

layer. The first layer provides means to interact with the software either via a Web User Interface (UI) or a Representational State Transfer (REST) Application Programming Interface (API). The Business Logic layer is responsible for multiple tasks including data management of cross-partner data objects, persistence which allows to decouple the data from the choreography instances, audit and monitoring tasks. Essentially, this layer groups all core functionality into several sub-components. The Resources layer provides all the data that is necessary for Business Logic layer [Hah+17b].

## 2.2 Software Reuse

The problem of a standardized addressing of heterogeneous data transformation applications described in Section 1.3 has a lot in common with the topic of *software reuse*. In order to distinguish the most suitable solutions we first need to provide a general view on the software reuse and discuss its various types.

### Historical Perspective and Modern Days

The concept of software reuse is not novel and dates back to Malcolm Douglas McIlroy's classic paper [McI68] "Mass Produced Software Components" presented back in 1968 at NATO Software Engineering Conference. McIlroy describes the bottlenecks in software engineering processes and emphasizes the importance of using industrial mass production techniques in software engineering. More specifically, analogously to subassembly in industry, he proposes to develop software by means of *component factories*. Instead of engineering the system *from scratch* it could be composed from reusable software components. In one of the examples, sine calculation is considered as a required-to-implement routine and the overall number of possible implementations complying to certain requirements equaled to 300. Instead of implementing all routines it could be more beneficial to pick a reusable software component from a dedicated repository. Component's suitability for reuse can be identified using a specific set of so-called *sale time parameters*. For instance, following dimensions could be considered:

- *Precision* of the output
- component's *robustness*, as a compromise between reliability and compactness
- choice of *particular algorithm* performing the calculation
- *generality*, as a degree of parameters' adjustability during runtime
- desired *interface* and *access* methods
- choice of *underlying data structures*.

The introduced notion of software reuse raises multiple questions. Among these questions are techniques for development and testing parametrized components, their categorization, distribution and delivery [McI68].

Horowitz et al. [HM84] emphasize that reusability comes in multiple forms, namely prototyping, reusable code, reusable design, application generators, formal specification and transformation systems, and Commercial Off-The-Shelf (COTS) software. Most of the listed above forms of reuse are linked to the concept of program generation based on some general-purpose language which in contrast to a Domain-Specific Language (DSL) is meant to describe a wide spectrum of applications. On the other hand, authors categorize the concept of code reuse as a rather narrow view on the problem. Additionally, Horowitz et al. list following problems of code reuse:

- **Identification methods**
  One of the main questions to answer is how to identify and specify reusable components independently of a domain.
- **Description techniques**
  Specification formalism choice is the next issue after the component has been identified. How can one describe the component in an understandable manner?
- **Implementation form**
  Next problem is which language should be used for implementation of the component. For instance, general-purpose programming language or a higher-level program design language might be used.
- **Categorization**
  Expecting the amount of components to be immense, how can similar components be described and grouped?

Moreover, the authors mention that code reusability could be problematic in situations when modification of the reusable component is needed. Such cases may require a deep knowledge of component's implementation which diminishes the advantages of code reuse [HM84].

Prieto-Díaz [PD93] presents a taxonomy of software reuse and assessed the state of research at that time. Following perspectives on reusability are discussed: (i) *By substance:* concepts, artifacts, processes; (ii) *By scope:* vertical, horizontal; (iii) *By mode:* systematic, ad-hoc; (iv) *By technique:* compositional, generative; (v) *By intention:* black-box, white-box; (vi) *By product:* source code, design, specifications, objects, text, architectures.

An extensive work by Mili et al. [MMM95] discusses various aspects and the state of research in the area of software reuse. In this work, emphasis is put on the type of artifacts being reused, listing data, architectures, detailed application design, and program reuse. The term *reusable assets* is used to describe both *products* and *processes* intended for reuse. Additionally, three categories of systems are listed:

- *reusable program patterns*, which describe the usage of source code or design patterns

- *reusable processors*, which are used for interpretation of high-level specifications
- *reusable transformation systems*, which involve developing activities in transformations

Reuse of domain knowledge in the form of models is helpful in multiple ways: it simplifies developer's understanding of the domain and serves as an initial point in system analysis. Also, it provides application-dependent classification of reusable components. Domain models incorporate entities and operations on them, constraints and relationships between the entities, and properties of objects which will be used for searching components. In other words, a domain model is a field-specific vocabulary describing entities, behaviors, and constraints. Furthermore, Mili et al. differentiate between issues of *developing* reusable assets and *developing with* reusable assets and describe details for both categories. While the former category is mostly about how to develop reusable building blocks using generative approaches, the latter incorporates issues related to component retrieval, composition and adaptation.

Ravichandran and Rothenberger [RR03] focus on advantages of black-box reuse in the context of *Component-Based Software Engineering (CBSE)* paradigm based on previously discussed concepts of software reuse. Software is built using compositional techniques with prepackaged components as building blocks. In its essence, a component is an executable piece of code providing a specific functionality via some interface. As an example of improvements in reuse component models like Common Object Request Broker Architecture (CORBA) [Obj17] and Enterprise JavaBeans (EJB) [Ora17b] are mentioned. The authors discuss differences of black-box vs. white-box reuse arguing that usage of the former can be highly beneficial. Additionally, a component-level reuse decision tree is presented with differentiation between in-house and market black-box components.

A literature review by Mohagheghi and Conradi [MC07] assesses software reuse effects in industry. For this purpose, eleven papers related to observational studies and experiments conducted in industry are analyzed. According to the authors, most of the papers are concerned with systematic reuse in the sense of having an explicit asset for reuse with a source code being a unit of reuse. One of the findings is a productivity gain for small and medium-scale studies. However, no consistent results for actual productivity are given. Another important point is the importance to identify the contexts which could benefit from reuse. The study suggested that black-box reuse and reuse with slight modifications resulted in lower development and correction efforts. An additional observation is the correlation between size and complexity of reusable assets and the frequency of reuse. Small modules or functions could be reused more often while larger reusable assets require complex design decisions.

Zaimi et al. [Zai+15] analyze third party libraries reuse in open-source software. The authors focus on the evaluation of reuse intensity, how reuse decisions evolve across time, and how reuse affects the product's quality. The study includes five Java projects and considered more than ten versions of each in order to monitor reuse history. As an outcome of the study the authors present several observations. Firstly, more libraries are reused over time signifying an overall increase of reuse intensity during the lifespan of a project. Secondly, developers tend to not revisit their reuse decisions. Thirdly, libraries

removal or updates occur rarely. In general, the study advocates for a more systematic and standardized approach in reuse as well as a regular review of decisions made.

The concept of *domain knowledge reuse* was already mentioned previously both explicitly [MMM95] and implicitly. For instance, the term *vertical reuse* is used to describe reuse of software within the domain, whereas *horizontal reuse* is applied across domains. Dabhade et al. [DSM16] present a survey on software reuse methods in the context of *domain engineering* processes. The development of components conforming to specific models should include such steps as categorization of components, specification of interfaces and protocols, as well as validation. While rigorous classification of reuse methods is not in the scope of this thesis, some methods described by Dabhade et al. are worth mentioning. *Program libraries and application frameworks* reuse basically operate at different levels of granularity in the sense of reusing dedicated sets of source code units tailored for a particular goal. *Legacy system wrapping* is a method of reusing an existing system in a new environment or applying different constraints. *Service Oriented Systems* is basically a concept of component-based software engineering applied to the world of service technology [Erl16].

The study by Vale et al. [Val+16] synthesizes academic research knowledge about CBSE in the period between 1984 and 2012. Therefore, the authors studied several questions, including research intensity in the field within the stated period of time, most investigated research topics, and in which domains CBSE has been applied. The interest in the field increased in late nineties and was at stable, but lower levels in 2010s. The authors assume that the lower rate of interest in the topic is connected with more rigorous reviewing due to large numbers of papers in the field as well as the integration of CBSE with new approaches such as software product lines, model-based engineering, software oriented architectures and cloud computing. The analysis of research topics popularity reveals that CBSE papers cover practically all areas of software engineering (such as testing, architectures, etc.) as well as specific concerns related to component-related topics (component specification, interaction and composition, component models, interfaces, etc.). Additionally, Vale et al. note that the field of CBSE is still active with multiple open problems.

To summarize, the concept of software reuse is an important part of modern software development. While various categorizations are presented in multiple papers, there is no standard classification of software reuse methods. In the context of this thesis, the concepts of *black-box reuse*, *component specification*, *search, retrieval, and composition techniques* are of great interest.

### Component Retrieval Methods

In order to find components which are suitable for reuse, efficient retrieval mechanisms are needed. This topic attracts researchers at least since early 1990s.

Podgurski et al. [PP93] explain the retrieval of executable software components based on behavior sampling instead of standard text retrieval methods. A component's behavior can be sampled by executing it using a user-provided input and comparing the output with the

user-provided output. The main idea is that if a component returns the expected values then it suits the requirements of a user.

Mili et al. [MMK+94] discuss various ways of component retrieval separating between the keyword-based methods which aim at returning a single suitable component and the composition techniques which discover component chains which suit user's needs. The so-called component composers might be considered in case a standard search does not return any suitable component. Mili et al. refer to the issue of component composition as *function realization problem* and prove it to be NP-complete.

Zaremski et al. [ZW95] describe a signature matching technique for component retrieval. For instance, if a user looks for a certain function, the function's type could be used as a way to retrieve a suitable one. In this case the function's type is the set of input and output parameters' types. Such signature is either provided or derivable because usually it is required by the compiler. The component can be either a function or a module consisting of multiple functions. Zaremski et al. discuss the differences between the function matching and module matching. Additionally, the authors consider exact matches as well as how matching relaxation can be achieved.

In subsequent work, Zaremski et al. [ZW97] discuss a specification matching as a way to determine if two components are related. This process underlies multiple questions including retrieval of a suitable component, reuse of the retrieved component, substitution of components, and whether one component is a subtype of another. A specification is considered to be a formal description of the component's behavior. Two components match if two conditions are sufficed: their signatures match and their specifications match. As with the previous work, function and module matching are explained and the matching relaxation techniques are discussed.

Bawa et al. [BK16] describe numerous retrieval techniques with their advantages and disadvantages, including:

**Keyword-based retrieval** is one of the simplest forms of search using the uncontrolled vocabulary. Basically, a user supplies free text keywords and the matching happens on the word-by-word basis with the list of indexed terms of the components. Techniques from information retrieval such as term frequency and inverse term frequency are used for components indexing. The resulting ranked list of components is returned to the user.

**Enumerated-based retrieval** is a one-dimensional retrieval technique based on mutually-exclusive category codes which can be further sub-coded. An example of such classification is the division between arts and science in a library. However, this technique is very strict in its mutual exclusiveness which makes it impossible to classify components as belonging to several categories.

**Attribute value retrieval** is a retrieval technique which uses the component's attributes. As with the library example, the attributes of a book such as its title, author, or publisher can be used to form a query looking for a specific book in a library. However, the set of attributes of a component might differ in various repositories or an attribute

can be dynamic for components meaning that it is not possible to know the set of attributes in advance.

**Faceted retrieval** is a technique which uses the domain knowledge to form a pre-enumerated set of so-called facets. Domain specialists carefully identify facets, e.g. the terms “function”, “procedure”, “objects” can be chosen as the functional facets whereas the terms “operating system”, “network parameters” as environmental facets. One limitation of this approach is to balance the complexity of facets structure: complex structure makes it more difficult for a user to understand it and the simple structure results in many components falling into the same classification.

**Query and browsing-based retrieval** relies on Graphical User Interface (GUI)-based user interaction with the component repository. In complex and ambiguous cases it can be more efficient for a user to navigate through the query’s results and view the components’ details.

**Other techniques** include signature matching, usage of Unified Modeling Language (UML) and Extensible Markup Language (XML) descriptions of components, behavior-based method, usage of genetic algorithms and many others. Additionally, combinations of the methods can be used for retrieval.

## 2.3 Legacy Code Wrapping

*Legacy code wrapping* is one of the software reuse methods. Compared to other reuse approaches, legacy software wrapping is a necessity rather than a free choice. Handling data transformation applications can also be seen as a work with the legacy code due to the fact that every application which is intended for reuse is an existing piece of software. Commonly, the software is not developed with the uniform access methods in mind and wrapping is the only possible way of integration. Therefore, in this chapter we provide more details on legacy code wrapping methods.

### 2.3.1 What is Legacy Code Wrapping

The term *Legacy Information Systems (LIS)* describes mission critical software which resist modification and evolution [Bis+99]. Such systems usually function due to historical reasons and share multiple common properties such as use of obsolete hardware, high maintenance costs, lack of clean interfaces, resistance to extension. Bisbal et al. [Bis+99] explain different strategies of coping with legacy systems namely, *redevelopment*, *migration*, and *wrapping*. Having the most impact on the system, redevelopment is basically a rewrite of the whole software system based on new requirements. Migration is less expensive and aims at moving the LIS to a new environment while keeping the original functionality and data. With the least impact on the system, the wrapping strategy aims to preserve the original functionality by surrounding it with new interfaces. Wrapping allows reusing legacy software with accordance to new requirements.

While the whole spectrum of legacy system modernization approaches (e.g. [ACD10; CD+00; FSB16; LZ16; RS16; SNLM16]) is not in the scope of this thesis, we further investigate available software wrapping approaches. Sneed [Sne00] discusses various types of wrappers and levels of granularity at which software can be wrapped. He lists four wrapper types, namely database, system service, application, and function wrappers. The main distinction is the type of object being wrapped. A database wrapper usually offers a common interface which simplifies access to one or more databases. Similarly, a system service wrapper allows clients to access standard services. Application and function wrappers operate on different scales. While an application wrapper encapsulates the whole program via a new interface, the function wrapper works with individual functions of a program.

In general, a wrapper is responsible for receiving an input from a source application, converting it to an internal format and passing it to the target application. After target program completion, the wrapper collects its output, converts it to a suitable external format and passes it to the source application. Thus, wrappers should have two interfaces: external public and internal private interfaces. Moreover, Sneed lists *message handler*, *interface converter*, and *IO simulator* as parts of wrapper's structure located in between external and internal interfaces. A message handler is responsible for maintaining input and output queues in order to guarantee processing order and correctness. An interface converter maps external and internal interfaces in both directions. In standard cases mapping is a 1:1 relationship, however this is not always the case. For instance, a 1:n relationship means that parameters have to be duplicated and m:n relationship requires careful values association. An IO simulator handles input and output operations of the wrapped object. It passes parameters from the external interface to the input and copies the output into the external interface's parameters. In other words, it encapsulates the target software via emulation of input and output functions without affecting it [Sne00]. Figure 2.3 demonstrates the connection between the introduced modules of a wrapper.

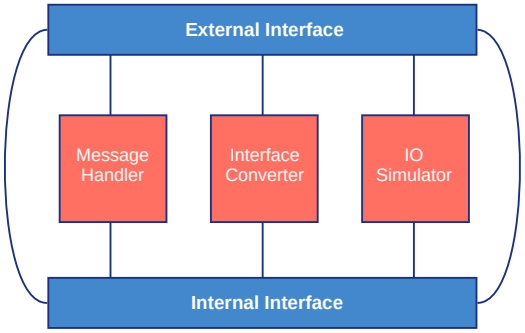


**Figure 2.3:** Modules of a wrapping framework [Sne00]

Wrapping solutions are usually oriented towards specific technologies, architectures and in order to improve readability we use following informal categories to group our literature review: *general frameworks and code wrapping techniques*, *service-oriented wrapping approaches*, and *grid and cloud-oriented wrapping approaches*.

### 2.3.2 General Frameworks and Code Wrapping Techniques

Juhnke et al. [Juh+09] introduce a method of wrapping a legacy code based on the Legacy Code Description Language (LCDL) framework. It provides an extensible legacy code specification model which later can be used for generation of executable wrapper code. The model incorporates all information required to describe binary and source code legacy applications. Emphasis is placed on the extensibility of the model which, for instance, allows the inclusion of new input / output sources or binding types. The main entity in LCDL model is called *service*. It consists of one or many *operations* which represent legacy code's methods and *bindings* which define a wrapper's type. Figure 2.4 demonstrates a simplified UML diagram of LCDL entities without properties. An *operation*

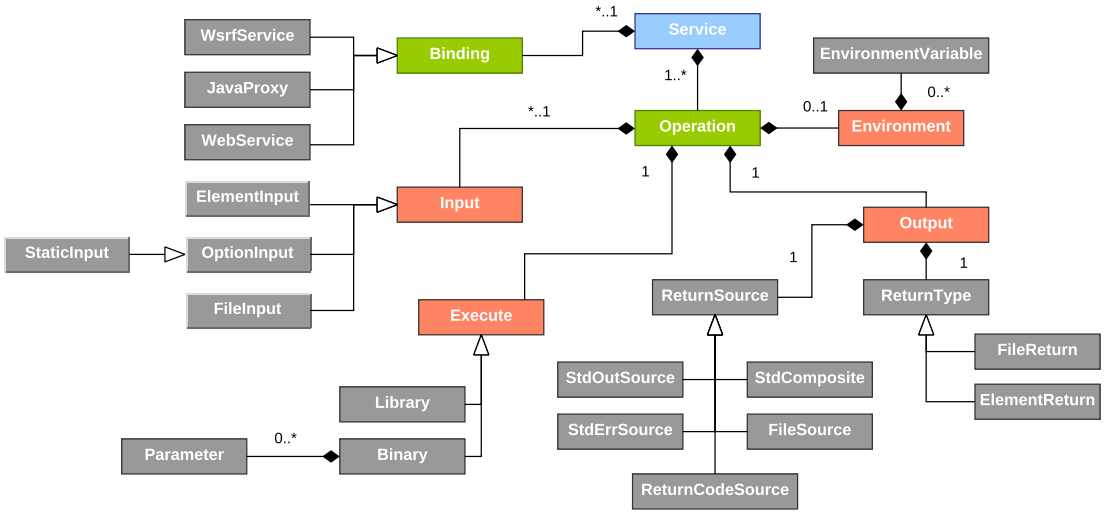


**Figure 2.4:** Simplified UML diagram of LCDL model [Juh+09]

incorporates several entities, namely, *input*, *output*, *execute*, and *environment*. An Execute entity represents information about the type of legacy code. A path to the library or binary needs to be specified. In case of a binary, explicit parameters might need to be defined. LCDL distinguishes the following input types: *ElementInput*, *OptionInput*, and *FileInput*. First one describes an arbitrary parameter of particular type which is passed to a Binding. An OptionInput represents a flag which can be set by a caller. StaticInput is a parameter which is always set. A FileInput repesents a file (the way of handling it depends on a particular Binding). As a next part, the output information must be specified. It depends on two elements: return source and return type. The former describes from which source the return value is obtained. For instance, it could be a standard output from a command line or a file. The latter defines in which form the output should actually be returned. Additionally, an Operation might require the specification of environment information. For instance, a specification of an environment variable [Juh+09].

The concept of *atomic domains* [HT02] is introduced by Haddad et al. as an attempt to switch focus from traditional component-level reuse to domain-level reuse. An atomic

domain is a collection of reusable assets having a common set of properties specific to related domains. Essentially, a subset of the domain which cannot be reduced is called an atomic domain. The inability to remove a component from a subdomain without changing the functionality or identity of the domain is the main property of irreducibility. As an example, the authors describe a set of operations related to robotic arm control (such as create an arm of any segments, rotate an arm around joint, extend/retract, etc.) as a robotic arm atomic domain. In a follow-up work they build a wrapper-based framework aimed at domain-specific software reuse on top of the atomic domain concept [HX06]. Figure 2.5 displays the general framework model. The main idea of the framework is to

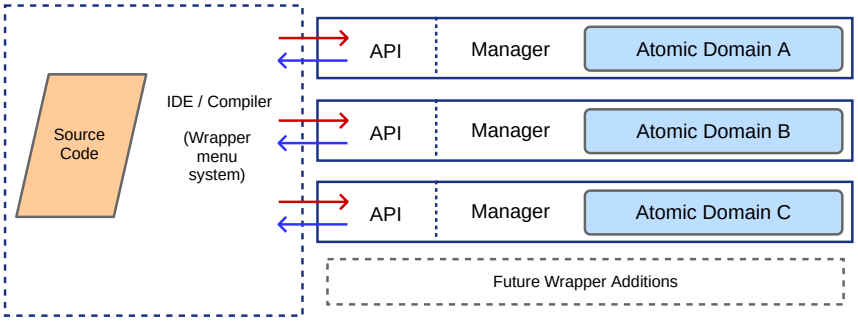


**Figure 2.5:** The general framework model [HX06]

reuse source code (modules, classes or functions) components related to atomic domains using domain-specific wrappers. Each atomic domain has a wrapper as its smart interface allowing the selection of a suitable component based on application's requirements. Haddad et al. introduce a wrapper's structure consisting of:

**Taxonomy:** consolidates vocabulary, patterns, algorithms and other information specific to an atomic domain,

**Application Programming Interface:** specifies data formats, parameters, constraints and expected behavior,

**Manager:** based on the API description the manager defines which component in the atomic domain is suitable for reuse and passes it to the client application,

**Communication Mechanism:** is responsible for presenting atomic domain components to the outside world, e.g. via messaging,

**Control Mechanism:** manages the program flow within the domain.

Assuming that atomic domains have been defined by domain experts, we briefly describe the remaining reuse steps. Initially, each atomic domain is mapped to a certain command which is referenced by an Integrated Development Environment (IDE). This command encapsulates all components within the domain and components can only be accessed via the manager. In case such a command is used in developer's code, during compile time the compiler sends necessary data to a dedicated manager via the wrapper's API. As a next step, the manager checks the command's context, identifies the suitable component and, if

this check is successful, it passes the component to the compiler. Afterwards, the single command in the developer's code is substituted with the received component [HX06].

Gannod et al. [GML00] introduce an architecture-based method for legacy command-line applications' wrapper specification and synthesis as well as discuss the aspects of wrappers integration with client software. The authors distinguish the following set of properties which appropriately describes command-line applications: *Command*, *Pre*, *Post*, *Signature*, and *Path*. A *Command* property describes how legacy applications can be invoked. *Pre* and *Post* properties describe commands defining pre- and post-conditions for legacy component invocation. A *Signature* property identifies the names and types of the input and output of the application. Finally, a *Path* property describes the path to the legacy application. In the proposed approach, the wrappers are generated using specifications which consist of the listed above properties. Additionally, such specifications are created using ACME [GMW10] Architecture Description Language (ADL).

Lee et al. [LCJ03] propose an XML wrapper API for Command Line Interface (CLI) systems. Their work focuses on problems of network management which commonly relies on the usage of CLI-based systems. Due to changes in the syntax of a command, its implementation might change as well. In order to tackle this issue, the authors propose to model groups of CLI-commands using XML templates and for further processing to convert these templates into actual commands using APIs. The main idea is to leverage hierarchical XML structures to define commands order and also support error handling in case a command in a sequence fails.

Wettinger et al. [WBL15] introduce a generic technique for generating public API implementations (APIfication) for executable programs. The aim is to avoid the tedious manual wrapping by means of standardized invocation mechanisms to support software reuse. Additionally, the authors present an extensible framework *any2api* as a realization of the introduced approach. The presented method does not necessarily aim at legacy software and can be considered as a general reuse methodology focused on executable applications. As a use case, the paper describes a deployment automation scenario for a web application in the context of cloud computing. To deal with the heterogeneity of technologies, interfaces, and implementation details, Wettinger et al. propose to wrap the invocation of various executables in a generic API. In the context of deployment automation this method helps to hide details about placement, runtime dependencies, or invocation parameters of an executable, as well as simplifies the return of formatted results. One important assumption underlying the presented method is that every executable provides a metadata description of input and output, dependencies, etc. The APIfication method which consists of eight steps is shown in Figure 2.6. Firstly, the desired executable is chosen for API implementation generation. Second and third steps are responsible for the definition of interface type (e.g., REST) and API implementation type (e.g., Java, Python). The next step (step 4) is about scanning the executable with its metadata to get the information about input and output parameters. Optionally (step 5), the refinement of information obtained from the previous step might occur. The generation of an API implementation happens next (step 6). To support API implementation's portability self-contained package is created (step 7), e.g., using Docker container engine. Finally, the resulting package is ready for use

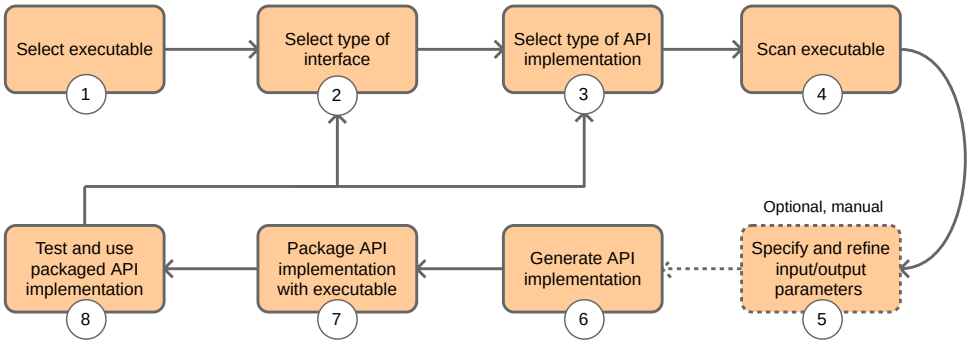


**Figure 2.6:** APIfication framework [WBL15]

(step 8) and from this point can be refined or modified by returning to selection (step 1) again [WBL15].

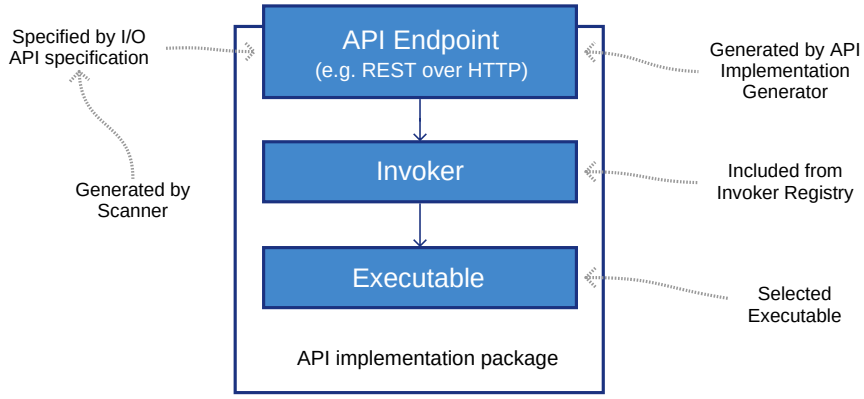


**Figure 2.7:** API implementation package [WBL15]

As a proof of concept, the any2api framework is implemented based on the described method. Therefore, Wettinger et al. introduce the concepts of *invokers*, *generators*, and *scanners*. An invoker is a special piece of software which is responsible for running at least one type of executable programs. If the required invoker is available in a corresponding invoker registry the next steps are performed. In order to proceed to the generation of portable API implementation, scanning has to be performed. A scanner is responsible for analysis of certain type of executables and the related metadata. As a result, it produces a specification which contains information about the input and output parameters, their data types and the mappings between the executable and its API to be exposed. A generator uses the resulting specification and corresponding invoker obtained from the dedicated registry to generate an API implementation. Finally, all artifacts are grouped by means of a self-contained package resulting in a portable API implementation supporting the chosen executable [WBL15]. Figure 2.7 demonstrates the structure of such API implementation package.

One can notice that the structure of packaged API implementation resembles the traditional wrapper shown in Figure 2.3. The *API endpoint* plays the role of an external interface

and the *executable* has its own internal interface. In its turn, the *invoker* incorporates functions of message handler, IO simulator and interface converter. However, the concept of packaged API implementations is on the higher level of abstraction as it also proposes wrapping the environment to support portability.

### 2.3.3 Service Oriented Wrapping Approaches

Zdun [Zdu02] conceptualizes a process of migrating legacy applications to the web.

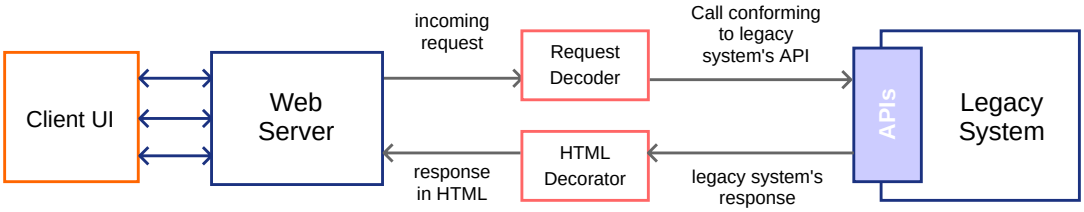


**Figure 2.8:** Simplistic Legacy to the Web architecture [Zdu02]

Therefore, the generic simplified architecture shown in Figure 2.8 is presented. Based on it, Zdun derives a generic process model consisting of four steps:

1. Provide an API to the web either using wrapping or redevelopment approaches
2. Implementation of a Request Decoder which maps Hypertext Transfer Protocol (HTTP) requests to wrapper's or legacy API
3. Implementation of an Hypertext Markup Language (HTML) Decorator which creates HTML representation of the response
4. Integration of implemented components with a web server

The simplified architecture is enriched by analyzing critical issues related to every step of the process listed above. The following issues are highlighted regarding legacy code wrapping: i) how requests should be mapped to the legacy system, i.e., how to approach such problems as responsible wrapper selection and invocation; ii) since HTTP is stateless, the wrapper has to maintain state and session information in order to handle asynchronous calls; iii) wrapper might need to be implemented not only for HTTP, but for multiple protocols which raises a question of wrappers integration and reuse; iv) how to abstract service providers in a wrapper in case the legacy system provides multiple services. Based on that, Zdun discusses various aspects of HTTP protocol handling (HTTP request handling, Uniform Resource Locator (URL) handling, session management and state preservation, authorization, encryption, logging, testing, and deployment) as well as content creation and representation approaches. Finally, a complex reference architecture integrating aforementioned concepts is presented.

Works by Sneed et al. [Sne06; Sne+06; SS03] discuss white-box approaches of wrapping selected functions from legacy software by means of XML descriptions and corresponding

wrapper generation. The authors describe the tool SoftWrap which is able to transform legacy functions' interfaces into Web Service Description Language (WSDL) interfaces. Supported languages are PL/I, COBOL, and C/C++. Apart from WSDL interfaces, this transformation is responsible for the creation of two additional modules which are used in a wrapper. These modules are responsible for transferring of input and output data between WSDL and legacy code's interfaces.

A white-box approach proposed by Guo et al. [Guo+05] focuses on generation of web services from Microsoft .Net legacy applications. The authors present a tool called Web Services Wrapper (WSW) which consists of two parts: analyzer and wrapper. The former is responsible for parsing and displaying legacy source code. The latter generates web service code for a chosen function and compiles it using the .Net compiler if restrictions (e.g., method must be public and not abstract) defined by developers are satisfied.

A method presented by Stroulia et al. [SERS02] in the context of the CelLEST project describes how to derive an executable specification of a service provided by legacy software by means of reverse engineering. This method follows the screen scraping approach in order to capture and model user interactions with legacy software. The main idea is to expose the desired parts of a legacy interface to a new interface. The presented CelLEST process is an interaction-based wrapping approach which creates a new interface working as a bridge between clients and the legacy software. It consists of five steps:

1. Tracing user interactions using the emulator tool
2. Modeling captured behavior as a state transition model
3. Mining task execution patterns
4. Modeling services based on mined patterns
5. Constructing web-based UIs for modeled services

Canfora et al. [Can+06; Can+08] introduce a technique of migrating interactive legacy applications to the web, i.e., involving form-based interactions with users. The goal is to provide a request/response-based interface for the legacy system. This cannot be easily addressed using techniques like screen scraping. Therefore, the authors propose a wrapping-based approach where the wrapper is responsible for autonomous handling of conversations between the legacy system and the client. However, in order to support conversations, the wrapper must know the conversation rules and be able to adapt to the execution flow. Considering the fact that a wrapper treats the legacy system as a black-box, the human-computer interaction model is needed in order to achieve this goal. The discussed resulting solution uses a Finite State Automaton (FSA) to specify the conversation behavior. The FSA is composed of states, transitions, and actions. A state describes changes from the system's start to the present. A state change is described by a transition which can be enabled in case some condition will be satisfied. An action defines what has to be done at a given moment. The resulting wrapper consists of an Automaton Engine, Terminal Emulator, and State Identifier. Additionally, a Repository is used to persist FSA and the corresponding Screen Templates (use case-dependent iterpretable descriptions).

Diamantini et al. [DPP05] describe how to automatically generate wrappers for Knowledge Discovery in Databases (KDD) tools in order to expose them as services. The authors implement an Automatic Wrapping Service (AWS) which generates wrappers for KDD tools. This enables the generation of a WSDL descriptor for a KDD tool based on an XML specification provided by a developer. This specification not only contains information necessary for wrapping, but also provides a formal description of the functionalities using a KDD ontology, optional linkable applications and tool performances. Furthermore, the paper provides a detailed description of specification information required for wrapper generation. Diamantini et al. distinguish the following data which needs to be specified: *name*, *input*, *output*, *language* and *description*. Input parameters can be obligatory, optional and hidden (in case they are needed, but not explicitly set by the client). The language specification describes Operating System (OS) and compiler-related settings.

Afanasiev et al. [ASV13] introduce MathCloud, a platform aimed at publication and reuse of scientific applications by exposing them as RESTful web services. One of the key points is to provide a unified interface which can be used to communicate with any kind of computational service. Commonly, the interaction between a service and a client in computational applications involves the following steps: (i) service receives a request to solve a task from a client; (ii) request contains a description of the task and includes all the required input; (iii) after request is processed, service returns output to a client. The authors propose a unified RESTful interface consisting of following resources: *Service*, *Job*, and *File*. These resources are identified by Uniform Resource Identifiers (URIs) and can be accessed using standard HTTP methods. A *Service* resource can be accessed using GET and POST methods. The former results in a service description being returned. The latter triggers the creation of a new task with the input specified in the request's body. A *Job* is accessible by GET method which returns the status and results of the job. Additionally, DELETE method can be used to delete the job and the resulting output data. A *File* resource can be accessed only using GET method which returns the results of the job.

The MathCloud platform consists of several components, including a *Service Container* also referred to as *Everest*, *Service Catalogue*, and *Workflow Management System*. Being the core of the system, the Service Container (Everest) provides adapters for various types of applications and implements a runtime for resulting RESTful services. The Service Catalogue provides means to discover, monitor, and annotate the deployed services. The WFMS handles the service composition tasks [ASV13].

### 2.3.4 Grid and Cloud-based Wrapping Approaches

Huang et al. [Hua+03] propose a semi-automatic approach to support running legacy C code as services which conform to the Triana [Tay+05] component model within a grid infrastructure. This approach relies on two tools, namely Java-C Automatic Wrapper (JACAW) and MEdiation of Data and Legacy code Interface (MEDLI). The former tool is responsible for automatic wrapping of legacy C code as Java code. JACAW takes C header files as input and produces corresponding C and Java files needed for making native calls.

The latter tool provides a GUI to facilitate conversion from Java code produced by JACAW to components used by the Triana scientific workflow system [Hua+03].

Delaitre et al. [Del+05] introduce Grid Execution Management for Legacy Code Architecture (GEMLCA) approach aimed at legacy application deployment as a grid service. It is a black-box approach suitable for multiple source languages including, e.g. Fortran, C, Java. Conceptually, GEMLCA allows exposing legacy applications which already run in the grid infrastructure as grid services. So-called GEMLCA Resource layer is responsible for presenting legacy applications to a client using an XML-based Legacy Code Interface Description (LCID) descriptor. This file stores information about the *execution environment* and a *set of parameters* required for the legacy application. For instance, the environment section stores data about the name and binary file of the legacy application as well as grid-specific information like which job manager should be used or the maximum number of allowed jobs. The parameters section describes a set of parameters along with their specific characteristics like name, friendly name, type (input or output), status (obligatory or optional), file or command line, regular expression which will be used to validate the input. Internal design of GEMLCA consists of three layers: *front-end*, *core*, and *back-end*. The front-end layer provides a set of grid services related to client's communication with the legacy code. The core layer is responsible for management of legacy code processes and jobs. The back-end layer facilitates communication with the grid middleware.

Glatard et al. [Gla+08] discuss how Service Oriented Architecture (SOA) can be advantageous for execution of scientific applications in the grid infrastructure. One of the main parts is the description of a generic web service wrapper which allows running legacy applications through a standard service interface. Another significant part of the work shows how the generic wrapper allows dynamic service grouping using the MOTEUR workflow system. The idea of such a generic wrapper is to expose a standard interface which will hide the grid infrastructure and which can be invoked by any client compliant to web services specification. To achieve this, a generic service relies on two different kinds of input: (i) legacy application command line format specification and (ii) legacy application's input parameters and data. Thus, the only required task a developer has to do is to create an XML descriptor containing the required information. Firstly, the *application's name* and *access method* have to be specified. The proposed implementation allows choosing between two access methods, namely URL or Grid File Name (GFN) which define how a wrapper will fetch the data. Glatard et al. distinguish two types of *input*, namely *data* and *parameters*. The former describes input files which can also be accessed using different methods and have additional command line options to be specified. The latter specifies command line values which are not files, thus there is no need to define access methods for them. Moreover, an *output data* access method and command line options have to be specified. Finally, *sandboxed files* might be defined. Those are the files describing all external dependencies like scripts or dynamic libraries which are implicitly required for the execution. As further steps, the authors describe how generic wrapping might be used for service composition by means of a workflow enactment system.

Balić et al. [BBW08] present the Legacy to Grid Framework (LGF) aimed at exposing legacy applications as grid services in a semi-automatic manner. The proposed structure of the

framework consists of three main parts: a *Service Client*, a *Service Manager*, and a *Worker Job*. A service client is responsible for transparent and isolated interactions between a client and a legacy application. In its turn, a service manager is a set of web services and their resources which are deployed in a hosting environment. This set of services is responsible for communication management between the client service and the worker job, creation of resources, and submission of worker jobs. A worker job is responsible for running the exposed functions of legacy application. Every client is mapped to exactly one worker job and the interaction between them happens via service manager's services. The worker job uses standard secure web service messaging for communication with other parts of the system.

Elkhouly et al. [Elk16] propose a Software Components as a Service (SCaaS) solution to deliver components for reuse based on the Software as a Service (SaaS) cloud computing delivery model. SCaaS allows deploying components on demand by means of cloud infrastructure and according to some licensing model. A queried component is firstly checked for compliance to requirements such as supported architecture, functionality, and interfaces. If check is passed a black-box reuse is used, otherwise white-box reuse is proposed. As white-box reuse involves modifications, the newly derived component is sent back to the cloud repository for future reuse. The authors focus on descriptions of *add* and *retrieve* modules. The former is responsible for adding components into a repository and consists of two indexing subsystems, namely concepts (functional description) and signatures (constraints) indexing. The latter allows searching for a component and proposes a suitable reuse method, e.g. black-box or white-box, depending on matching results, e.g. *exact match* or *similar*. The model of SCaaS is shown in Figure 2.9.

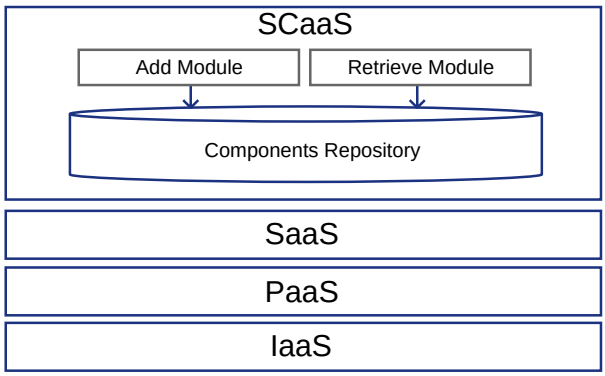


**Figure 2.9:** SCaaS model [Elk16]

Sukhoroslov et al. [SA14; SVA15] present Everest, a web-based platform for reuse of scientific applications which uses the Platform as a Service (PaaS) cloud delivery model. Everest is a further development of concepts from the aforementioned MathCloud [ASV13] platform. Everest consists of client and server-side parts. The former part offers a web user interface and client libraries. The latter is composed of three layers:

**REST API** is a single entry point for accessing the platform's functionalities. HTTP is used for requests with JavaScript Object Notation (JSON) being an interchange format. A

unified web services interface for accessing the deployed applications is implemented as a part of the REST API.

**Applications layer** is responsible for hosting the applications intended for reuse. An application is treated as a stateless black-box with sets of inputs and outputs which operates independently of other requests. Applications are created by clients who can also configure access permissions.

**Compute layer** is responsible for execution of applications using remote computing resources. Everest does not offer any infrastructure for the execution of applications and fully relies on computing resources provided by users. This layer controls a process of jobs execution. After the application is invoked via the REST API and the corresponding job is created, the Compute layer controls such actions as staging the input files, task submission and monitoring, or obtaining the output.

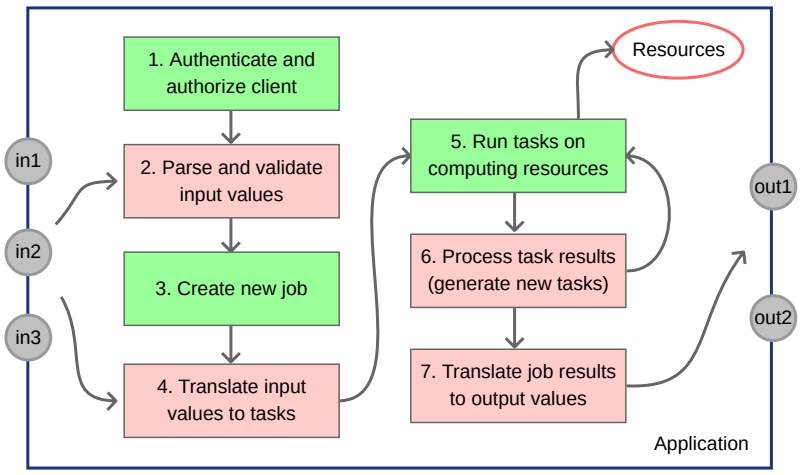


**Figure 2.10:** Structure of Everest application [SVA15]

Figure 2.10 demonstrates how a request to an Everest-deployed application is handled. Steps colored in green are application-agnostic and can be implemented similarly for any kind of application. However, steps colored in red depend on a certain application. Everest expects an application's description consisting of two parts, namely *public information* and *internal configuration*. The former provides specifications of input and output, as well as the information related to application discovery and how a client can communicate with it. The latter is responsible for forwarding requests to an application and generation of the results. For instance, the internal configuration of command-line applications includes the specification of mappings between the input values and the task command as well as the specification of how a task's output is mapped to the output values. As Everest does not offer its own computing resources, users can attach their own resources. For this purpose, a software called *agent* which runs on the resource is used. An agent supports various types of resources and can be integrated with them using corresponding adapters [SVA15].

Unlike in cloud or grid-based wrapping approaches, we are interested in decoupling the process of handling data transformation applications from a particular infrastructure. Firstly,

the cloud and grid infrastructures are not always available to the scientist willing to perform a computer-based experiment. Setting and configuring a private infrastructure can also be problematic in terms of resources and costs. Secondly, sharing applications by publishing it to, e.g. the cloud, is not always wanted by the scientist. We are interested in finding a simplistic way to publish and run data transformation applications in a technology and infrastructure-agnostic way which requires less effort from the publisher.

## 2.4 Virtualization

The invocation of heterogeneous data transformation applications is another problem from Section 1.3 which requires special care. Depending on the implementation technology, software might rely on particular dependencies, e.g. additional applications, thus making it impossible to invoke it on a computer not having such dependencies available. One possible solution to this problem is to use the virtualization technology. In this section we discuss the related work on various types of virtualization and their comparison. Additionally, we describe the relevant work on how the container-based virtualization might be used in the context of computer-based scientific experiments and which drawbacks it might have.

### 2.4.1 Hypervisor and Container technology

A *Virtual Machine (VM)* is commonly defined as an isolated and efficient replica of a real machine [PG74]. The concept of virtualization has been in the scope of numerous research papers since at least 1970s, e.g. discussed by Meyer et al. [MS70], Goldberg [Gol73], or Popek and Goldberg [PG74]. Popek and Goldberg [PG74] define a virtual machine as an environment generated by a *Virtual Machine Monitor (VMM)*, or *hypervisor*. VMM is a software which has three major characteristics: (i) it provides the environment for applications which is basically indistinguishable from the original machine; (ii) execution of the applications might only result in insignificant decrease of performance; (iii) system resources are completely controlled by the VMM. The authors differentiate between bare-metal (type-1) and hosted (type-2) hypervisors. The latter type operates on top of a host's operating system.

Hypervisor-based virtualization is a traditional way of achieving isolation and resource control for co-location of multiple workloads [Fel+15]. Workload's isolation guarantees the impossibility of interference with the execution of another workload. Resource control is an ability to bind the workload to a specific set of resources. When executing a workload inside a dedicated VM, these two key requirements are fulfilled because the VM both isolates the workload and constrains the resources (as it is configured with some resource limits). Compared to native execution, this, however, is more expensive performance-wise.

As historical inspiration for container technology, Bernstein [Ber14] mentions Unix operating system's *chroot* command (introduced back in 1979). This command allows changing

the root directory for a currently running process and its children. The resulting environment, for instance, can host a virtualized replica of a software system. An extended implementation of chroot called *jail* was added to FreeBSD in 1998. Six years later, an enhanced version of this concept was introduced in Solaris 10 operating system and called *zones*. Solaris 11 presented *containers* which were based on zones. With the advent of the Linux OS, an evolved container technology based on such concepts as *kernel namespaces* [BN06] and *control groups (cgroups)* [Men+17] replaced these previous variants. We focus more on Linux containers later in this chapter.

Being a lightweight alternative to the hypervisor-based approach, *container-based virtualization* operates on a different level of abstraction [MKK15]. As hypervisors operate on the hardware level they need to virtualize hardware and related drivers. Moreover, every VM uses a separate OS on top of the virtualized hardware. Figure 2.11 shows the structure of hypervisor-based virtualization with respect to the types of hypervisors. In order to

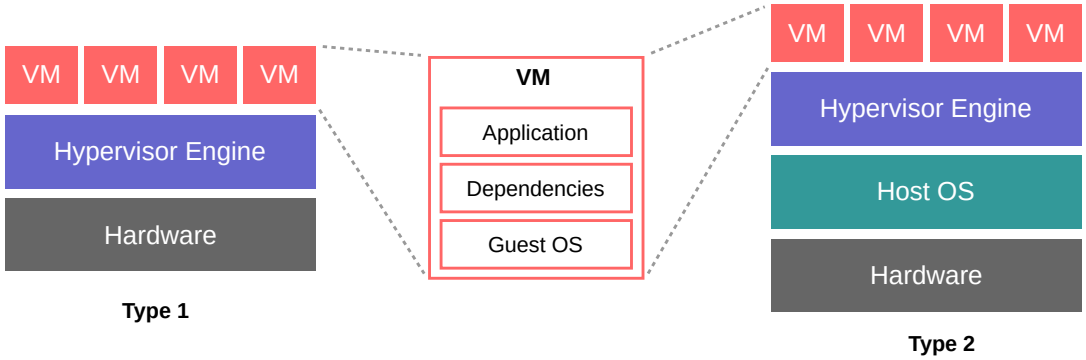


**Figure 2.11:** Hypervisor-based virtualization [MKK15]

avoid such overhead, containers switch the level of abstraction from hardware to operating system level. This is achieved by sharing the operating system's kernel. Every container runs on top of the host's operating system. Figure 2.12 demonstrates the structure of container-based virtualization.

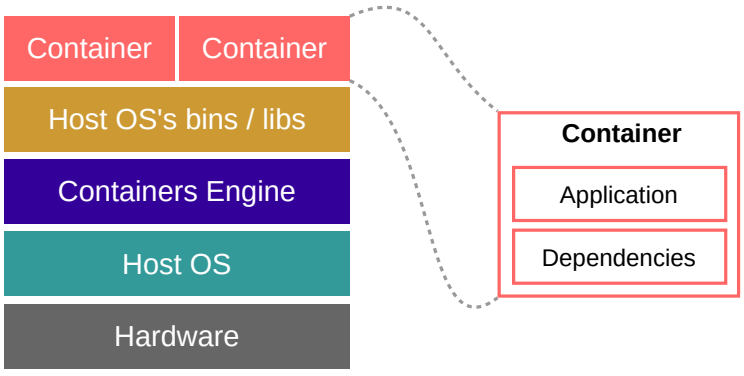


**Figure 2.12:** Container-based virtualization [MKK15]

As was mentioned previously, one of the features used in Linux containers is called *kernel namespaces* [Fel+15]. In a very general sense, namespaces establish "boundaries" for groups of logically-related objects. Kernel namespaces allow creating "separate instances of previously-global namespaces" [Fel+15]. When a process is associated with a namespace it can only see and access that namespace's resources. There are several kinds of namespaces implemented in Linux and each is responsible for isolation of a certain resource type. For instance, a mount (mnt) namespace is responsible for maintaining mounting points and a process ID (pid) namespace provides processes with unique identifiers. Other examples include network (net), interprocess communication (ipc), and UNIX Timesharing System (UTS) namespaces. This powerful mechanism can be used in multiple scenarios including the creation of isolated containers which cannot see and access objects located outside. As a result, processes seem to run within a container in the same way they run on a regular Linux system. However, they share the kernel with processes from other namespaces. In contrast to a VM, a container is not required to run a complete operating system inside and in fact can represent only one process. Thus, depending on the container's contents, it could be either a *system* or an *application* container. The former acts like a complete OS with system daemons like init, inetd, cron running. The latter only runs an application which results in less consumption of resources. In some cases, there is a need to relax the degree of isolation for a container. This can be achieved by sharing resources among the containers, e.g. using *bind mounts* which allows making the same directory visible for multiple containers. Also, an inter-container communication or an interaction between the host and container happens with the same efficiency as a regular Linux inter-process communication.

Another feature that is used in Linux containers implementation is *cgroups* [Fel+15] which allows grouping processes and control their resource usage. In the context of containers, cgroups is used for managing container's resources, e.g. memory limits and CPU consumption. Modifying container's boundaries is as easy as applying changes to a cgroup which corresponds to this container. Additionally, cgroup allows terminating all processes inside the container. One problem related to resource management is that resource constraints imposed on a container are not known to the processes running inside. This might lead to an over-allocation problem if an application will try to use all available resources of the system when in fact it can access only a subset of them.

Commonly, LXC [Lin17] is used as a synonym to Linux containers. However, LXC is one of many container management tools available today. Other examples of container engines are Warden [Fou17], Docker [Doc17b], and Rocket (rkt) [Cor17].

### 2.4.2 Comparison of Virtualization Techniques

According to Bernstein [Ber14], most commercial cloud providers use hypervisor-based virtualization, although there are examples of container usage to support cloud delivery models like Infrastructure as a Service (IaaS) and PaaS as well. The choice of a virtualization technique highly depends on the task's specifics and multiple research papers examine the differences between hypervisor-based and container-based virtualization focusing on

various details [DRK14; Fel+15; Joy15; LK15; MKK15; Mor15; Yam15]. This subsection briefly discusses findings in several of mentioned comparisons.

Dua et al. [DRK14] compares the usage of hypervisor-based and container-based virtualization in the context of the PaaS cloud delivery model. A description of various implementation options of PaaS using containers and virtual machines and the comparison of several container engines is given. The authors mention an increase in popularity of containers for PaaS solutions due to better performance and reduced startup time. Additionally, Dua et al. highlight three features that can help to improve the adoption of the technology, namely a standard container format, enhanced container security, and OS independence.

Felter et al. [Fel+15] compare the performance of non-virtualized Linux to hypervisor-based virtualization (specifically, Linux KVM [Kiv+07] feature) and container-based virtualization (Docker engine) by using various benchmarks and measuring the overall performance of database products like MySQL [Ora17a] and Redis [Red17]. Docker was either equal or better than the KVM-based solution in every performed test. Furthermore, both KVM and Docker had insignificant overhead for memory and CPU performance. A common approach is to use VMs for IaaS and containers for PaaS. Felter et al. mention the lack of technical reasons for such distinction which leads to a broader application spectrum for container-based technology.

Yamato [Yam15] presents an evaluation and comparison of performance for bare-metal, Docker-based, and KVM-based servers deployed on Openstack [Ope17]. The resulting findings predictably show that bare-metal outperformed both Docker and KVM. Docker performs better than KVM in most disciplines. However, file copy operations were better for the KVM-based server which resulted in a total index of Docker being not much higher than the index of KVM. Additionally, the startup time was measured for all types of servers. In this case, it took significantly longer time for a bare-metal server to start than for both Docker and KVM. Compared to KVM, Docker startup time took less time due to absence of necessity to boot an OS.

Morabito et al. [MKK15] compare the performance of KVM as a hypervisor-based approach, LXC and Docker container engines, and OSv [Clo17] which is an open-source lightweight cloud OS intended to run on top of a hypervisor. One of the findings in this comparison shows that containers introduce insignificant overhead. Although both LXC and Docker perform well, security might be an problematic issue.

Morabito [Mor15] presents a comparison of power consumption for various virtualization technologies including both hypervisors (KVM and Xen [Xen17]) and container engines (LXC and Docker). Results demonstrated no noticeable difference for idle state, CPU, and memory performance. However, for network performance's comparison results for hypervisors were different. In contrast to containers, hypervisors consumed more power because of additional layers in the hypervisor environment through which network packets have to go.

### 2.4.3 Virtualization For Research Reproducibility and Scientific Containers

The notion of reproducibility is a basis of scientific research. The results of experiments are usually published together with the steps describing how they can be achieved. In case a description allows obtaining the semantically equivalent outcome then the results are called reproducible. Follow-up scientific work can be confidently built on top of such findings. As was previously discussed, e-Science experiments are computer-based. Reusing the scientific software in many cases relies on the idea of reproducibility. Although software might seem to be deterministic in its nature, many factors influence the outcome of computations. In order to reproduce a software-based experiment, a scientist often needs to repeat exactly the same packaging, installation, and configuration steps which can be poorly documented or even partially omitted [PF16].

Piccolo et al. [PF16] describe various tools and techniques simplifying the computational reproducibility of research. Examples include the usage of custom scripts allowing automatic execution of software or software frameworks which simplify the handling of dependencies. Literate programming as a combination of narrative description with the code, workflow management systems, hypervisor and container-based virtualization techniques are also among the options to support reproducible research. Being the lightweight alternative to hypervisors, container technology and in particular Docker [Doc17b], attracts scientists as a means to achieve reproducibility. Containers allow preserving the whole execution environment including all the related dependencies. As a result, software which was packaged as a container is incredibly simple to install and configure.

Chamberlain et al. [CS14] discuss how scientific research can benefit from the Docker containers. The main idea is to package software as layered containers depending on the use case. A container which is used as a basis is called a *base container*. It packages the libraries and instruments required for the software to run. Both *development* and *release* containers are built on top of the base container. The former includes an additional software useful for the development and testing processes while the source code is not included and accessed directly from the host's file system. This simplifies keeping the source code in the latest state. The release container is also built on top of the base container. However, it includes the source code alongside its dependencies. As a result, a release container can be distributed for further scientific work reproducing exactly the same environment which was used for original experiments.

Boettiger [Boe15] analyzes technical challenges to computational reproducibility and discusses how Docker can be used as a solution. Boettiger lists four technical challenges preventing the computational reproducibility. Firstly, a need to recreate an original environment of the experiments, i.e. tackling the "dependency hell". Secondly, experiments often lack a precise documentation on how to configure and run the software. Another issue affecting the reproducibility is a so-called "code rot". Dependencies are not static and evolve over time, e.g. release of new versions with fixed bugs or modified functionality. The knowledge of software's tolerance to the changes in dependencies is also an important point in reproducibility. Additionally, Boettiger discusses the adoption complexity of some existing solutions like workflow technology or virtual machines. Often, the scientists are

not familiar enough with these technologies or it takes a lot of effort to adopt them in experiments. Using Docker containers, the dependencies can be packaged into a binary image of a container. This approach helps to encapsulate an environment thus releasing the stress from a scientist who is willing to reproduce an experiment. Moreover, the Dockerfile which is a file of a specific format allowing to build a Docker container image can serve as a proper documentation. Furthermore, the Docker engine supports the images versioning, packaging images into tarballs, and layering images using the latest stable releases of well-known software, e.g. Linux distributions. As a result, the scientist can examine if the image built using the latest versions is not affected by code rot comparing to the self-packaged tarball.

Although Docker is one of the most widely used enterprise container engines, it was not designed with scientific applications in mind, e.g. running containers in a high performance computing environment. One of its disadvantages is the requirement to run the Docker daemon under root privileges which introduces additional security risks [KSB17]. To tackle Docker's limitations in the context of scientific computing, a number of scientific container engines were introduced including Singularity [KSB17] and Charliecloud [PR16].

## 2.5 Container-based Approaches in e-Science

Nowadays, the container-based virtualization is a common way to encapsulate the scientific software. Not only the single pieces of software can be encapsulated using container engines like Docker [Doc17b], but also the whole software chains representing a particular computer-based experiment. This allows a scientist to focus on the computer-based experiment itself rather than on setting up the environment and configuring the respective software. This approach might be particularly useful in our context. Therefore, in this section we list some related solutions using containers as means to encapsulate and run the scientific software.

Hosny et al. [Hos+16; Alg17] present AlgoRun, a Docker-based container template which can be used for packaging and provisioning of command line algorithms via a REST interface. The main goal is to simplify reuse of scientific algorithms by packaging their source code with the corresponding dependencies as Docker containers built on top of a specialized AlgoRun template container. A scientist who is willing to publish an algorithm needs to provide a description of the algorithm conforming to a specification format, provide its source code, and create a Dockerfile based on the AlgoRun Docker image which serves as a wrapper allowing to interact with the algorithm via a REST API. The packaged algorithm consists of three parts:

**Description part** is a file in JSON format stored in the *algorun_info* folder. It contains the information about the authors and publishers as well as the details about input, output, and how the algorithm can be invoked via a command line.

**Algorithm's code part** is a complete algorithm's source code which is stored in the *src* folder.

**Dockerfile** is a Docker-specific file allowing to build a container image. This file has to be created by the publisher of the algorithm and the image has to built on top of the AlgoRun container image.

Moreews et al. [MSM+15] introduce BioShaDock, a registry for bioinformatics applications wrapped into Docker containers. This software provides means for authorized users to describe, register and build public Docker images. In order to register a Docker container a user needs to provide a Dockerfile and additional metadata describing the scientific application. After the container is registered, the image building process as well as additional integration steps are performed automatically on a specified server. After the image is available in BioShaDock it can be used by a scientist, e.g. on a local machine using Docker or on a cluster which integrates a Docker scheduler front-end.

Kim et al. [Kim+17] propose a technique for execution of multi-step bioinformatics pipelines as pre-configured Docker containers called Bio-Docklets. The idea is to encapsulate the complex behavior into a Docker container and expose a single input and output endpoints for user which allows executing the pipeline. From the user's point of view the invocation is identical to running one particular application. The processes of instantiation, controlling the execution are controlled by special Python scripts.

Belmann et al. [Bel+15] present an approach to expose bioinformatics software using Docker containers with a standardized interface called *bioboxes* describing which input files and parameters are required and what is the output result. The focus is put on a need to standardize the interface for containers in order to achieve interchangeability in bioinformatics pipelines.

To summarize, the idea to use container technology for packaging an application looks very appealing. The container encapsulates all required dependencies and a uniform interface might be created in case a proper specification is provided by the publisher of the software. None of the listed solutions suits our needs due to various reasons: tight coupling with a particular container technology, need to manually create technology-specific descriptions, e.g. Dockerfiles, lack of automation and extensibility of the underlying conceptual models, or overall complicated reuse process. As we considered related work from the field of bioinformatics, most of the approaches aim at targeted reuse of a particular bioinformatics algorithms or pipelines. In contrast, we are interested in an extensible and technology-agnostic way of reusing the data transformation applications requiring less efforts on the consumer's side.

# 3 Concepts and Design

In this chapter we present our concepts and design the software framework which supports handling data transformation applications. The chapter is divided in eleven sections and starts from the assumptions in Section 3.1 which serve as a basis for the concepts. Section 3.2 analyzes various interaction scenarios among the actors involved in the process of handling the data transformation applications. Section 3.3 focuses on the description and packaging of the data transformation applications. Section 3.4 presents the abstract design of the framework for handling data transformation applications. The remaining sections focus on specific parts of the framework developing them and presenting the refined versions. Section 3.9 introduces the refined framework's architecture. Section 3.10 presents more details on the interaction with the framework. Finally, the description of how the presented concepts can be integrated with the TraDE Middleware is given in Section 3.11.

## 3.1 Data Transformation Logic and TraDE Middleware

The specifics of a data transformation application might vary depending on the usage scenario. For instance, source code modifications or user interactions might be required by certain applications. The diversity of usage scenarios make it difficult to derive a common abstract model for data transformation applications suitable for any possible case. Considering the high-level view on the data transformation in the context of TraDE Middleware we can describe the target application being similar to a first-order function which takes a vector of input values and produces a vector of output values. However, this simplistic view is not really suitable for cases where, e.g., a user interaction is needed. In general, multiple details affect the way how we can abstract the data transformation applications. The concepts presented in this work are based on several assumptions allowing to clearer define the data transformation applications.

**Black-box approach** Treat heterogeneous data transformation applications as atomic reusable entities. This assumption is based on several ideas. Firstly, an application can be provided as a binary executable, thus making the white-box analysis complex and unreliable. Secondly, the provided application must remain immutable throughout its lifecycle in order to guarantee the behavior desired by its provider. This also implies, that provided applications require no modifications and are ready-to-use.

**No user interaction required** As was discussed in the related literature part in Section 2.3, the wrapping of user interaction is a complex task which can be tackled by

maintaining the states of interactions, e.g., using a finite state automaton. The common way of performing data transformation in e-Science relies on files and often does not require a user's participation apart from the initial invocation routine. While handling an interactive data transformation application is an interesting and challenging task, in this thesis we focus on automatically-performed data transformations.

**Application is runnable** We assume that a provider knows the requirements of the data transformation application including its dependencies, configurations and the overall execution routine. This implies that apart from what was specified by a provider no further information is needed to successfully run an application.

**Applications exchange files** There are various ways of how applications can accept and produce data. Files are frequently used in e-Science as one of the most portable ways to transfer data among diverse scientists. While options such as using input/output data streams or printing the data onto the screen are available, files can serve as a good starting point for describing the data transformation in the context of scientific workflows. In this work we mostly focus on the applications accepting files as input and producing files as their output. Nevertheless, the concepts presented throughout the work can be extended in order to suit other data exchange cases.

## 3.2 Interaction Models and Main Actors

The process of reusing the data transformation logic relies on several distinct roles. As a starting point, we can distinguish three main actors analogous to the classic SOA roles: an *application provider*, an *application consumer*, and a *registry*. Basically, an application provider makes the data transformation logic available for reuse. An application consumer is interested in reusing the published data transformation logic. This interaction is made possible via the application registry which serves as a hub for providers and consumers. Worth mentioning that the application publisher and the actual provider are not necessarily the same roles. At first, we consider a scenario where the publisher and provider of a service is the same role and the data transformation applications are only implemented as web services. The simplified SOA-like interaction model is shown in Figure 3.1. One of the biggest advantages of SOA which we can benefit from in this case is an implementation transparency. The invocation of a desired service happens at the provider's side only using the binding information obtained from the registry. In such an interaction scenario, the registry only offers means to find a service and provides its corresponding binding information. The technical details about the actual implementation remain unknown to the consumer. However, the type of applications intended for reuse is not limited to web services. This means that invocation of a published application is not achievable in a standardized manner as in SOA, e.g., using a WSDL interface. Hence, the handling of non-standardized applications puts an additional stress on every distinct role in the interaction model. Although the application is still invoked on the side of a publisher, the information about the custom interfaces has to be present at the registry. Additionally, the access information like Secure Shell (SSH) credentials might be needed as well. The lack of

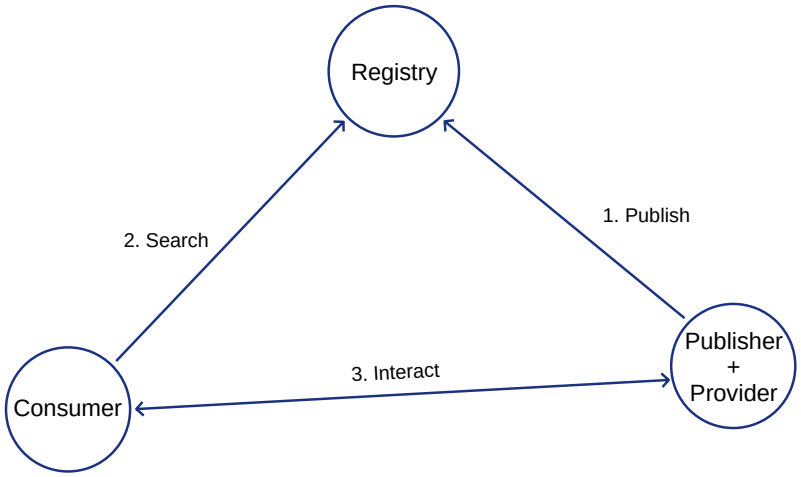


**Figure 3.1:** Simplified SOA-like interaction model

standardized access requires an application consumer to individually handle each desired application. In case of high reuse costs, the usage of such architecture is not beneficial. Moreover, in many cases a standard invocation is not even possible without additional wrapping, e.g. an application's response might be just saved as a file on publisher's side. An example of such interaction model is shown in Figure 3.2.

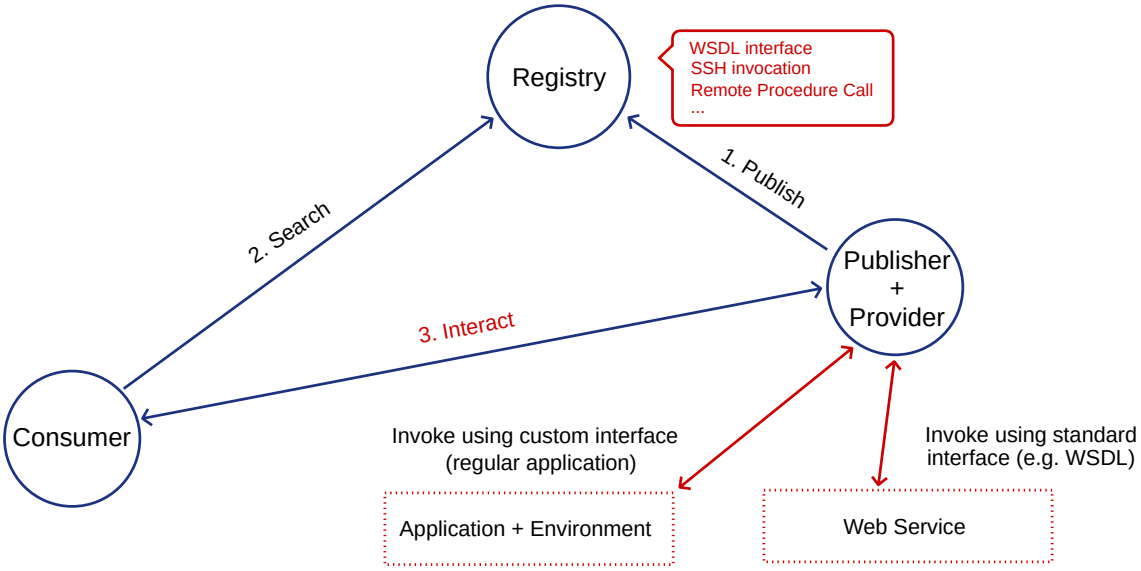


**Figure 3.2:** More refined SOA-like interaction model

In fact, these problems are among the reasons why the SOA concept was introduced after all. Unfortunately, handling different application types in a SOA fashion is not directly applicable in our case unless all applications are wrapped as web services prior to publishing. Such scenario is desirable yet not realistic. Wrapping an application as a web service is not a trivial task and requires special skills. However, the scientists who intend to publish their

data transformation applications for reuse are not necessarily the experts in the field of service technology. A requirement to wrap the target application as a web service before publishing is a limitation which should be avoided.

The publisher and provider roles were combined in previous examples. The separation of these roles allows discussing the ways applications can be published and how it affects the provisioning. An example of the interaction model with distinct publisher and provider roles is shown in Figure 3.3. As with the previous models, the consumer role is interested in reusing the available applications matched by some search criteria. Multiple publishers are responsible for making the applications available via the registry. An intermediary zone referred to as a *hub zone* incorporates the registry and multiple providers. The hub zone is meant to separate the publishing from consuming roles and highlight that provisioning happens independently. This modification is not really different from the previously discussed models, because the consumer faces the same difficulties working with the various custom interfaces as before.

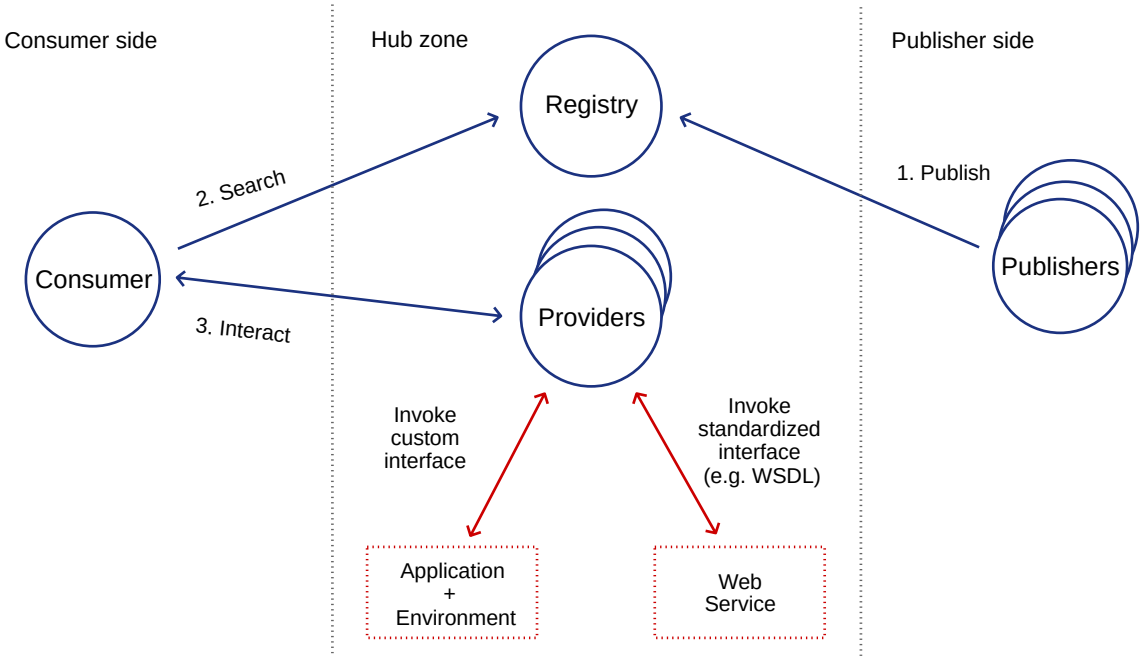


**Figure 3.3:** Interaction model with separate publisher and provider

Interestingly, the provider's responsibilities depend on what information is offered by the registry, i.e. how the applications are published. On the conceptual level, publishing a web service implies that it is available at some location via a standardized interface. On the other hand, publishing a regular application does not necessarily mean that it has to be available somewhere. In fact, the way an application is published defines how it can be provided later. For instance, if the application is published as a standalone self-contained package then a completely independent actor who has means to invoke such packages might act as a provider. Conversely, if the interface and invocation details are the only information offered by the registry then only the actual provider is capable of invocation. These different

scenarios show that a published application can be classified either as remotely-available or self-provisioned. The former case is about a consumer communicating directly with an actual provider. The latter case allows the consumer or some intermediary role to become a provider, assuming that the capabilities to instantiate and invoke applications are supported. A combination of both is also possible, however this introduces additional complexity, e.g. which provisioning type is preferable. The interaction model shown in Figure 3.3 does not reflect how a consumer can actually interact with different types of providers. In fact, this interaction is as inconvenient as in previously discussed models. A consumer needs to handle each provider's technical details independently which is problematic in e.g. automatic reuse scenarios.

One way to standardize the interaction between a consumer and various data transformation applications of choice is to introduce some proxy role which will be aware of application's provisioning details. On the conceptual level, this role separates the outside transformation interface from the inner application-specific interfaces. Considering the black-box view on the data transformation, the outside interface can be generalized in order to uniformly support application-specific interfaces. In other words, the purpose of the proxy provider role is to wrap interfaces of data transformation applications. However, instead of forcing publishers to wrap their applications the actual wrapping can happen afterwards by means of a proxy provider role which is aware of the application's provisioning details. Figure 3.4 illustrates an interaction model with a proxy provider role.

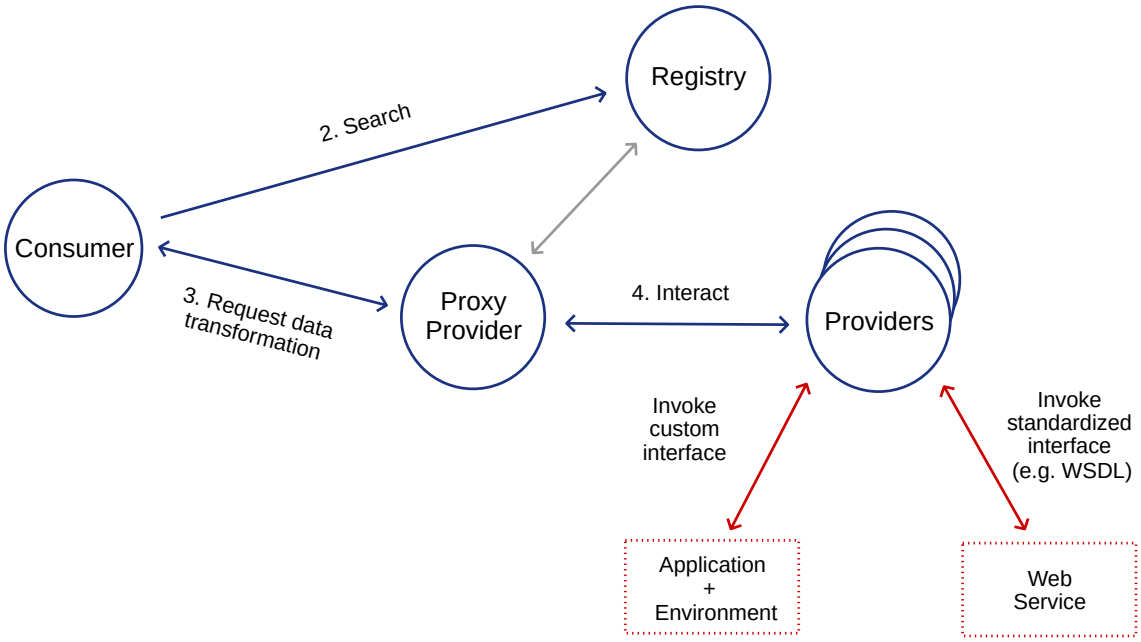


**Figure 3.4:** Interaction model with a proxy provider role

There are several ways of how a proxy provider role can fit into the interaction model. One obvious option is a standalone role which is responsible for mediation between the uniform transformation requests and the actual application interfaces. From the implementation point of view, this role can be either a completely separate piece of software or be a part

of a consumer. In the former case, a consumer must be able to invoke a standardized transformation interface at some location (potentially obtained from the registry) and the proxy provider role must be able to instantiate an application or handle the web service invocation based on the provisioning details obtained from the registry. As a result, less effort is needed from a consumer point of view, but a separate proxy provider role has to be implemented. In the case when a proxy provider is a part of the consumer, a set of proxy providers related to the corresponding provisioning type must be implemented. Additionally, the consumer must be able to launch packaged applications or support other provisioning types. Such scenario puts a lot of burden on the consumer and is not very practical. From the integration point of view, this is similar to the problem of multiple distinct wrappers versus an enterprise service bus. An increase in implementation complexity diminishes the benefits of reuse for multiple independent consumers. Sending a standardized request to some location and getting a result of the data transformation looks much more appealing.

Another option is to unify the proxy provider role with the actual provider at publishing time based on its provisioning type, i.e. for every provisioning type use a standardized wrapper type. As in the previous example it requires having an implemented set of generic proxy providers conforming to a uniform transformation interface on the outside and tailored for a specific inner interface, e.g. handling a remote web service invocation or invoking a command line application. However, these generic proxy providers do not need to be implemented by a consumer because the unification happens at publishing time. This puts additional burden on the registry role. On one hand, the resulting interaction model looks similar to the case when a proxy provider is a standalone role. On the other hand, a consumer himself must be able to launch the resulting wrapped application packages and interact with them. The registry in this case serves more like a static files repository for downloading the wrapped data transformation applications and running them on the consumer's side. Such scenario has several disadvantages. Firstly, it introduces the implementation overhead for the consumer. Secondly, the inclusion of identical proxy provider inside every application package based on its type introduces redundancy. Moreover, the consumer might be willing to use a slightly different implementation of a proxy provider. The scenario with a standalone proxy provider role allows postponing this decision to the time when a transformation is requested.

In the context of this work, we are interested in providing a standard way to invoke any published data transformation application for any interested consumer. The interaction model with a standalone proxy provider shown in Figure 3.4 reduces implementation overhead for consumers thus making it more attractive for reuse. In further sections we discuss the design and implementation details of this interaction model.

## 3.3 Description and Packaging

### 3.3.1 The conceptual meta-model of a data transformation application

In order to reuse the data transformation applications in a standardized way, uniform description and packaging mechanisms are needed. The previously described LCDL framework [Juh+09] or the AlgoRun container template [Hos+16] are used as an inspiration for how the data transformation applications can be conceptualized for wrapping purposes. A resulting conceptual meta-model shown in Figure 3.5 illustrates the main entities of the data transformation's application. This model is made extensible and can be modified to support additional types of entities depending on the use case.

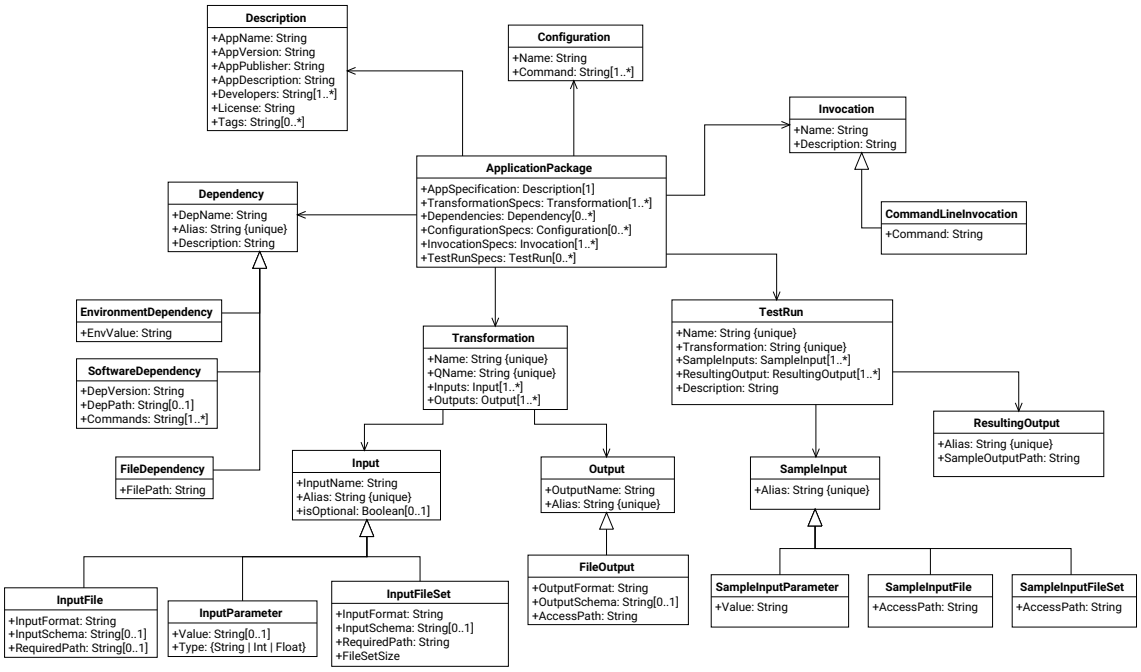


**Figure 3.5:** The conceptual meta-model of a packaged data transformation application

The main entity in the model is called *ApplicationPackage* and it consists of multiple properties describing various aspects of the data transformation application: AppSpecification, TransformationSpecs, Dependencies, ConfigurationSpecs, InvocationSpecs, and TestRunSpecs. The *AppSpecification* property is of type *Description* and contains the general information about the application. The *TransformationSpecs* property is a set of one or more data transformation specifications described by the *Transformation* entity. The *Dependencies* property is a set of zero or more application dependencies described by the *Dependency* entity. The *ConfigurationSpecs* is a set of zero or more application-specific configuration commands described by the *Configuration* entity. The *InvocationSpecs* is a set of one or more specifications of how an application can be invoked described by the *Invocation* entity. Finally, the *TestRunSpecs* is a set of zero or more specifications of how a particular data

transformation supported by the application can be verified for correctness described by the *TestRun* entity.

Multiple applications can perform the same kinds of transformations or can even be semantically equivalent. A set of general properties related to the application's description is one of the parts required for the search and publish operations. The *Description* entity consists of following properties containing a textual information related to an application: name, version, publisher, description, developers, license, and the list of tags. While most of the property names are self-explanatory, some observations might be done for further concepts, e.g. search and publishing which will be described in later sections. A specification of the application's name is insufficient as a unique identifier because applications can have the same names. It is rather an informal way to address an application. As a consequence, some combination of the properties should be considered in order to uniquely identify the application. The actual format of the version is not enforced in the model and a simple string of any format might suffice. It can serve as an additional mechanism for versioning of the same applications (the ones which have only the version property different). However, if used as a part of the unique application key, the version property will separate the different versions of the application as unrelated applications. Application's publisher is another string property which adds more details but cannot be used on its own to characterize applications. For instance, the same application can be published by multiple different publishers in case it was previously shared using other means. Depending on how this property is filled in, it can be either implicitly linked with a publisher's credentials or explicitly provided by the publisher. Next property in the list is an application's description. It is intended solely for human-readable description of an application which can be used for GUI-based work with the registry. For instance, having a proper textual description might simplify the manual search for an application or manually-performed administrative tasks. The application's developer and licensing information is reflected in the corresponding properties. Multiple developers can be specified and in comparison with the publisher information, this information might look more trustworthy as a part of the application's identifier. However, as multiple developers can be specified, their order in the specification plays an important role for generating an identifier. Additionally, there is no guarantee that developer information will not be partially omitted if the same application is published several times by different publishers. The last property providing the general information about the application is called Tags. More precisely, this is a set of strings which can be used for application search. For instance, some data transformation applications might be tailored for very specific tasks related to certain projects and general purpose applications simply do not fit in these scenarios. A publisher can provide a set of exact tags, e.g. a project name or custom transformation format in order to simplify the search process. However, the consumer will be expected to know the exact tags in order to find this application.

In theory, a single application can be responsible for multiple kinds of data transformations, e.g. a conversion to several image formats like JPEG and PNG based on specified output file extension. Thus, the single application package might have multiple transformations specified. More precisely, at least one transformation has to be present otherwise the package specification makes no sense. Perceiving the application as a black-box, we can

describe its transformations using its inputs and outputs. Additionally, it has to have a unique name. An application can consume one or more inputs and produce one or more outputs. The *Transformation* entity describes supported data transformations and is shown in Figure 3.6.

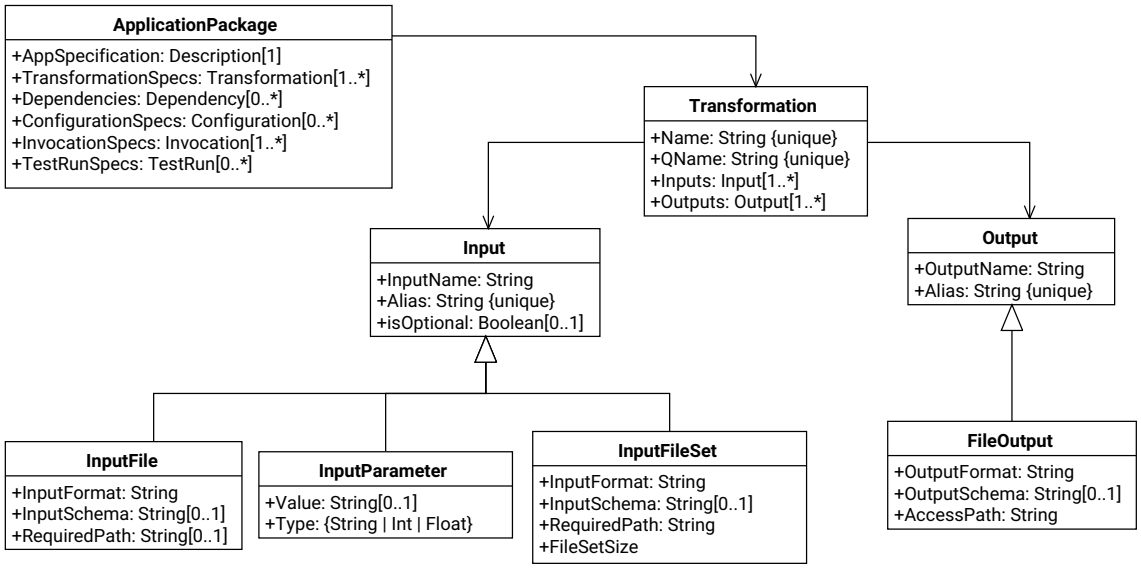


**Figure 3.6:** The constituents of a transformation specification

The property *Name* contains a unique string (only in the scope of current application) which can be used to allow addressing the distinct transformations. Another possible way to identify a certain type of transformation is to introduce an abstract functional description similar to a "PortType" in WSDL. The property *QName* is a unique string allowing to link a particular application's transformation to an abstract description used in the choreography modeling with the TraDE concepts applied. The publisher of the application and the choreography modeler have to agree on the list of such abstract descriptions in advance. As a result, searching an application can be simplified to looking directly for an identical QName. A generic *Input* entity can be described by three properties: its name, an alias, and whether it is optional or not. The name of the input is a string property which is necessary for improving the readability of the specification, especially in GUI-based interaction scenarios. Additionally, it can be used for renaming the input using a publisher-defined value in case an application supports only predefined input names. An alias is a unique string identifier of the input which can be used for ordering multiple input parameters in the invocation command. Each alias has to be unique only within the current application's transformation specification, meaning that the same aliases can be used for different transformations. Another input property is a Boolean value specifying whether the input parameter is optional. For instance, some transformations might allow specifying optional parameters like the output resolution in case of an image conversion. Thus, specifying that a parameter is optional signals that the transformation can proceed even if this parameter is not specified. For simplicity, only optional inputs can be supplied with this parameter and if it is omitted then the input is considered to be mandatory. An excerpt from the

model shown in Figure 3.6 illustrates three types of input: an *InputParameter*, an *InputFile*, and an *InputFileSet*. The former describes a value of a simple type, e.g. numeric or string, needed for the transformation to take place. For example, a specification of some numeric coefficient. This input type is described by an optional value property. In case of its presence, the input parameter is basically set by default at specification time and thus can be directly used during the invocation. However, if the value is not set then the parameter has to be provided in the transformation request. Additionally, parameters can be of different type, hence the types have to be specified in order to simplify processing of the values. One of three options can be chosen: a string, an integer, or a float parameter.

As was described previously, the usage of files as the unit of interchange is quite common in scientific workflows, although other methods are possible. Assuming the interaction model with the standalone proxy provider role, all input or output data can be collected as intermediary files for further processing, which allows us to focus mainly on files. The *InputFile* is described using three properties: *InputFormat*, *InputSchema*, and *RequiredPath*. In simplest cases the data transformation takes one input file, e.g. a plain text file, and produces one output file, e.g. a PDF file. In more complex cases, a data transformation might combine multiple input files of different formats into one or many possibly different output files. In both cases, an input format defines the file's specifics and serves as a validation marker describing which input files are accepted. *InputFormat* property is a string which describes the input file's format. It can contain a regular file extension like PDF or PNG. Another option is to store a Multipurpose Internet Mail Extensions (MIME) type, e.g. "text/plain". The latter option describes custom formats in a more understandable fashion as it can contain a hint on type of data. This can be helpful in case several custom formats have the same extension, but describe different types of data. Another property which characterizes the input is its schema. Two files of the same format may not necessarily be supported by the same data transformation application. For instance, we can consider two versus three column Comma-Separated Values (CSV) text files and a data transformation application which can only work with a two column CSV. In such case, the application can simply fail due to the wrong file format. The file's schema is usually stored in a separate file of specific format depending on the input file's format, e.g. XML Schema for XML, or JSON Schema for JSON. The *InputSchema* property in the model is a string defining the path where the file's schema is stored. The *RequiredPath* property is optional and describes the path where an input file has to be stored in case an application requires a static path value. For example, an application might look for a file located in the same directory where the application's files are stored. Likewise, the output files are described by their format, schema, and path properties. The path describes a location where the resulting output file is going to be saved.

In some cases, multiple input files of the same type are expected by the application. In Section 1.2.6 we discussed an example of simulation involving the transformation of multiple snapshots into a video. Technically, all snapshots are of the same type and used by the application implicitly, meaning that the invocation command does not list the files which are going to be used for the transformation. Instead, an input parameter defines how many files the application needs to read from its root folder. Such dynamic nature of input

requires special modeling. *InputFileSet* is a type of input describing a set of files having the same type. It has common properties with the *InputFile*, but in this case we assume that such set is used only implicitly, thus a *RequiredPath* property is mandatory. The set's size can be described by *FileSetSize* property which can be either a number or a reference to an alias. In the former case the application's input size is static and a numeric value is used to describe the set's size. The latter is related to cases when the amount of input files is defined using an input parameter as with the video transformation example. In such case an input parameter's alias could be used to identify the size of the *InputFileSet*.

Commonly, the applications can be executed only if their dependencies are available in the environment. A self-contained application package must have all the necessary dependencies specified. This problem is referred to as dependency hell by Boettiger [Boe15]. In some specific cases, software might require some hardware dependencies fulfilled, e.g. a certain amount of processing power or memory has to be available. In this work we focus on software-related dependencies which can be of multiple types. The model shown in Figure 3.5 allows extending it by adding other types of dependencies required for a specific use case. The excerpt from the model shown in Figure 3.7 demonstrates the *Dependency* entity containing software-related dependencies required for the invocation of an application.

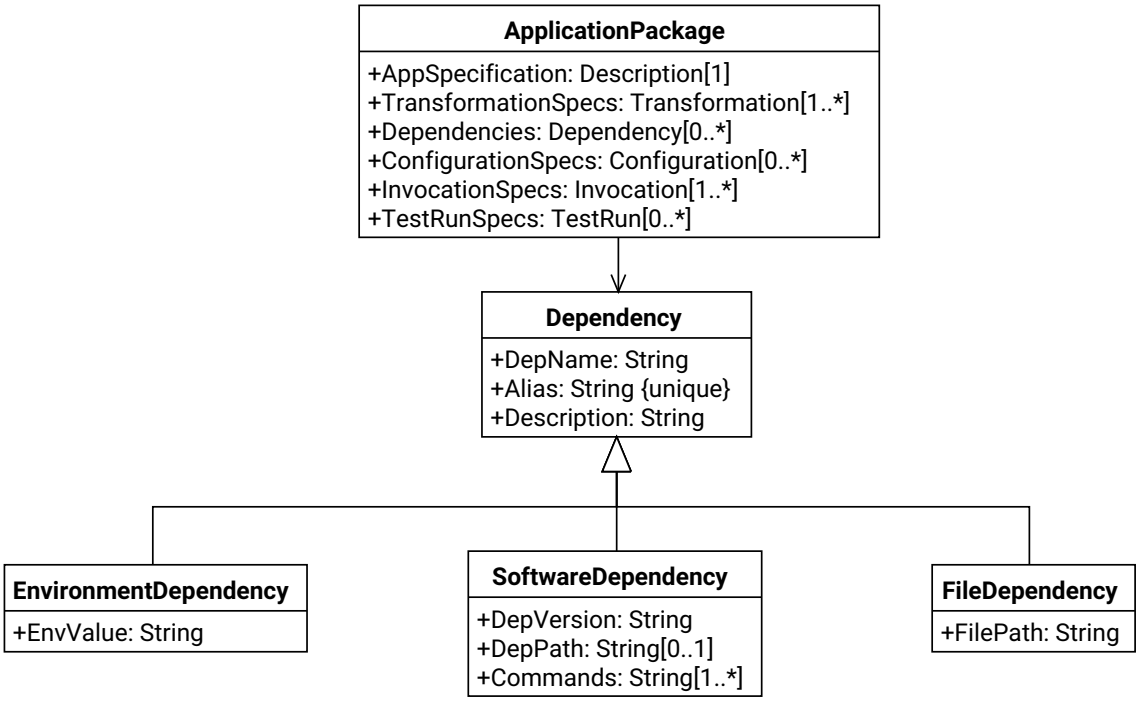


**Figure 3.7:** The constituents of a dependency specification

A generic dependency entity is described by three properties, namely *DepName*, *Alias*, and *Description*. The *DepName* is a string identifier containing the dependency's name which can be used for readability and identification purposes. The property called *Alias* is a unique identifier which can be used in installation or configuration commands related to

this dependency. For example, a dependency might need to be configured for the proper invocation of the application. An alias needs only to be unique within the boundaries of the current application's specification. A textual description is an additional information about the dependency which can be used for a GUI-based interaction with the registry. We distinguish three types of dependencies which extend the generic dependency entity: *EnvironmentDependency*, *SoftwareDependency*, and *FileDependency*. In some cases, an application might depend on a specific environment variable to be set in the operating system. One example is the presence of Java's path variable in case the invocation relies on *java* alias in the command line interface. As environment variables are key-value pairs, this type of dependency has to have a name and a value. For this purpose, the *EnvValue* property contains the value and the *DepName* property inherited from the Dependency entity contains the name of the environment variable. The next dependency type in the conceptual model is called *SoftwareDependency*. Basically, it describes any kind of third party software which is required by the data transformation application in order to run. A software dependency is described by a version, a path, and a set of commands required for its installation. An application might only work with a particular version of a software, thus the specification of a version number should be provided. This is also useful for providing the information about the dependencies in GUI-based interaction with packaged applications. A publisher might decide to provide the dependency's files for installation. A path property describes where the corresponding dependency's files are located. Furthermore, the installation commands have to be documented as a set of strings which makes the dependency specification complete. However, the software installation process is defined by multiple various aspects, e.g. operating system, package managers, network accessibility. Thus, it is difficult to provide installation commands which will suffice for every single provisioning method. For instance, the commands which will work simultaneously on Windows and Linux distributions, or even Linux distributions with the different package managers. As a consequence, this requires some assumptions on the implementation level regarding the supported format of the commands. Another dependency type is *FileDependency*. In some cases, a software might depend on some external files which need to be explicitly modeled not as a part of application's files. For instance, some patched file might be modeled explicitly to highlight the fact that an application is updated. Another example is documentation or localization files. Additionally, a software dependency might itself depend on a file, e.g. an installation script. Thus, a software dependency's command property can be specified using the alias of an installation script modeled as a file dependency.

Apart from its dependencies, an application might require additional configurations prior to invocation. For instance, file dependencies have to be copied into another location or installed software dependencies need to be configured. A *Configuration* entity shown in Figure 3.8 contains the specifications of additional configurations to be made which are needed for application prior to invocation. A configuration is defined by two properties: a name, and a set of required commands. As was discussed previously, command's format has to be specified more precisely on the implementation level.

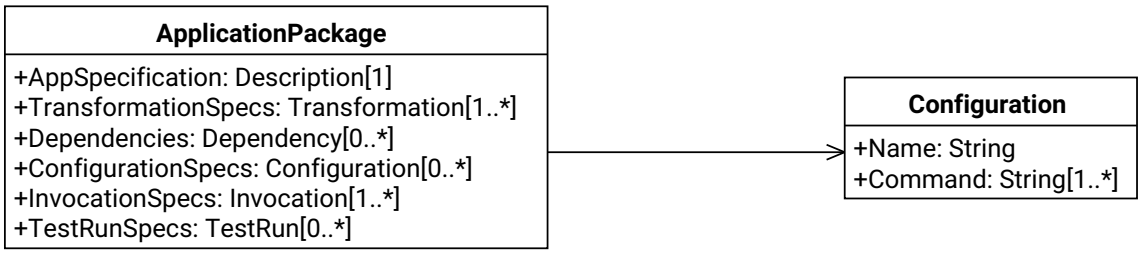


**Figure 3.8:** The configuration specification

As a next step in the application packaging specification, the ways of how the application can be invoked have to be defined. The *Invocation* entity shown in Figure 3.9 contains available invocation methods for the packaged application. Provisioning types discussed

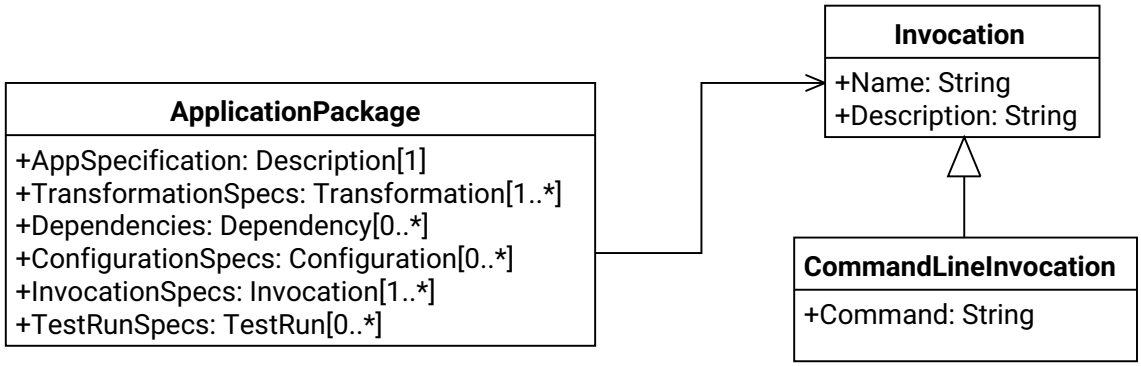


**Figure 3.9:** The invocation specification

in Section 3.2 determine how an application can be invoked. Commonly, applications support CLI-based invocation. Considering only the data transformation applications and assuming that no user interaction is needed, we assume that a data transformation application provisioned alongside its files, dependencies, and other related specifications, can be invoked using the command line interface. In case of web service invocation, the command line utility such as curl [cur17] can still be used. The results of invocations have to be obtained differently depending on the application or web service's response specification. A packaged application might support multiple invocation methods, e.g. a web service offering both REST and WSDL interfaces. Thus, one or more invocation specifications can be provided by the publisher. The generic invocation entity has a name and a description as its properties. Both properties are for readability of the specification and can be used for GUI-based interaction with the registry. A command line invocation is an extension of the generic invocation entity which is described by an invocation command. A command which invokes an application includes the input parameters usually ordered in a predefined way. Input specifications in the model are provided with the alias property which can be used for ordering of input in the invocation command specification. For example, an application has three input parameters with aliases $input1, $input2, and $input3. Then, the invocation command can be specified as "java application $input2 $input1 $input3". On the conceptual level, the model does not enforce checks of how many

input parameters are specified and whether the number is equivalent to the number of aliases used in the input mapping string. However, these checks can be introduced in the implementation part specifically at the time when an application is going to be published. This rises the questions of how we can verify the "correctness" of a packaged application and what is a correctly packaged data transformation application.

There are several aspects which can be checked in order to verify the correctness of a data transformation application. First of all, it is useful to make sure if the provided transformation can be invoked and behaves as intended by its publisher. With a black-box view on the application, the checks related to source code are not the option. The conceptual model provides means to make a published application verifiable. The *TestRun* entity shown in Figure 3.10 describes the specification of a sample application's run.

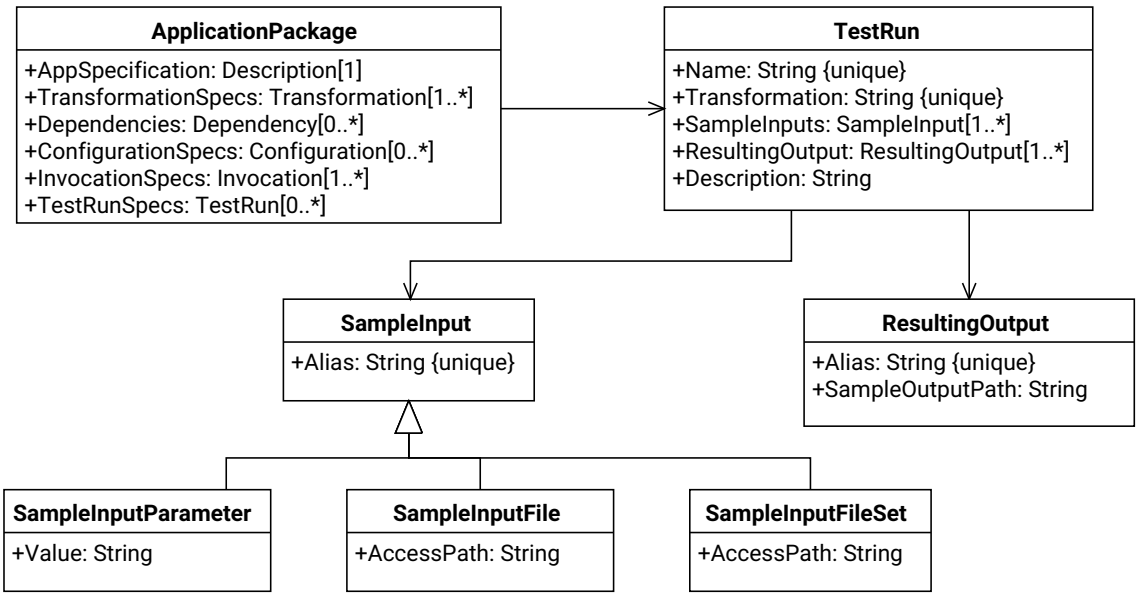


**Figure 3.10:** The test run specification

The main idea is to provide a set of predefined test inputs and corresponding canonical outputs related to a particular transformation supported by the application. Essentially, such predefined sets represent instances of successful invocations. In other words, running the transformation using a test input should produce a specified output. Before the application is published, it can be invoked using the provided test inputs. The resulting outputs are compared with the test outputs of the corresponding TestRun specification. If the results are identical then an application can be considered correct. Each TestRun specification is described by a unique name, related transformation's name, sets of sample inputs and resulting outputs, and the description of the TestRun which can be used for providing an additional information to the consumer. Specifications of both inputs and outputs have to include aliases used in the actual transformation's specification. This allows correlating sample inputs in order to properly invoke the application and to correctly identify the outputs. In fact, having the input and output examples help users to better understand the data transformation application by analyzing them in conjunction with the other available

information. However, this is a very simplistic view on the application correctness, e.g. the presence of bugs in the code are not considered due to the black-box view on the application. Moreover, from the security point of view, blind trust in a user-submitted application is a serious risk as malicious code can be published.

### 3.3.2 Packaging the data transformation applications

The conceptual model discussed above allows creating a standalone package of a data transformation application. However, the idea of publishing the application files and dependencies in a standardized fashion implies having a precise description of how files can be accessed. Figure 3.11 demonstrates an example of a standardized application package's structure. For instance, if we assume that the application's dependencies are always stored

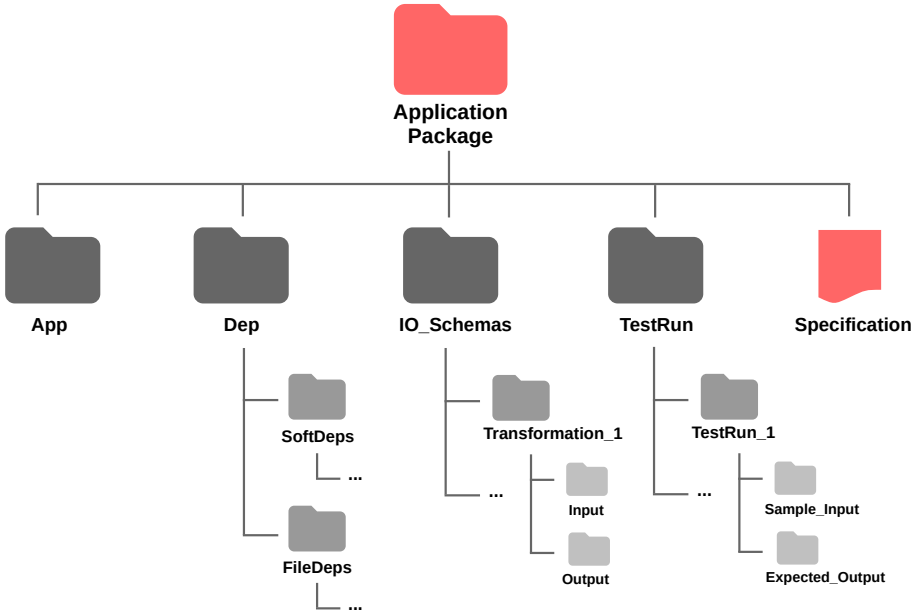


**Figure 3.11:** An example structure of an application package

in the *dep* folder, then the path for every dependency only needs to include the actual dependency-related information and the parent folders can be ignored. Similarly, the input and output schemas, application files and other related information can be accessed using the predefined package structure. The presented conceptual model has to be materialized as a *Application Package Specification* file of some format and stored together with the other artifacts. Conversely, if the package's structure is not standardized this introduces additional clutter in the registry.

Especially with respect to several path properties discussed before it makes sense to focus more on potential ways of accessing the files. One important point related to publishing the files is whether it is acceptable to use links to remote files or not. There are two stages at which this question is meaningful: *before* and *after* the application is published. In the former case it is a suitable scenario as it gives more freedom to publishers. A publisher

might simply use the link to a public file sharing service instead of downloading the actual files and placing them in the package's folders. However, when the application is already published it is not safe to use links as remote files might change, e.g. a new version was released, or become inaccessible. A published application represents a certain state with a specific version, dependencies, and invocation details. This information is fixed for a package and with new versions the application should either be updated or published as another version. If the links represent remote files then all changes to these files become transparent for a consumer and this might lead to an improper and inconsistent data transformation process. Having this considerations in mind, it is preferable if the application package will contain the actual files which are listed in the specification. As a result, two different states of the packaged application representations can be distinguished: one that contains both, file paths and remote links, and another which only contains file paths. The former reflects pre-published state and the latter represents the application package after it was published. The "materialization" of the remote links has to be done in-between these states. This step can also be considered as a part of testing an application for correctness: in case some of the specified remote files are not available it makes no sense to publish an application and the process can be stopped. After the remote files are successfully accessed they can be copied and packaged. Then, the representation of the application model needs to be changed in order to reflect the local paths instead of remote links. The corresponding paths in such case must either explicitly reflect the package structure or the package structure has to be standardized, e.g. as shown in Figure 3.11.

## 3.4 Handling Data Transformation Applications: A Generic Framework

When introducing the interaction model with a proxy provider role in Section 3.2, we discussed why shifting many responsibilities to the consumer makes the reuse process very consumer-specific. On the contrary, we are interested in consumer-independent methods of reuse. One consequence of such view on the role of a consumer is a simplified process of integration with the TraDE Middleware which, in this case, can be considered just as another consumer type. Although the consumer role is the main beneficiary of the application reuse process, the key responsibilities fall on the publishing, storing, and provisioning roles. A publisher is expected to provide a correctly-formed application package. After it is published no further participation from the publisher is required. The storing and invocation-related functionalities thus have to be handled somewhere in-between the publisher and consumer. Basically, the interaction model with a proxy provider and the registry located in the hub zone described in Section 3.2 covers these requirements. Figure 3.12 illustrates a generic layered architecture for reuse of the data transformation logic which consists of several parts: (i.) a user interaction layer; (ii.) a security layer; (iii.) a request router; (iv.) a repository; (v.) a task manager; (vi.) and a provisioning layer. The two main components, namely a *repository* and a *task manager* are located on the bottom level of the architecture. The former is responsible for publishing and lookup of the applications. The latter handles the transformation requests depending on how an application was published. In a scenario

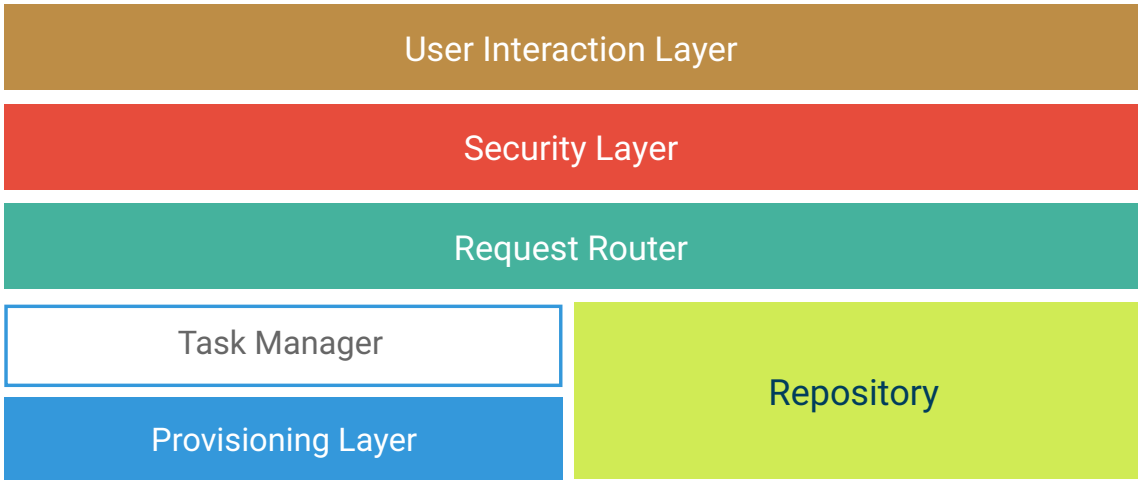


**Figure 3.12:** A generic layered architecture for data transformation applications reuse

where an application is published as a self-contained package a provisioning layer is needed for handling the invocation and returning the results back. If the application was published as a reference to a web service then the task manager is responsible for the mediation of the request and response. The three top layers provide means for a secure user interaction with the repository and task manager. Starting from the bottom, each part of the architecture will be described in more details in the next sections leading to a more detailed architecture to be derived in the end.

## 3.5 Repository

### 3.5.1 Publishing Scenarios

The main purpose of a repository component in the architecture is to allow storing and accessing provisioning-ready data transformation applications. An application is considered to be ready for provisioning when it is successfully published which means that at least an application package has to be generated and at most it has to be tested and provisioned using a provisioning technology like container virtualization, e.g. Docker. Important point here is that a package provided by the publisher differs from a provisioning-ready package that has to be generated by means of the repository. The publisher is only required to provide an application package conforming to the structure shown in Figure 3.11 which consists of application-related files and an application package specification file. The latter is a technology-agnostic description of the application package based on the conceptual model presented in Section 3.3. Storing an application package without relying on specific provisioning technology allows reusing the same package for multiple provisioning engines of choice. Although this format neutrality adds an implementation complexity of performing the conversion into a target format supported by the engine of choice, the publisher is not required to provide technology-specific artifacts in advance. This approach simplifies

the publishing routine for scientists who are not familiar with implemented provisioning technology.

The process of publishing might include a combination of several steps including the process of downloading remote files, generating the *Provisioning Specification* meta-description file supported by an engine of choice, e.g. a Dockerfile, testing the package using this generated specification. Several listed steps already rely on actual provisioning which leads to a question of whether the repository needs to support the provisioning-related functionalities. To better understand the connection between provisioning and actual consuming of the data transformation logic we will start from the publishing process. More specifically, there are several scenarios of how an application can be published:

**Publishing only** In this case, a package has to be generated and stored without any further steps. However, in case an application packaged together with its files and dependencies, the package generation itself requires some modifications to original publisher's package to be made. Firstly, the remote files have to be downloaded and stored inside the corresponding folders conforming to the directory structure described in Section 3.3 and shown in Figure 3.11. Next, the original application package specification has to be edited in order to reflect the changes in file paths. Note, that it is beneficial to store both versions of application package specification: the original which was provided by the publisher and the modified one. In such scenario, more information about the package is available which can be potentially useful for the analysis and optimization of the packages. Furthermore, for the package to become provisioning-ready it has to possess a provisioning specification in a format which is used by the corresponding provisioning engine, e.g. a Dockerfile. The generation of this specification must happen as a part of the publishing procedure. The resulting package is portable and can be used by any authorized third party which supports the provisioning engine of choice. Potentially, multiple provisioning engines can be used implying that several provisioning specifications can be generated. A package then needs to be stored using the storage implementation of choice. Publishing an application as a remote web service requires much less overhead, although the test specifications can be provided together with the application package specification for verification purposes.

**Publishing with verification** Next option is to publish an application and verify its correctness using provided test run specifications. The publishing part is similar to the previous option, whereas the verification part involves some additional steps. After the provisioning-ready package is generated, the repository has to invoke the application using the provided test specifications. However, this task requires support of the provisioning engine of choice if invocation must happen on the side of the repository. While a combination of the repository with actual provisioning is possible, it makes the implementation heavier and more monolithic. Additionally, it makes the repository technology-dependent and in case several provisioning technologies are planned to be used it is unclear if the repository needs to implement all of them or only a specific subset. The task manager component with the help of provisioning layer is responsible for handling the provisioning-related tasks. In fact, the repository

can have a dedicated task manager only responsible for verification purposes. In order to run the test specification, a portable application package has to be "deployed" on the provisioning engine of choice. Thus, a request for running the application has to be sent from the repository to the responsible component, i.e. the task manager. After the application is successfully invoked and the results are obtained from the task manager, the comparison must happen. In case, the results are identical to the provided output in the test specification, then an application is marked as verified. This ensures the correctness and might be used in consumer's decision process in case several compatible data transformation applications are found and one has to be chosen. Note, that actual provisioning already happened, e.g. a Docker image was built and is available for the task manager. At this point, there are two options: to keep the built application image (or, e.g. a virtual machine) or delete it, as it was meant only for testing purposes. We will delete the image by default after the test invocation takes place and results are returned.

**Publishing with provisioning** While publishing is similar to the first option, the testing does not take place in this scenario. Instead, an application is provisioned using one (or potentially several) task managers relying on their own provisioning layers. In order to reflect that an application was provisioned, the repository has to maintain this information. After the provisioning took place, an information about the corresponding task manager has to be added to the respective package's information. For instance, if a database is used, then a dedicated table or field has to store the related data. This simplifies addressing the corresponding task managers when the search results for data transformation application are returned to the consumer.

**Publishing with verification and provisioning** In this scenario, an application is published, verified, and provisioned. Basically, this is a combination of all possible options. However, one additional issue that have to be considered in case an application's verification was not successful is whether an application has to be provisioned or not. Technically speaking, at the moment when verification fails the application still remains provisioned as the corresponding image was not deleted after the transformation result was returned back to the repository. As a consequence, if provisioning has to be undone another request from the repository has to be made.

### 3.5.2 Generation of a Provisioning-ready Package

After the publisher chooses the most suitable publishing scenario and provides the packaged data transformation application, the generation of the provisioning-ready package must take place. As discussed previously, "materialization" of the remote links has to be completed at first. In order to separate concerns, a sub-component of the repository called *package generator* is responsible for performing the generation of a provisioning-ready application package. The original application package specification must be updated in order to reflect the changes in file paths. This updated version of the specification has to be included inside the package as the main specification required for creation of a provisioning specification. While various provisioning technologies can be used, e.g. different container engines or

virtual machines, it is difficult to support all possible options. Firstly, the conversion of an application package specification has to be done to every supported format. Secondly, all resulting provisioning specifications need to be packaged alongside other artifacts. Although such difficulties are possible to overcome from the technical point of view, the overall process becomes more time-consuming. For this reason, we assume that the repository supports some default provisioning technology. This assumption allows focusing on specific provisioning specification format but does not limit the usage of other chosen technologies. As the updated application package specification is included into the package, the conversion can happen on the corresponding task manager's side. Another option is to allow publishers to choose which provisioning specifications have to be generated. The contents of a provisioning-ready package are shown in Figure 3.13. On one hand,

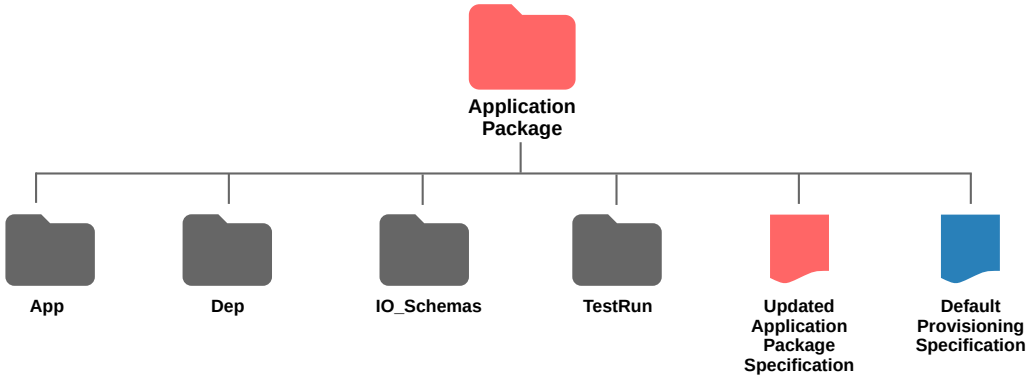


**Figure 3.13:** Provisioning-ready application package

the original application package specification, i.e. with a combination of file paths and remote links, could also be added into the provisioning-ready package. On the other hand, its presence there is redundant as it will not be used for conversion into a provisioning specification format. It can only be used for provenance reasons, e.g. analyze remote file locations, compare if the versions have changed. Basically, we can store the original application package specification separately without any consequences, e.g. as a separate file next to the provisioning-ready package or in the database together with the information required for searching the data transformation application.

### 3.5.3 Storage

After the provisioning-ready package is generated, it has to be stored in the repository. This implies that a storage layer is available in the repository component. We focus on following open issues:

- selection of the storage type to be used;
- which information has to be stored;
- how to avoid publishing duplicates.

Regarding the first question, a database technology is an obvious answer. However, storing the application files in a database might be cumbersome. This applies to the cases when an application is published as a self-contained package with all the related files and dependencies. In case an application is available only as a remote service then the most common scenario would be a publishing of a single application package specification without any other artifacts. On the other hand, for a remote web service it is still possible to provide test run specifications. The study by Sears et al. [SVIG07] indicates that for storing files larger than 1MB it is preferable to use the file system instead of a database. One of the most influencing factors is the fragmentation of a file system or a database. As file systems are designed specifically to deal with files, they show better performance over time as a file system gets more and more fragmented. Assuming that files are stored on the file system, there is a need to maintain file paths information as well as other specification-related details. This is a lightweight textual information which can be stored in a database. Depending on the choice of implementation this can be a RDBMS or a NoSQL system, e.g. a document or a column store. As a result, provisioning-ready application packages are going to be stored using a file system, and the specification-related information will use a database.

In fact, the files belonging to the provisioning-ready application package can be used only when they are together, hence an archive can be created during the generation and placed in an application-specific directory as a way to optimize the usage of storage space. A database needs to store the path to this archive on the file system. However, it is not sufficient to store only the path information. In order to implement the search, the database needs to contain specification-related information. Basically, the whole modified application package specification or its parts can be copied to the database. For instance, the general information and dependency names can be used to find an application. Additionally, the information about task managers where a data transformation application was provisioned has to be stored in the database.

An application-specific directory has to be unique for every published application. Its name has to be a unique identifier generated at publishing time. In theory, this could be any abstract numeric identifier of sufficient length. However, this approach makes it easier to publish duplicates as an identifier does not correlate with the application in any way. Taking a set of variables, a skolem function [EE01] returns a unique value. However, the task of finding a suitable set of variables which can be used for the generation of the unique identifier is not a trivial task. The main obstacle is the absence of guarantees that some information in the specification will not be omitted by the publisher. Hence, the variables which form the unique identifier have to be made mandatory in the application package specification. In theory, an application might be published by different publishers in case of large and distributed projects. Thus, the publisher information is not a reliable variable even in combination with other variables. However, similar reasoning can be applied with regard to the developers information as well. People might leave and join the projects, making the list of developers not constant. Both publishers and developers can be used for analyzing the potential duplicates as well as generation of a unique identifier, but they do not eliminate the problem of duplicates. As a starting point, the publisher information will be used in

combination with other variables to form an identifier. The reasoning behind this is that the publisher information is always present and easier to check. Multiple applications can have the same name which makes this information useful only in conjunction with additional variables. Another variable we can add to the identifier is the application's version. In this case different versions will have unique identifiers and if the version information will be used at the end of an identifier, the packages related to the same applications of different version will be located close to each other in the file system. For example, if user *John Doe* publishes a version *1.0* of the application called *Image Transform*, then the generated identifier might look like *johndoe:imagetransform:1.0*. Note, that the actual format of how identifier can be concatenated is implementation-specific and could look differently, e.g. it could be a standard directory path *johndoe/imagetransform/1.0* meaning that folders will be nested in this order. Then, if the same user publishes the version 2.0 of the same application, the generated identifier will look like *johndoe:imagetransform:2.0*.

Using the *(publisher, name, version)* combination in order to generate a unique identifier for an application it is possible to improve the chances of finding a duplicate during publishing, but does not guarantee that duplicates will not be present. Moreover, it still gives no guarantee about the uniqueness of an identifier as it is possible that the same publisher will publish two applications accidentally having the same name and version, but performing different transformations which will result in generating the same identifiers for both of them. One way to make it more precise is to also include the information about developers. The chances that two different applications have the same names, versions, developers and published by the same publisher are low. Moreover, as the applications are published by the same person, the ambiguous cases when the same identifiers are generated for different applications can be refined at publishing time. In order to concatenate developers they have to be sorted beforehand and some string optimization techniques might be applied, e.g. using only the first and the last letter from names and surnames. According to a study by Tagliacozzo et al. [TKR70] the error rate for first and last letters in names is typically lower than for in-between letters. This can help to tackle misspelled names, although the possibility of mistakes is not eliminated. Additionally, using only parts of the names can help reducing the overall length of the identifier which can grow big in case of multiple developers with long names. Following the previous example, if the developers names are John Smith and John Doe while the rest information is the same, the generated identifier might look like *johndoe:jndejnsh:imagetransform:2.0* if the developers were sorted by their surnames and only the first and the last letters from the name and surname were used. The discussed notion of separation between the database and file system parts for some application can be visualized as shown in Figure 3.14. Figure 3.14 lists following details which needs to be stored in the database: the information about the application, provisioning, and verification as well as the file system path linking with the provisioning-ready package. The data stored in the database is meant to provide means for searching and invoking the application and can be obtained from the application package specification provided by the publisher. Basically, the conceptual model of application's description stored in the database needs to reflect the model described in Section 3.3. However, it can be simplified as many technical details can be omitted or shortened. Additional data can be derived from the specification to provide a consumer with statistics about the application,

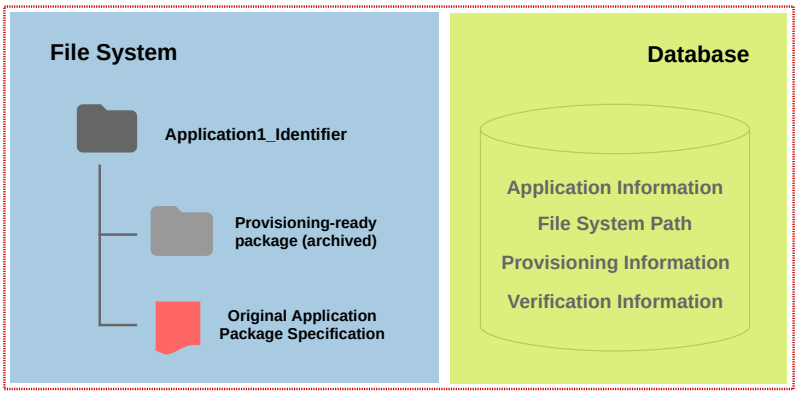


**Figure 3.14:** Storing the published data transformation application

e.g. by summarizing the total number of dependencies. On the other hand, some parts of the application package specification have to be optimized for search implementation. One example is a set of transformations supported by the data transformation application.

### 3.5.4 Search

Assuming that multiple applications were successfully published, the repository now must provide means to search for those applications which are compatible with the consumer's request. Depending on the level of details there are several ways to check the application's compatibility. Although other input types such as input parameters are possible, the input files play the most important role in defining the transformation. In a simple case where an application only takes one input file and produces one output file it may look as simple as checking the compatibility of an input/output format pair. For instance, if an application transforms the data from a .txt to .xml and a consumer requests a transformation providing matching information about the input and output formats then a transformation might be considered as compatible. However, on the more detailed level one can argue that if the schema of the provided input does not match the schema supported by the data transformation application then compatibility can not be assured. For example, if an application requires a 4-column CSV text file as its input and the user provides only a 2-column CSV how the application will handle the missing details is implementation-specific. The behavior of such data transformation is non-deterministic as it may succeed, fail or produce an incorrect output. In case both, a consumer and a publisher provide input schemas, the comparison can happen to verify an application's suitability for the request. However, this is not always achievable as providing an input schema is not obligatory.

With this considerations in mind it makes sense to differentiate between a search *without* and *with* schema verification. The former relies only on comparison of the information provided in application package specification. This is the simplest way of looking for a suitable application and is applicable if an application does not impose specific requirements on input schemas. However, as described previously, it might introduce incorrect application's behavior and the error handling therefore should be implemented on the task

manager's side. A request for an application without verifying the schema's compatibility can still contain various details about the potential application. The main prerequisite for suitability is the equivalence of the input and output file formats in the request itself and the application's meta-data. When specified, input and output files have a format description field. Note that we still consider the simplest example with one input file and one output file. In such case, the transformation can be described by a tuple *(input file format, output file format)*, e.g. (txt, pdf). This information is crucial for checking the application's basic suitability by comparing it with the same tuple derived from the request. Such comparison is similar to signature matching described by Zaremski et al. [ZW95] and allows limiting the number of applications for further refinement of the query. The signature of the application in this case is a combination of its input and output file formats. Other types of input or output are not yet considered in order to simplify a primary compatibility check. Additionally, the search can be refined using other information from the conceptual model, e.g. tags, using keyword-based retrieval techniques [BK16]. As a result, the response's precision will be higher but every returned application will not be checked for actual compatibility.

In a scenario with multiple input and output files, a tuple *(input file format, output file format)* needs to be modified. Both input and output parts of the tuple now become tuples themselves: *((input file format-1, ..., input file format-n), (output file format-1, ..., output file format-n))*. One pitfall, however, is an ordering of the inputs and outputs in the tuples. On one hand, if only the exact match will be allowed, then potentially suitable applications will be rejected due to the different ordering. On the other hand, if an exact match is not enforced then the task manager has to deal with the mappings between the inputs/outputs supplied in the request and the actual applications' inputs/outputs. For example, a signature derived from the request for the transformation of two input files (txt and dat formats) into two output files (txt and pdf format) might look as follows: *((txt, dat), (txt, pdf))*. If there is an application with specifications in different order, e.g. *((dat, txt), (pdf, txt))*, the exact matching will result in empty set. In contrast, if the order is ignored then one suitable application is returned in response to this transformation request. At this point, if the application will be invoked, the order of inputs has to be adjusted by the task manager as well as the order of outputs has to be restored for the correct response. Moreover, if input/output contains several files of the same type, the mapping between the request and the application requires special handling. For instance, techniques like probabilistic mapping or invocations of the application for every possible order can take place. These invocation strategies might be chosen based on the total number of similar input/output formats or consumer-provided constraints, e.g. the allowed number of invocations. An assumption that a consumer can deal with the ordering of outputs makes the system less reliable in case of automatic invocation, hence similar strategies have to be applied before a response is sent back to the consumer.

The search with schema verification is only possible if input schemas are specified for the application. Moreover, the schema comparison is not a trivial task due to potential difference in schema formats. For instance, the schema in a request might be provided in XML, whereas the application's input schema is a JSON schema. For such cases schemas

have to be transformed into a common format for further analysis, e.g. building a parse tree and comparing the nodes. A naive way to compare schemas of the same format is to check for identity, i.e. schemas have to be equal. This can be helpful for checking the simple comma-separated formats. However, such view on the problem does not take into consideration that the same structures can be characterized by different schemas, e.g. using substitutable data types. In general, schema comparison is time-consuming and has to be explicitly listed in consumer's request as an additional check. Moreover, the system has to implement the schema comparison mechanism to support certain types of schema definition languages, e.g. XML Schema or JSON Schema. Therefore, schema comparison is out of scope in the context of this work.

In fact, comparison of the application's signature with the signature derived from the request better ensures compatibility if other types of input are added into the signature. Considering the multiple input/output scenario and the ordering issue discussed previously, the equal number of the input and output types in a request and application's specification can be used to decide if an application is suitable for response. As was discussed before, for both input and output files, the equal amount of corresponding formats is the prerequisite for compatibility. If input parameters are added to the signature, some applications having an equal number of input and output files, but lacking or having more parameters than the request can be considered less suitable and ranked accordingly in the response. However, the ordering problem which was previously discussed arises for other input types as well.

The signature comparison allows forming the preliminary candidates list which can be further refined based on the information available in the consumer's request. Thus, it is beneficial to support as much information from the application package's conceptual model in the request as possible. The search using tags is one of the options from the general information about the application. However, the search through an uncontrolled vocabulary might result in imprecise results. On the other hand, limiting a publisher's choices for describing an application might lead to potentially big number of results returned. Not only the general information such as tags, developer or publisher might help for the desired application to be found but also certain dependency information in case the consumer knows exactly which application is needed. The structures of request and response were not discussed yet and will be conceptualized in the next sections.

One significant detail that has to be considered is that an application might support multiple different transformations having independent signatures. This implies that the same application can be returned in response to different consumer's requests. For this reason the signature strings can be generated for each supported transformation and used for comparison. Moreover, resulting signatures have to be associated with an application independently of each other. Basically, each transformation has to be considered as a separate application but in the conceptual model of the information stored in the database they belong to the same application.

### 3.5.5 Searching for Composite Data Transformations

Apart from the simple search which directly returns matching applications it is also possible to search for composite data transformations involving multiple piped applications. Basically, this task is similar to the function realization problem described by Mili et al. [MMK+94]. In our context every data transformation application is being treated as a black-box meaning that the composition can happen based on the analysis of inputs and outputs. As a starting point, we consider a trivial composition technique shown in Figure 3.15. It is similar to one of the examples from Mili et al. [MMK+94] discussing how software components can be joined in order to get a desired output. In our case the data transformation applications can be "chained" in such a way that the desired output is produced. The overall idea is to check for compatibility of a requested input, intermediate outputs and inputs in order to derive a requested output. As a first step, the requested transformation's input is checked for compatibility with available applications' inputs, then the resulting outputs need to be checked with the remaining application's inputs and so on. The complete algorithm allowing to find trivial software component compositions is mentioned in Mili et al. [MMK+94]. We informally discuss how it can be applied for chaining the data transformation applications. Firstly, a directed graph has to be constructed in order to reflect the connectivity information. For this purpose, for every available application we need to compare the compatibility of its inputs with the outputs of other available applications and vice versa. If an output is compatible with an input then the outgoing edge is created, whereas the incoming edge describes the opposite case. Considering that one application might support several data transformations the graph construction might take longer. Then a breadth-first search can be applied in order to find the shortest path from the application compatible with the requested input to the application producing the requested output. The overall procedure is costly performance-wise and a couple of issues can be checked prior to searching for a composition. One important point is to check if there is no application producing the requested output then the search for a trivial composition will always return an empty set. Additionally, if there is no application consuming the requested input the search for a composition cannot be started either.

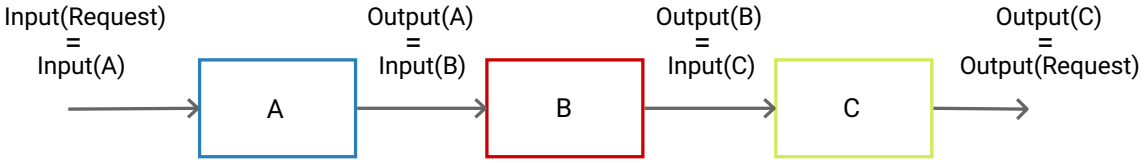


**Figure 3.15:** A trivial composition example based on [MMK+94]

However, this rather simplistic view on composition does not cover all possibilities. For instance, Figure 3.16 demonstrates an example inspired by discussions of component composition in Mili et al. [MMK+94]. It shows how at first the requested input is split into two parts and consumed by two distinct data transformation applications, then the resulting outputs are combined in order to be consumed by another application which eventually produces the requested output. In general, we can say that the lack of applications consuming the requested input or producing the requested output does not necessarily

result in an empty set of possible compositions as inputs or outputs can be split. With this approach, various outputs could be combined in order to produce the requested output leading to a bigger amount of available composition options. Furthermore, this leads to optimization problems like minimizing the number of redundant intermediate transformations. The algorithm for searching component compositions in the work of Mili et al. [MMK+94] is proven to be NP-complete. The problem of application composition is not in the focus of this thesis. Instead, we briefly discuss how a composition of applications can be invoked and which issues might arise before.

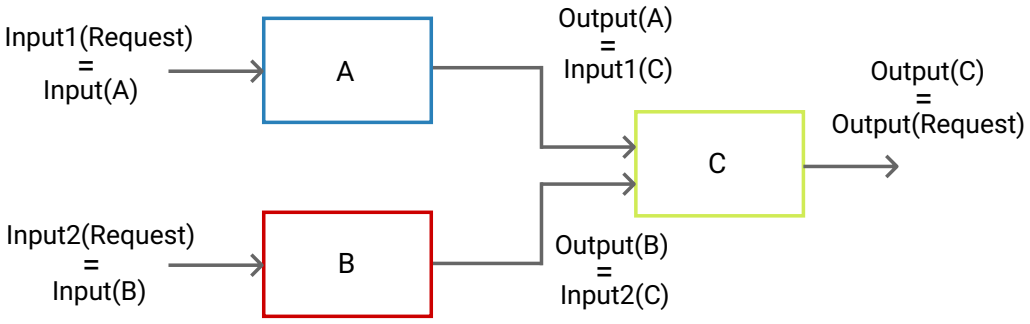


**Figure 3.16:** A composition involving input splitting inspired by [MMK+94]

As with the simple search example, relying only on the compatibility of input and output types is not sufficient for declaring applications compatible with the request due to possible schema incompatibility issues. However, the overall process of checking a composition becomes very complex if schema comparison is enforced. Moreover, if several application compositions are available the process becomes even more complex due to necessity of comparing the schemas for every case. Thus, it makes sense to leave the errors handling to the invocation stage and perform only simple input and output type checks.

The process of searching for a composition might be time-consuming. Thus, it is better to explicitly allow searching for a composite data transformation. For this reason, the consumer's request must contain a corresponding directive. In fact, there are two possibilities to search for composite data transformations: *on-demand* and *offline* search. The former is related to processing the consumer's request which allows a composite search. The latter refers to searching for composite transformations independent of incoming requests and caching the returned results. While on-demand search is the most obvious solution, it also makes the request processing time significantly longer. On the other hand, the difficulty of knowing which composite transformations to look for in advance makes the offline search not trivial. One way to gather potential compositions is to cache consumer's requested transformations and use these for offline search. Additionally, it is possible to select particular groups of input and output types suitable for transformations into each other. To be more precise, for many types the data transformation between them might be useless, e.g. transforming video formats into textual formats. For this reason, the types can be separated into dedicated groups, e.g. textual or image, and composite transformations might be search offline for groups of interest. For example, composite transformations for the textual types. Worth mentioning that the search will still involve all the available

applications, but the starting and ending sets will be limited to those supporting the desired types. Offline search can be seen as an optimization which allows reducing the search time in case an actual consumer's request will be received.

When a composition is found and chosen by the consumer to be invoked the task manager has to handle the invocation process. Abstracting from the task manager's specifics which was not yet discussed, we can mention a couple of issues requiring attention during the process. Firstly, different types of applications might be involved in a composite data transformation, e.g. a web service and a self-contained application. Hence, the task manager must support the invocation of every involved type of application. Another important issue is handling intermediate inputs and outputs in-between the invocations. The most important question is how the storage has to be organized. Moreover, the errors require special handling. If an intermediate application fails, task manager has to know how the results need to be handled, e.g. retry to trigger the application again, let the consumer decide or stop processing the transformation request.

### 3.5.6 Optimizations

There are potential optimizations which might be useful throughout the repository's life cycle. As a means to group them we use an abstract sub-component of the repository called *optimizer*. In particular, this sub-component can be responsible for the task of deduplication, searching the composite data transformations or performing data mining tasks like clustering the dependencies based on the supported transformations. In this subsection we briefly discuss how optimizer can solve these tasks.

Unfortunately, the problem of publishing duplicates cannot be completely solved with application identifiers as specifications might come, e.g. with spelling mistakes. The search for duplicates can be run by the optimizer in the background based on the defined set of parameters describing the application, mainly from the application's description part. However, opposing to the publishing case an efficient comparison technique has to be used in order to avoid comparing every possible pair of applications which leads to a quadratic complexity. One option is to use a Sorted Neighborhood Method [HS95; HS98] which requires generating a key for every application, sorting them based on the key, and comparing all pairs of the applications within a sliding window of some fixed size. This allows reducing the overall number of comparisons. In our case, the key for applications has to be based on the combination of the information which will most likely remain constant, e.g. the supported transformation, concatenated information about the developers. After the potential duplicates have been identified they have to be marked and resolved. However, the actual resolution can happen only in a semi-automatic fashion as one or several publishers have to decide if the applications are in fact duplicates or not.

The composite data transformations can also be searched by the optimizer in the background, e.g. using one of the methods mentioned previously. After a composite transformation has been found it needs to be stored for future use. Thus, the conceptual model of the database information has to allow storing such data. Additionally, as the set of

applications is constantly changing some composite transformation might become obsolete due to removal of some applications. For this reason, the list of found composite data transformations has to be periodically refreshed.

Another enhancement is the usage of data mining techniques in order to derive new information from the repository. This can be used in various scenarios. For instance, the GUI-based interaction with the repository might rely on demonstration of applications and clustered groups of applications based on the transformation type are much easier to analyze. Another scenario is the discovery of associative rules based on previous consumer's request and trying to suggest a suitable application. The analysis of repository's contents might also be implemented as a part of optimizer's responsibilities.

### 3.5.7 Refining the Repository

Having discussed various aspects of publishing and storing data transformation applications we can refine the architecture of the repository by zoning the responsibilities into separate sub-components as shown in Figure 3.17. The bottom part is the storage layer which is

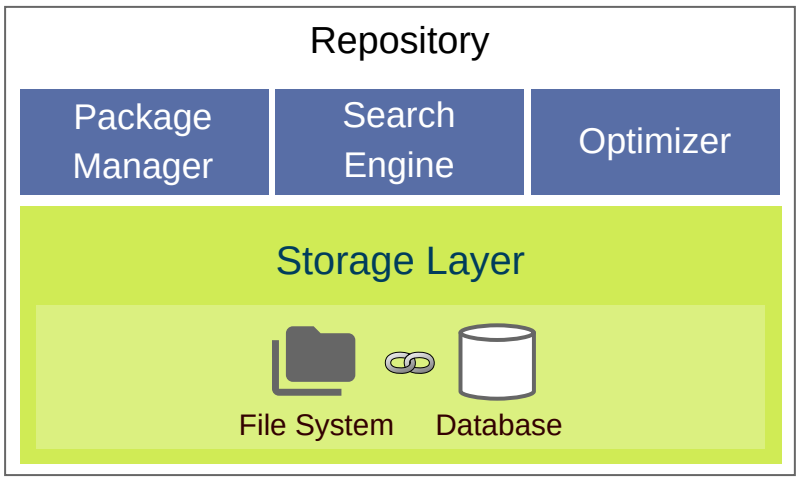


**Figure 3.17:** The refined architecture of the repository

based on the combination of a file system for storing provisioning-ready packages and a database for storing application-related information. On top of the storage layer there are three sub-components grouping the previously discussed aspects:

**Package Manager** is responsible for generating a provisioning-ready package and handling other package-related tasks.

**Search Engine** is responsible for searching suitable transformations and search-related tasks.

**Optimizer** is responsible for possible optimizations and enhancements.

To be more precise we need also to include the interaction and security layers to the repository as it can be seen as a standalone piece of software functioning independently of other components. Thus, the repository has to provide means for secure interaction with the publishers and consumers. In our case the repository alone is not sufficient for handling the data transformation tasks as the invocation part is missing in such setting. For this purpose, we discuss the interactions and security layers as a topmost layers of the overall architecture in the meantime omitting them from the refined architectures of the sub-components.

## 3.6 Task Manager and Provisioning Layer

Essentially, the task manager's responsibilities can be divided into two categories: performing the transformation and deploying an application in case it is not yet deployed. The latter can also be a sub-part of the former in case an application specified in the request was not previously deployed. Figure 3.18 illustrates the task manager's architecture consisting of the application deployer, task processor which is responsible for running a set of tasks, and provisioning layer at the bottom. In the generic architecture we listed the provisioning layer as a separate block to demonstrate that there is a need for a technology supporting the invocation of self-contained application packages. In fact, this layer is coupled with the task manager supporting a certain technology, e.g. invoking Docker containers via Docker's API. Therefore, it makes sense to discuss the task manager and provisioning layer together as a whole. As with the repository, we omit the user interaction and security layers in the task manager's architecture. Although it also can be considered as a standalone piece of software, in our case it is preferable to discuss it as a part of the system having common user interaction and security layers.

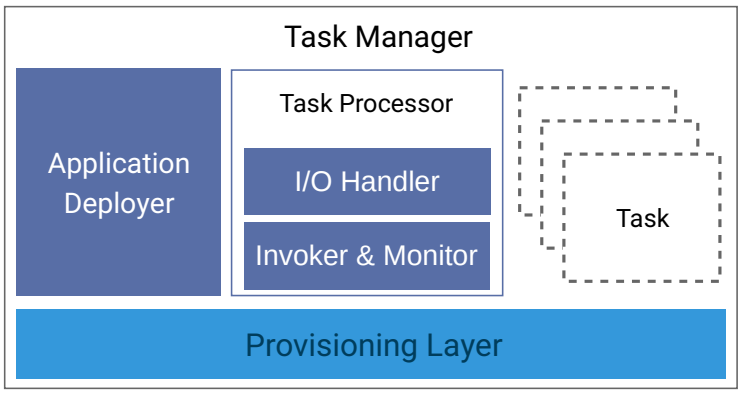


**Figure 3.18:** Task Manager's architecture

The *Application Deployer* is a sub-component responsible for deploying the data transformation applications using the supported provisioning layer. In this work we focus on container-based provisioning as it is a lightweight alternative for hypervisors and may also serve as a building block for other available options when needed, e.g. containers used inside virtual machines. Although containers introduce simplicity, they also have

limitations like a weaker security model. As was discussed in Chapter 2, there are scientific container solutions improving the security and other related issues. Worth mentioning that the application deployer is only needed for reusing the applications which are published as self-contained packages. Deploying a web service would mean that it was also published alongside its files and other related information. Publishing scenarios discussed in Section 3.5 list several options of when an application can be deployed. Basically, all these options differ only in the type of request sent, i.e. either a deployment during publishing or during transformation, but the deployment process stays the same. These two examples of a request originate from different senders, a publisher or a consumer. In case of a deployment during publishing, the request explicitly states that an application has to be deployed as it is the part of the publishing process. However, when a consumer requests for a transformation a list of suitable applications is returned in response. This implies that a consumer needs to choose one of the applications based on some criteria and send another request stating that the transformation has to be performed by the chosen application. At this point, there is a possibility that an application was not deployed previously and before the transformation takes place the deployment must happen. This information is also available in the request itself due to the provisioning information maintained by the repository. As a result, the request has to be split into deployment and transformation requests and processed by the task manager consecutively. When an application was successfully deployed the confirmation must be sent to the repository in order to update the provisioning information for the respective application. In addition, the specification information of the newly deployed application might be cached in order to simplify the access to the invocation-related information. In this case caching will also introduce the need to synchronize the data in order to reflect the changes in the repository. Without use of caching the task manager will need to get this information either from the repository or access the specification directly inside the container in order to invoke the application.

The *Task Processor* is responsible for the invocation of the application and control over the execution process. This includes processing of the input and output, invocation and monitoring, returning the results, and clean up. If an application was published as a web service, the task manager only needs to act as a proxy redirecting the requests and responses. Figure 3.18 shows the task processor consisting of two sub-components: *Input/Output (I/O) Handler* and *Invoker & Monitor*. The former is responsible for preparation of the inputs and outputs. Requests for the data transformation obviously should contain the inputs provided by the consumer of the application. The inputs might be provided in either push or pull-based manner. The case when a consumer directly includes the inputs into the request is a push-based approach. It is not preferable in case large inputs need to be provided. As an alternative, a consumer might provide the links referring to actual inputs and task manager then can download the payload. With this approach a request becomes more lightweight and flexible, as various potential locations and public services can be used as a source of input. On the other hand, it makes no sense to specify parameter inputs as links. For this reason, the request can be a combination of both push and pull-based approaches: using the links for file inputs and embedding parameter inputs into the request. As soon as the request is received, the I/O Handler needs to prepare the inputs, meaning that the input files need to be downloaded and parameter inputs need

to be saved as well. Then the application has to be invoked using the prepared inputs. Note that an application container can be instantiated prior to input preparation and wait until everything is ready for the execution. The Invoker and Monitor sub-component is responsible for invocation of the application based on the information from the application package specification and monitoring its state during the execution. Every request must be handled in a separate container to avoid overlapping of inputs and outputs. As a result, an individual task has to be created for every instantiated container and after completion it can be eliminated. It is rather impossible to organize a monitoring on the application level due to legacy nature of the applications. In many cases an application does not provide a way to check its state. As an option, a monitor can, e.g. check the contents of the output folder or use operating system's tools to get the information about the respective process. From a consumer's perspective, as soon as the transformation task is started there must be a way to check for an execution state via a standardized heartbeat-like request. Basically, the task manager must be able to tell if an application is still working, failed or completed. Both failed and completed states can be identified by checking the state of the process in the operating system. However, if an application froze it might appear that it still works. In such case specifying a timeout either in the request or in the task manager could be helpful to decide if an execution was successful.

## 3.7 Request Router

Essentially, the *request router* is analogous to an Enterprise Service Bus (ESB) [Cha04] in its purpose. On one hand, it is possible to omit the request router and work independently with the repository and the task manager via their interfaces. On the other hand, the number of repositories and task managers may be bigger than one and using a single endpoint with a standard interface for interaction is more convenient. Therefore, a request router serves for a single endpoint for potentially multiple task managers and repositories. Additionally, it enforces the compliance with a standard interface. An interaction between the repository and task manager is required in several scenarios, e.g. publishing with testing, and without request router both repository and task manager need to know each other in order to communicate. If several repositories or task managers are available this task becomes even more complicated. Instead, every sub-component needs to communicate only with the request router which keeps track of the routing logic. Moreover, it can also provide load balancing in case multiple sub-components of the same type are used.

## 3.8 User Interaction and Security Layers

In order to safely interact with the overall architecture user interaction and security layers are needed. The former provides means to communicate with the system in a standardized way. In reality, the user interaction process is different for publishing, i.e. working with the repository, and for reusing published applications which involves interaction with

both the repository and provider of the logic. In our case the topmost user interaction layer is basically a common ground for interfaces of the repository and task manager. More specifically, defining an standard interface on the higher level enforces any possible implementation of each sub-component to comply with it. User interaction can happen in different ways: in particular, we are interested in a separation between a GUI-based interaction and working directly with the API. These types of interaction are useful in different scenarios. For instance, publishing an application without relying on the GUI is less convenient for publishers as many details have to be specified and possibly refined. On the other hand, the process of reusing an application involves several distinct actions which may or may not leverage from GUI-based interaction. For instance, searching for an application can be done in both ways: either viewing a list of available applications via GUI or sending a request containing all the required information and getting a list of suitable applications. The latter option can be a part of an automated interaction when the consumer is another application. The returned list of available applications is then checked based on some criteria and the most suitable option is chosen for performing a transformation. With these considerations in mind, we describe the user interaction layer consisting of two parts: a GUI and an API as shown in Figure 3.19.


**Figure 3.19:** Updated User Interaction Layer with the Security Layer

The concept of trustfully running third-party applications is far from being safe. A security layer needs to rely on a set of complex measures in order to provide consumers with safe-enough user experience. The fact that a transformation runs inside a container makes it look safer. However, a consumer has to be protected from reusing malicious applications which can distort the result of a computer-based experiment, produce potentially harmful output, etc. A set of required security measures is not in the focus of this thesis, however we mention several aspects which might be helpful. The main security issue is the way how applications are published. Unauthorized publishing should be prohibited or marked accordingly in the application's information. In order to be authorized, the system has to keep track of users, e.g. by means of a database. Respectively, a user registration procedure has to be implemented. The same approach can be used for consumers prohibiting any unauthorized requests. Additional security mechanisms like Role Based Access Control (RBAC) can be applied for control of who can consume the applications.

## 3.9 Refining and Zoning the Framework

Combining all discussed concepts together we can now refine the framework using the architecture demonstrated in Figure 3.12. After substituting every sub-component with its refined version, we get the result shown in Figure 3.20. Additionally, we use the red

dashed lines in order to group the sub-components into conceptually-related zones which can be used when developing a prototype, e.g. as a set of microservices. The repository and task manager are represented as atomic sub-components, whereas the combination of the request router, interaction and security layers is called the *Interaction Endpoint* zone. The idea is to have a single endpoint for communication with all the available repositories and task managers which are compliant to the interface offered by the interaction zone. On one hand, this introduces additional complexity for handling requests as they need to be routed and the actual repositories and task managers have to be known. Moreover, such interaction endpoint is the bottleneck and the single point of failure in the system. On the other hand, having one interaction endpoint simplifies the overall interaction process for publishers and consumers and enforces the sub-components to comply with a standardized interface. In the next section, we discuss how consumers and publishers can interact with the framework.

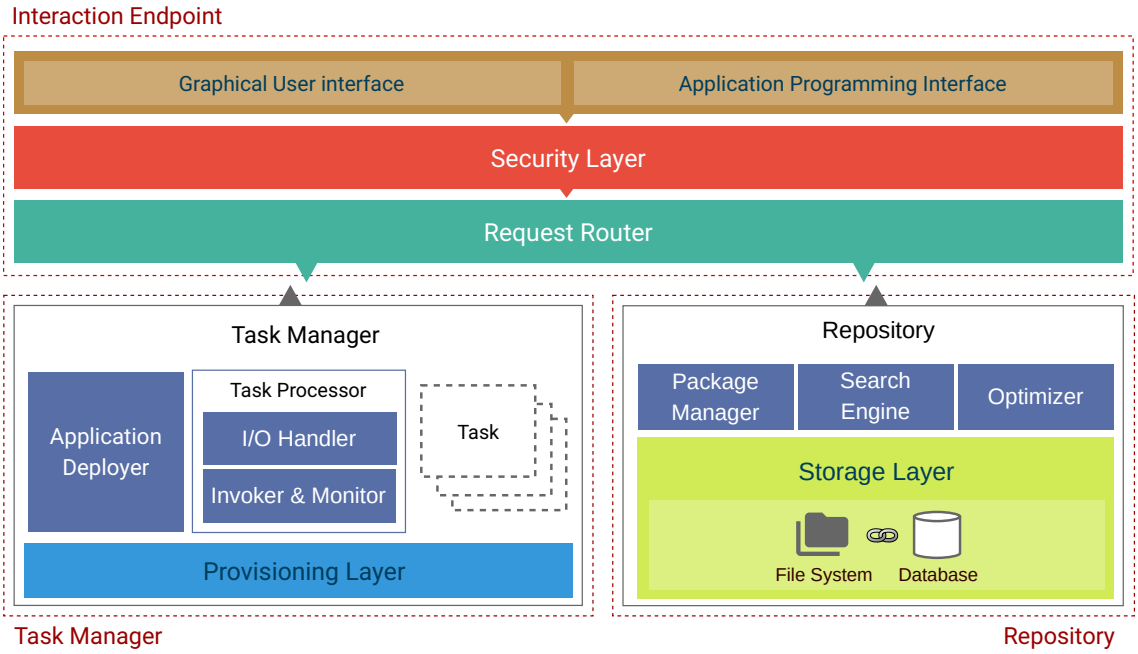


**Figure 3.20:** Refined view on the framework's constituents

## 3.10 Interaction with the Framework

The discussion of overall framework is not complete without explaining the interaction processes from various perspectives, i.e. describe publisher's, consumer's, and framework's view on the interaction. Before reusing an application, it has to be published. Figure 3.21 shows an abstract diagram of the generalized publishing process from the publisher's perspective. In order to publish an application, the publisher has to provide the application package specification and prepare all related files and dependencies. Additionally, the publisher has to specify whether an application needs to be provisioned and tested. We

consider publishing to be a semi-automatic process and assume that it is GUI-based as it offers a more user-friendly way to control various specification parameters and prepare related files.


**Figure 3.21:** The generalized publishing process from the publisher's perspective

From the user's, i.e. publisher's perspective, it is preferable that the process is simple and intuitive. The hidden details of the publishing process are easier to discuss from the system's perspective. By system, we mean the combination of framework's sub-components involved in the process. Figure 3.22 illustrates the publishing process from the system's perspective. After receiving the publishing request, several steps have to be performed for a

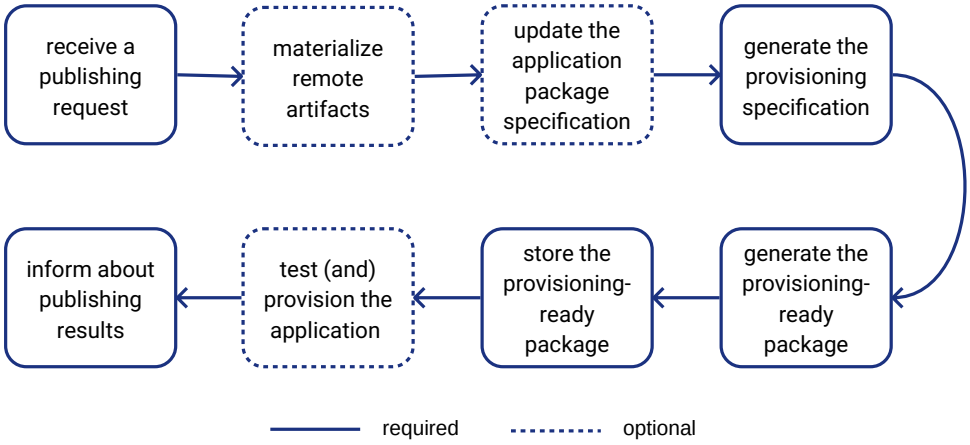


**Figure 3.22:** The generalized publishing process from the system's perspective

process to finish. Optionally, in case the application package specification contained remote links to artifacts they have to be materialized and the specification have to be updated. The next step is to generate the provisioning specification supported by default, e.g. a Dockerfile. The resulting set of files then needs to be archived into a provisioning-ready package which will be stored on the file system. Corresponding database entries have to be created in addition. Depending on the publishing scenario chosen, the testing and provisioning of the package might be required. Finally, the publisher needs to be notified about the results. Several steps might involve errors which need to be handled. For instance, the remote files specified in the application package specification might be unavailable which leads to stoppage of the process. As an optimization measure, several retries might be performed in order to get the files. However, in case the files are still not available the publisher has to be informed about the problem as a next step due to the impossibility to continue the publishing process without the refinement of remote links. Basically, in case of errors during the next steps the overall process has to be halted and the publisher needs

to be informed. One issue worth mentioning is the case when the test of an application is finished and the produced output is not equal to the output provided in the test run specification. Essentially, at this step an application can be published. On the other hand, due to different results the application must be considered incorrect. Thus, it has to be marked as producing inconsistent results and the publisher has to be notified about this fact.

From the consumer's perspective the process of reusing an application must be simple to implement, leading to a lower cost of the software reuse. Figure 3.23 illustrates the abstract process of reusing a data transformation application as it can be seen from the consumer's point of view. Firstly, the consumer needs to search for a suitable application by sending a request which eventually has to be delivered to the repository. Potentially, the number of suitable solutions can be greater than one resulting in a list of applications which can be used. For instance, such list might include single atomic applications and composite applications. The framework does not offer means to choose which particular application is the most suitable for a consumer's use case. This decision making process has to be performed on the consumer's side. Potentially, the framework might rank the applications in the list based on various criteria such as performance properties or credibility which can simplify choosing the most suitable application for a consumer. However, every use case might have specific requirements and the overall task of choosing the right application is left for the consumer. After the decision has been made, the request for transformation

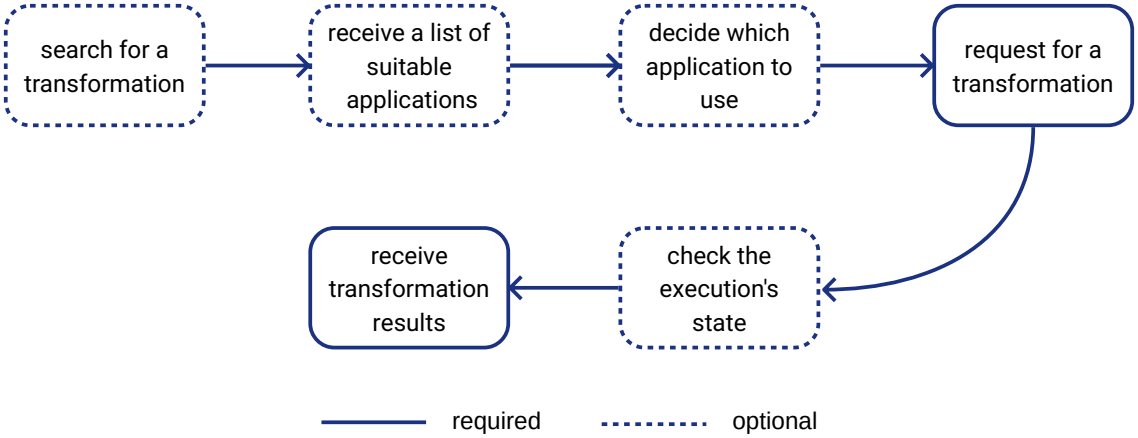


**Figure 3.23:** A consumer's perspective on the process of reusing a data transformation application

has to be sent which eventually reach the task manager. At this point, all the consumer is interested in is getting back the results of the transformation. Worth mentioning, that the whole process of searching for an application is optional as the consumer might already know which transformation application is going to be used, e.g. by means of the QName property discussed in Section 3.3.

One optional step that might be useful is the monitoring of the application's execution process. For this reason, a standardized monitoring request has to be allowed. One important issue is that the transformation task is created after receiving a request which

leads to the requirement of sending a task's reference to the consumer in order to allow monitoring. This process is transparent to the consumer, but the knowledge of task's identifier is also crucial for correlation of the response. Therefore, the monitoring request has to contain a task's reference in order for the system to process it. As soon as the transformation is completed, the results have to be returned to a consumer. There are several options of how the results can be returned. Obviously, the response can contain the results directly. Another option is to temporary cache the results and send a generated unique link to the consumer. Basically, these are the push and pull-based scenarios. The latter is preferable in case the transformation is needed only as a temporary step for performing one or more additional data transformations. In such case, the consumer does not need to resend the same inputs as outputs for a new transformation and the task manager can directly reuse the inputs stored locally.

Another view on reusing the data transformation applications is from the system's perspective. Figure 3.24 shows what happens after the transformation request is received by the task manager. Depending on whether the application was provisioned before or not, two optional steps might be needed. Firstly, the provisioning-ready package has to be downloaded from the repository. Considering the fact, that the task manager does not know the repository in advance this information has to be provided. Since the information about provisioning is already known at the time of the transformation search, the response to the consumer with the list of suitable transformations can contain the abstract reference to the repository. When the consumer sends a transformation request, the abstract repository reference can be substituted with the actual endpoint information by the request router. As a result, when the task manager is ready to process the transformation request it already knows all relevant information, i.e. provisioning status and the repository information. Hence, the task manager can download the respective package in order to deploy it in a subsequent step.

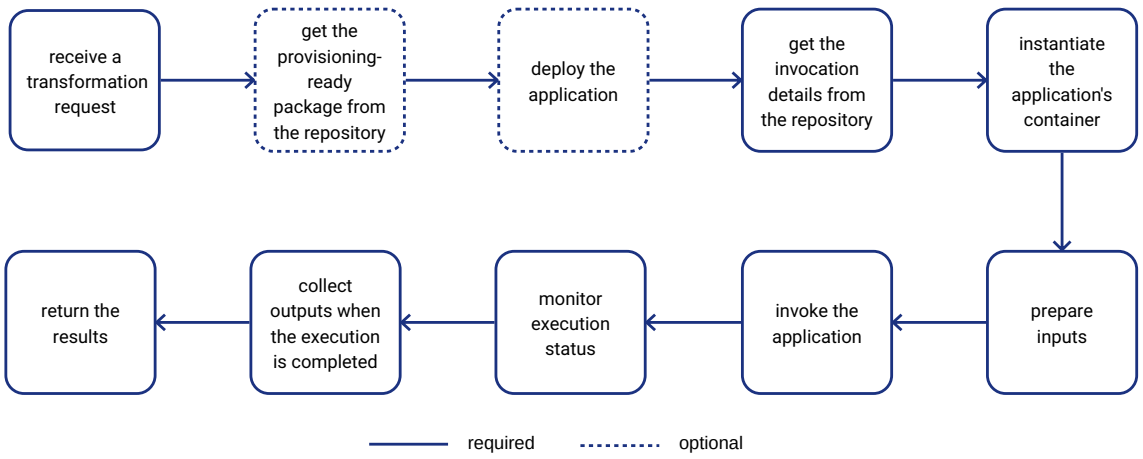


**Figure 3.24:** A system's perspective on the process of reusing a data transformation application

Next step is to get the invocation details from the repository. This information can also be accessed in the provisioning-ready package or cached during the deployment. Without

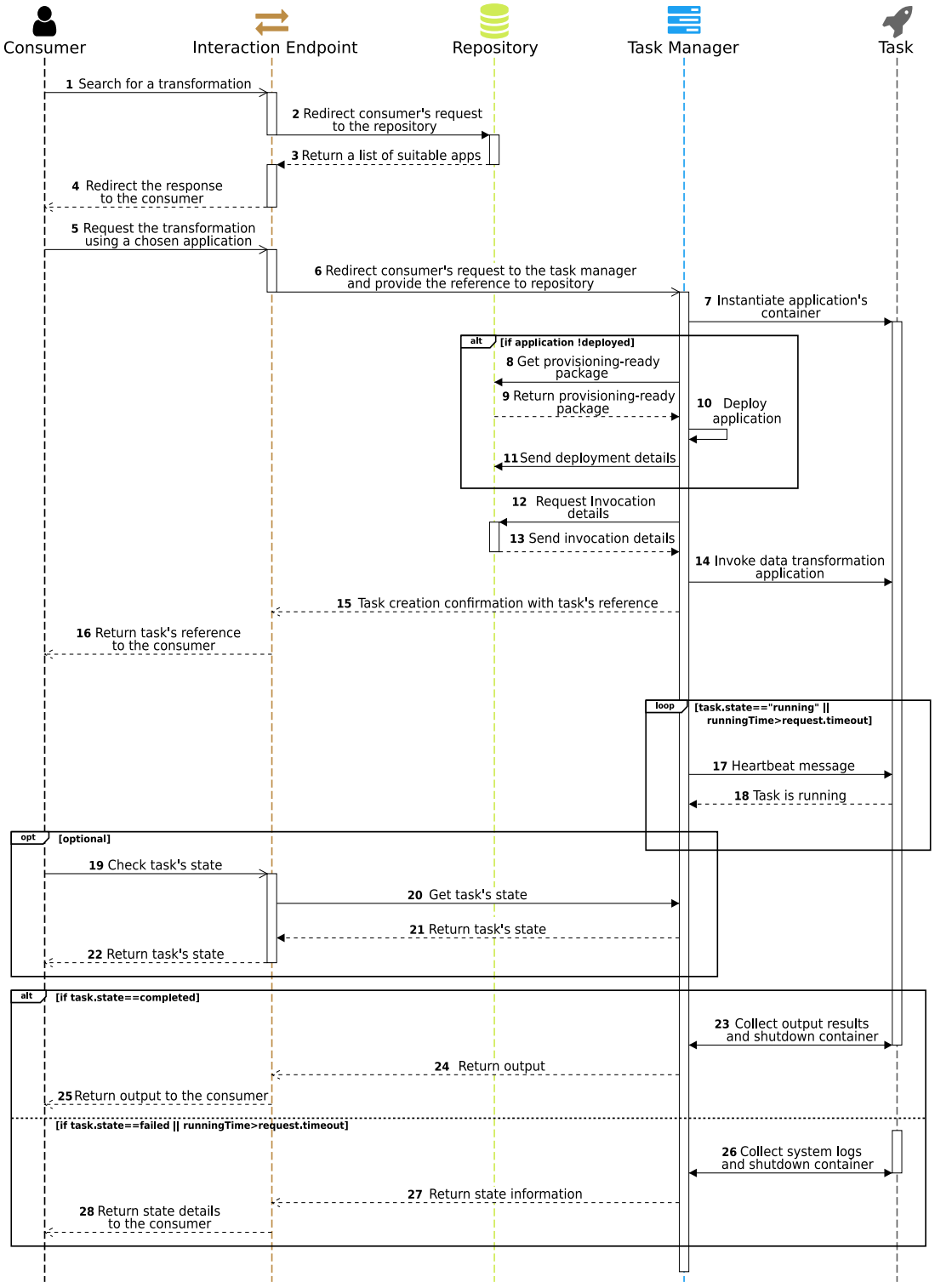


**Figure 3.25:** An UML sequence diagram of the interaction with the framework

going further into possible optimizations, we assume that this information is obtained from the repository. Afterwards, the application's container needs to be launched and the inputs have to be prepared as application package specification states. The preparation of inputs includes obtaining the input files using the remote links in request and placing them in a required location in the application container. For instance, if the application requires having inputs in a specific folder with respect to the executable, then the inputs have to be copied in this folder. Another option is when the invocation string contains the list of files specified. In such case the inputs have to be copied into a predefined folder and the executable can be invoked using a concatenation of the static path and file's name. The application can be invoked after these steps are completed. Unlike in consumer's perspective where monitoring is optional, the system has to monitor execution as there is no other way to understand the state of execution. Moreover, the published data transformation applications are diverse and offer no standardized way of checking their state. One way to check the application's state is to get the process information from the operating system. Another option would be checking the output folder in case the files are saved somewhere and compare the number of output files. However, even when the number is equal we cannot guarantee that these files are complete and not being populated. In addition, it is also possible to log the application's output from the console to the file for additional information. For the case when the application is running for too long a task can be stopped using a timeout specified in the request. As soon as the task is completed the outputs have to collected and returned to the consumer.

An example shown in Figure 3.25 illustrates generalized communications between the consumer and framework's parts involved in a reuse scenario. Basically, the general process combines the consumer's and system's perspectives on application reuse. The consumer communicates with the interaction endpoint without the need to care about the exact repository and task manager involved. This example does not contain all possible issues which might arise during the interaction, especially with respect to potential errors. For instance, in case the application was not deployed due to a failure, the transformation cannot proceed and the corresponding response has to be returned to the consumer. Similar behavior is expected when invocation details are not accessible from the repository. As soon as the transformation is invoked, the consumer has to receive the task's reference to be able to send the monitoring requests. On the task manager's side the monitoring has to be performed constantly using a specified timeout. Based on the task's state the corresponding response has to be returned to the consumer.

## 3.11 Integration with TraDE Middleware

In order to use the framework in conjunction with the TraDE Middleware we need to distinguish the most suitable integration options. However, before going into details, first we must understand the goal of integration. The publishing process can be seen as an independent part of the framework due to its semi-automatic nature. Obviously, it is an important part of the process, but the actor interested in performing the transformation

assumes that applications are present in the repository. For this reason, we assume that the publishing is performed independently and does not depend on specific choreography and data flow in particular. More specifically, the main goal of integration is to support the transparent data exchange by means of offering the data transformation logic for inclusion into the data flow. In this case, the TraDE Middleware can be seen as a consumer, first searching and then reusing the chosen application.

There are four common ways to integrate software: *file transfer*, *shared database*, *remote procedure invocation*, and *messaging* [HW12]. The file transfer implies using files as a common data transfer mechanism. This technique is not suitable when applied to our use case due to several reasons, including handling a large amount of files is costly performance-wise, synchronization of the files might be needed to avoid staleness. The shared database integration is also not suitable for this particular integration due to a couple of issues, e.g. the database might be a bottleneck when many requests have to be processed, an asynchronous communication is complex.

Instead, we focus more on remote procedure invocation and messaging integration techniques. The former is about providing an interface for other applications which allows communicating directly with an application [HW12]. Considering the whole framework as a single piece of software which provides an interface, the remote method invocation might be a suitable integration technique. Essentially, the TraDE Middleware needs to use the interface of the framework in order to send standardized requests and receive the results. Due to implementation specifics of the TraDE Middleware, web service-like interaction is more preferable than using more tightly-coupled solutions like Java Remote Method Invocation (Java RMI).

Making a step further, we can try to apply the messaging approach to integration of the framework with TraDE Middleware. Using messages as a data transfer mechanism have multiple advantages, e.g. reducing the coupling between applications, or supporting asynchronous communication [HW12]. It relies on the powerful concept of messaging systems which are dedicated pieces of software supporting message exchange. In the framework we have a zone called interaction endpoint which consists of request router, user interaction and security layers. The combination of API, security measures and a request router can in fact be seen as a message-processing system with some messaging system serving as a basis. A secure GUI-based interacton in this case needs to be handled separately.

The messaging approach is preferable in case of high messages amount. In general, we consider a web service-like integration where the TraDE Middleware uses a standardized interface offered by the framework. For this purpose a simple API client has to be integrated to the TraDE's side for enabling communication.

# 4 Implementation

This chapter discusses the details of the data transformation applications handling framework's prototypical implementation. In Section 4.1 we present the details about the prototype's architecture. Section 4.2 goes into the details about the framework's API specification. Section 4.3 describes how the conceptual model of a data transformation application introduced in Section 3.3 is represented in the implementation. Section 4.4 provides the details about the publishing and generation of an application's provisioning specification. Section 4.5 describes the details about requesting a transformation.

## 4.1 Architecture of the Prototype

Although the refined framework's design shows potentially multiple independent components communicating via request router, for the prototypical implementation we choose to unify one repository and one task manager as parts of one monolithic web application. This allows us to focus on public API instead of going into details of inter-component communication. Figure 4.1 demonstrates the architecture of the prototype which is based on the refined framework's structure discussed in Section 3.9.

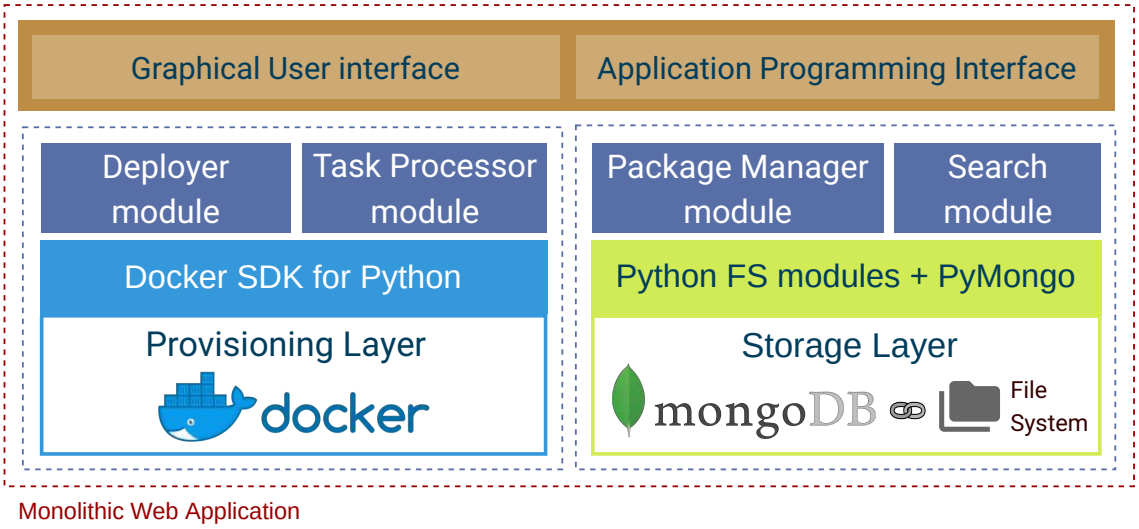


**Figure 4.1:** The architecture of the framework's prototypical implementation

We use Python[1] as the main programming language. However, a stack of additional technologies is required to support the discussed concepts. For the provisioning layer we use Docker [Doc17b] as technology supporting container-based virtualization (see Section 2.4). As a consequence, we need to generate Dockerfiles as provisioning specifications. While for the prototype we are bound to Docker, the applications package specification supplied by the publisher is also included into the final application package. This ensures that the repository remains technology-agnostic and another type of provisioning specification can be generated if needed. In order to communicate with Docker programmatically, we need to use the Docker SDK for Python[2] which allows using Docker's API from within a Python application. While we don't list every single sub-part of the framework used in the prototype, we introduce logical groupings of framework's components and their sub-components into separate Python modules. This logical grouping allows splitting the prototype into standalone microservices exposing their own distinct APIs which can then be composed in future to form the overall framework.

We implement the repository's storage layer as a combination of a file system and a database using the approach discussed in Section 3.5.3. The file system is used for storing the application-related files, whereas the database stores a metadata about the applications. As a database, we use MongoDB[3], a non-relational document-oriented database. We choose to use a non-relational database due to several reasons. It offers expressive querying capabilities, supports secondary indexes and at the same time it is schemaless which is advantageous for our use case. One important thing to consider is that the repository is mainly used for reading. The only writes that can happen are either related to publishing or updates which are meant to happen rare. Most of the database operations are reads in order to find a suitable transformation or obtain the provisioning-ready package. One of the biggest advantages is the absence of a schema which makes storing the application's data extremely flexible, e.g. when the conceptual model needs to be extended. In addition, the model of a data transformation application shown in Section 3.3 is nested which requires creating many redundant tables if a relational database is used and the data needs to be normalized. For instance, the information about dependencies, configurations, invocations is only useful when bound to a particular application. MongoDB stores JSON documents which simplifies storing nested objects. Moreover, it allows creating interconnected collections of documents making it possible to separate parts of the application package specification like dependencies or transformations into separate collections. As JSON is commonly used as a format of data interchange, the whole documents can be returned directly as a response to API calls. In order to work with MongoDB we use PyMongo[4] as a Python database driver. For working with the file system we use a set of standard Python libraries. One important point is that we rely on the file system of the machine which runs the prototype. However, this can be extended, e.g. a distributed file system can be used. As with the task manager's sub-components,

[1] Python Software Foundation, Python: https://www.python.org
[2] Docker Inc., Docker SDK for Python: http://docker-py.readthedocs.io/en/stable/
[3] MongoDB, Inc., MongoDB: https://www.mongodb.com/
[4] MongoDB, Inc., PyMongo: https://api.mongodb.com/python/current

the repository's parts are separated into separate Python modules. The optimizer subcomponent is not in the scope of this work, hence we omit it in the implementation.

The prototype's functionality is exposed via a REST API and partially via the GUI (publishing the applications and basic querying). To support web-related functionality we use Python Flask microframework[5]. For specification of the REST API we use Swagger[6] which is an open source implementation of the OpenAPI Specification [Ini17]. A Swagger API specification is a YAML Ain't Markup Language (YAML) file containing all the interfaces-related information in a human-readable format which later can be used for the generation of the client and server boilerplate code for various languages and frameworks including Python Flask.

## 4.2 API Specification

Having discussed the refined architecture of the prototype and technologies used, we can go into details of the API specification. One of the primary goals we need to achieve is an identification of the resources and analysis of the operations which can be performed with them. Table 4.1 lists the resources of interest for the prototypical implementation along with their descriptions and root paths. The *Applications* resource describes a data transformation application and is based on the application package specification which in its turn is a representation of the conceptual model introduced in Section 3.3. Essentially, the *Transformations* resource represents a part of the application which we also make accessible independently as a separate resource. The *Tasks* resource identifies the transformation tasks created by the task manager component described in Section 3.6. We use plural

| Resource | Root Path | Description |
|---|---|---|
| *Applications* | **/apps** | This resource identifies a collection of data transformation applications which were published to the repository |
| *Transformations* | **/transformations** | As one application might support multiple transformations, we use a separate resource to identify a collection of available transformations |
| *Tasks* | **/tasks** | This resource identifies a collection of transformation tasks requested by the consumer |

**Table 4.1:** Resources descriptions

forms of the respective nouns to describe the collections of resources which is one of the

[5]The Pocoo Team, Flask: http://flask.pocoo.org
[6]SmartBear Software, Swagger: https://swagger.io

recommended ways for specification of REST APIs [Mas11; Mul13; SB15]. Note, that we do not identify security-related resources such as users or sessions as we omit the security layer in the prototype.

According to Fowler's description of Richardson's REST maturity model [Fow10b] the next step is to introduce a proper usage of HTTP verbs with respect to the identified resources. Table 4.2 demonstrates the usage of HTTP verbs for Applications and Transformations resources. We introduce the latter due to the different search semantics: looking for a specific application versus searching for any suitable transformation available in the repository. For the prototype we support searching for both, applications and transformations. The search for a suitable application relies on tags defined in the application package specification. Searching for a specific transformation is possible using the transformation's signature generation discussed in Section 3.5.4. The signature is generated using the data supplied in the search request. The composite search described in Section 3.5.5 is omitted in the implementation. As shown in Table 4.2 the combination of an HTTP GET

| Resource path | HTTP Method | Description |
|---|---|---|
| **/apps** | GET | Retrieve the list of data transformation applications from the repository |
| **/apps** | POST | Publish a new data transformation application to the repository |
| **/apps/{appID}** | GET | Find the data transformation application by its identifier |
| **/apps/{appID}** | PUT | Update an existing data transformation application by completely replacing it with a new one |
| **/apps/{appID}** | DELETE | Remove the data transformation application from the repository |
| **/apps/{appID}/transformations** | GET | Retrieve the list of transformations supported by a particular application |
| **/transformations** | GET | Retrieve the list of available transformations in the repository |

**Table 4.2:** The usage of HTTP verbs for Application and Transformation resources

request with a specific resource, e.g. */apps* or */transformations*, or a sub-resource, e.g. */apps/*{*appID*} results in triggering a corresponding search. While looking for a specific sub-resource requires providing only its identifier, searching for collections of resources relies on specific query parameters. For the */apps* resource we use an HTTP GET request which includes the query parameter "tags" allowing to search for applications by tags specified in the application package specification. Other application-related query parameters can be introduced in addition. However, for the basic prototypical implementation we

distinguish semantically between searching for an application which groups potentially multiple transformations and searching for a specific transformation, e.g. using its qualified name or based on the signature matching. Thus, for the */transformations* resource we introduce following query parameters: “qname”, “pnum”, “infile[]”, “infsets[]”, “outfile[]”, and “strict”. Searching a transformation by its qualified name (“qname” query parameter) is very convenient in scenarios when both publisher and consumer of the application agreed on using standardized abstract identifiers. The search process becomes a trivial task in such case as only transformations with matching qualified names are returned. Other parameters allow describing the transformation to search for. The “pnum” parameter allows specifying the number of input parameters a transformation should be able to process. In the prototype we do not differentiate between the type of parameters. The “infile[]”, “infsets[]”, “outfile[]” parameters are the arrays of formats for InputFiles, InputFileSets, and OutputFiles respectively. The “strict” query parameter defines which type of signature string discussed in Section 3.5.4 will be used for searching: either with optional input parameters or without them. The latter is less strict due to a smaller number of constraints. Note, that search for an application or a transformation are semantically different as resulting responses contain different entities. The response for searching a transformation returns a list of suitable transformations with links to their parent applications, whereas the response for an application directly contains all suitable applications. When a consumer decides upon which transformation to use, the transformation request sent to the framework must contain both references to the application and its transformation. In general, more complex search capabilities can be implemented by introducing special search-related resources which support HTTP POST requests carrying a more complex structure. An example of a GET request which searches for Transformations with two input parameters, consuming a TXT and a DAT file as InputFiles and producing a PDF as OutputFile will look as follows: */transformations?pnum=2&infile[]=txt&infile[]=dat&outfile[]=pdf*. Note that for formats the regular file extensions are used instead of MIME types which take more space. On the repository side we have a static MongoDB collection for mapping between MIME types and file extensions for construction of proper signature strings in case application package specification uses MIME types to describe the formats.

An application can be published using HTTP POST method with the */apps* resource. The response in this case is a JSON representation of a newly-published application. We provide more details about the publishing in later sections. For the prototype we implement a simple update of applications which uses HTTP PUT replacing the chosen application with the information from the request. Note, that a sub-resource identifying a certain application has to be used in this case. Worth mentioning, that a set of checks is performed in order to update an application and the update happens if only everything was successful. Otherwise the publisher gets an HTTP status code 400: “Bad Request”. For more fine-grained updates it is possible to use HTTP PATCH referring to specific resources like dependencies or configurations. We omit this level of details in the prototype to support only the sufficient set of resources. Likewise, an existing application can be deleted using HTTP DELETE method. In addition, the set of transformations supported by a specific application can be received by addressing the dedicated sub-resource.

In order to execute a transformation, a corresponding task has to be created by the Task Manager as introduced in Section 3.6. Therefore, the REST API allows consumers to create transformation tasks by interacting with the Tasks resource. Table 4.3 lists the task-related resources and the HTTP methods they support. An HTTP POST request to the */tasks* resource returns a created task's resource representation. In case a consumer is interested in checking the task's state, an HTTP GET request to a corresponding sub-resource using the received task identifier should be made. Another action on the task resource supported by the prototype is the cancellation of a task by updating its state through an HTTP PUT request. One can argue that the HTTP DELETE method should be used to cancel a task instead. However, we are interested in separating the cancellation and deletion actions for logging purposes. Cancellation is basically an update of the task's state, while the deletion means that the task will be completely removed. Only canceled tasks can be removed, hence the deletion might require canceling the task if it was not canceled before. For logging in the task manager we use plain text files.

| Resource path | HTTP Method | Description |
|---|---|---|
| **/tasks** | POST | Trigger the creation of a new data transformation task |
| **/tasks/{taskID}** | GET | Check the state of the data transformation task |
| **/tasks/{taskID}** | PUT | Cancel an existing data transformation task |
| **/tasks/{taskID}** | DELETE | Delete an existing data transformation task |

**Table 4.3:** The usage of HTTP verbs for Task resource

The last level in Richardson's REST maturity model [Fow10b] relies on the so-called Hypertext As The Engine Of Application State (HATEOAS) concept. Basically, the main idea is to support the navigation through API using links to relevant resources [SB15]. For instance, if a certain application was returned in response to publishing action, it also needs to contain the links at least to itself and its transformations as shown in Listing 4.1. The

**Listing 4.1** Example of Hypermedia Controls for the case when an application is published

```
...
"links": [
   {
      "rel": "self",
      "uri": "/applications/someAppID"
   },
   {
      "rel": "application.transformations",
      "uri": "/applications/someAppID/transformations"
   }, ...
]
...
```

usage of “self” might be helpful in cases such as obtaining a resource representation from HTTP POST request where actual address of the resource is different from the originally requested one [SB15]. This particular example is only useful in case the publisher wants to use this application as a consumer after publishing. However, this approach simplifies the exploration of the details (if needed) about applications and transformations received as search results. For content negotiation in the prototype we use JSON format as it is less verbose than XML and integrates easily with MongoDB.

## 4.3 Application Package Specification

Before discussing the publishing details, we need to describe how the application package specification introduced in Section 3.3 is implemented and which way of publishing we use for the prototype. Essentially, an implementation of a data transformation application package specification can be seen as a DSL. For the implementation we choose it to be an external DSL meaning that it does not use the syntax of some general purpose programming language [Fow10a]. This allows reusing application package specification in a technology-agnostic manner. Although supporting the control flow logic in DSL might potentially be useful in describing data transformation applications, we choose to use a declarative style as it makes the overall description process easier for a publisher. An external DSL can be based on either a custom syntax, e.g. a newly-created language, or a commonly-used representation formats such as XML or JSON. We use JSON due to several reasons. First of all, basically all the information in the prototype is exchanged in JSON and JSON-based description speeds up overall processing, e.g. parts of the specification can be directly copied into corresponding MongoDB collections. Additionally, the newly created syntax for a declarative DSL has to be thoroughly documented and the creation of a specification might require more effort. In contrast, for JSON we use JSON Schema and validate the publisher’s specification against it.

The conceptual model of application package specification introduced in Section 3.3 can be represented in JSON almost as 1:1 mapping. Only some parts might differ slightly due to the choice of data structures. For instance, we group the same types of entities into separate arrays to simplify parsing of the specification. Listing 4.2 shows how different input types are represented by separate object properties. We use the same approach for other similar entities in the conceptual model, e.g. types of dependencies. We omit listing an abstract example of the specification as most of the mappings are easy to reproduce. However, in Chapter 5 we demonstrate how concrete data transformation applications from the motivational example presented in Section 1.2.6 can be described using the discussed JSON format.

**Listing 4.2** Example of input type groupings in transformation specification

```
...
"transformations": [
   {
      "name": "text-to-pdf",
      "qname": "uniqueQualifier",
      "inputParams": [ { ... }, ... ],
      "inputFiles": [ { ... }, ... ],
      "inputFileSets": [ { ... }, ... ], ...
   }
], ...
...
```

## 4.4 Publishing and Generation of Provisioning-ready Specification

When an application's specification is created in JSON and all related files are prepared, the publisher can choose the most suitable option among the publishing scenarios discussed in Section 3.5.1. Then, a corresponding HTTP POST request can be issued against the API. From the technical point of view there are several ways of publishing a specification and related files. First of all, supplying the application's files separately might become a tedious task due to a potentially large number of files. The easiest way would be to attach an archive of all related files including the application package specification. However, sending an archive included in the body of the request is not directly possible because of additionally required information which has to be specified apart from the archive files, e.g. whether an application has to be provisioned and (or) tested. One possible way is to use *multipart/form-data* content type for sending a form containing the application metadata and attached files or send structured JSON requests with the base64 encoded files inside. However, if the archive's size is large, sending such POST requests will take more time. Moreover, if the validation does not succeed the publisher needs to upload the files again. Another option is to split the publishing process into two stages, first specifying the metadata and then providing the files archive. From the conceptual point of view, we cannot create an application entry in the database until the files are received and validation takes place. For the prototype we use a simple option of sending a JSON request containing the related metadata plus a remote link to the application archive, e.g. to a Dropbox[7] or a GitHub[8] repository. Sending such requests is simple and a publisher does not need to upload files again if the validation was not successful. In the metadata the publisher also needs to specify which publishing scenario is going to be used, e.g. including provisioning. Listing 4.3 shows a publishing request's body example. The specification of application's name is not necessary as the application package specification contains all the required data. In case the selected publishing scenario succeeds, the publisher receives the JSON representation of a newly-published application. As MongoDB allows creation of

[7] Dropbox, Inc.: https://www.dropbox.com
[8] GitHub, Inc.: https://github.com/

**Listing 4.3** Example of a publishing request

```
{
   "name": "dummyApp4Publishing",
   "deploy": true,
   "test": false,
   "archiveURL": "link/to/dropbox/archive"
}
```

interconnected collections of documents, for optimization purposes we also use a separate collection for storing the transformations and maintaining the references to respective applications. Note, that having denormalized data is common in non-relational databases and some transformation-related information is also stored in the collection related to applications to speed up the querying process.

During the processing of a publishing request several steps are completed as described in Section 3.5. Firstly, if remote dependencies are specified then the materialization and update of the application package specification happen. Next, the respective entries in database collections are created based on the application package specification but including additional information such as generated transformations' signatures or numbers of dependencies. We do not demonstrate a complete MongoDB document's structure of the application as it is similar to the conceptual model discussed in Section 3.3. However, we focus on some details by showing the excerpts of the document's model. Listing 4.4 shows how the general information about the application taken from the application package specification is enriched with additional data like information about the file system's path, providers, validation status, and other statistical information. The providers are specified

**Listing 4.4** Specification of the general information about the application

```
...
"appInfo": {
   "appID": "dummyID",
   "appName": "dummyApp",
   ...
   "tags": ["tag1", "tag2"],
   "validated": false,
   "path": "path/to/app/archive",
   "providers": [
      {
         "providerQName": "TM1",
         "pkgID": "some Docker Image ID"
      }
   ],
   "transformationsCount": 0,
   "envDepsCount": 0,
   "softDepsCount": 0,
   "fileDepsCount": 0
}, ...
```

using abstract identifiers and Docker image identifiers. The repository does not keep the information about task managers assuming that routing is performed by request routers

which can substitute abstract identifiers with the real ones when needed. And responses with providers' abstract identifiers can be passed directly to consumers without a risk to provide sensitive information about internal endpoints.

Afterwards, the provisioning-ready specification, i.e. a Dockerfile, needs to be generated based on the application package specification. The resulting set of files and specifications has to be archived and stored on the file system. The copies of schema files, sample inputs and outputs from test run specification are also left outside the archive to allow analyzing them via GUI. Additionally, based on the publishing scenario provisioning and testing might happen. After everything is finished, the respective information in the database has to be updated and a successful response is returned with the JSON representation of the application. Commonly, the Dockerfile is constructed using some image as its basis [Nic16]. This defines which OS and potentially other software will be used. One important issue is to use a proper OS which will be compatible with the specified dependencies and especially the commands which use specific package managers like *apt-get*[9]. In the prototype we use the *ubuntu:14.04* base image for generation of Dockerfiles. This fact also leads to limitations like usage of apt-get as a package manager. However, the conceptual model of the data transformation application can be extended to support the specification of a certain OS as a dependency and including, e.g. its exact version. The Dockerfiles' syntax is very concise and consists of short instructions, e.g. FROM, RUN, or COPY. The main task is to map the required information from application package specification to Dockerfile instructions and based on these mappings to generate the actual Dockerfile. For instance, the publisher's information is used to specify the image maintainer with the respective *LABEL maintainer="John Doe"* instruction. We do not include the full description of all instructions related to creation of predefined folders for storing application's artifacts as well as instructions related to setting file permissions. Instead, as an example we describe how the folder for application's files is created and the correct permissions are set. Consider the excerpt from a Dockerfile shown in Listing 4.5. Firstly, a corresponding environment

**Listing 4.5** Example of Dockerfile instructions for copying application's files and setting permissions to execute them

```
FROM ubuntu:14.04
LABEL maintainer="JohnDoe@doe.com"
ENV APPHOME /app
RUN mkdir ${APPHOME}
COPY app ${APPHOME}
WORKDIR ${APPHOME}
RUN chmod -R a+x *
...
```

variable is generated with the help of the *ENV* instruction and a respective directory is created using *RUN mkdir $APPHOME*. Next, the application's files are copied using *COPY* instruction into the predefined folder in provisioning-ready package's structure described

[9] Canonical Ltd., AptGet: https://help.ubuntu.com/community/AptGet/Howto

in Section 3.3. The last steps are about changing the working directory using *WORKDIR* instruction and setting the file permissions, e.g. *RUN chmod -R a+x **.

In some cases it is a direct 1:1 mapping, whereas in other several instructions have to be used. Table 4.4 describes how particular information from the application package specification is mapped to Dockerfile instructions. Basically, all files provided along with the

| Application's Information | Docker Instruction | Description |
|---|---|---|
| Publisher | **LABEL** | As the publisher is responsible for the application, the value Author in metadata will contain the information about publisher |
| Application files | **COPY** | Application's files need to be copied into a respective predefined folder |
| Environment Dependency | **ENV** | The values of environment dependencies defined in application package specification are set using this instruction |
| Software Dependency | **COPY** | If provided, software dependency's files need to be copied into a respective folder relative to the predefined software dependencies folder |
| | **RUN** | The set of specified installation commands have to be executed |
| File Dependency | **COPY** | File dependencies need to be copied into a respective folder relative to the predefined file dependencies folder |
| Configuration | **RUN** | Specified configuration commands need to be executed |
| TestRun | **COPY** | Specified sample inputs and outputs need to be copied into the predefined testrun folder |

**Table 4.4:** Mapping of the application's information to Dockerfile's instructions

application such as software or file dependencies and sample inputs and outputs in test run specifications have to be copied using the COPY instruction. The environment dependencies can be set using the ENV instruction. Any specified command needs to be executed using the RUN instruction. As commands are provided in arrays, the generator loops through items creating a RUN instruction which executes multiple commands separated by semicolon. An important point is to use exactly one RUN instruction for semantically-related commands.

Essentially, when a RUN instruction is executed a new instance of a shell is created. Hence, running two connected commands, e.g. open a folder and rename a file in this folder, using different RUN instructions will result in an error. We assume that the order of items in the array is correctly specified by the publisher, hence this order is used for ordering the generated instructions. A similar approach is used for the overall ordering of dependency-related instructions in the Dockerfile. The overall generation process consists of several steps: parsing the application package specification, iterating over the groups of entities and mapping the information into Dockerfile instructions. In general, the main idea is to copy all application's artifacts inside respective predefined folders and run the specified commands.

## 4.5 Requesting a Transformation

After finding a suitable transformation, the consumer needs to send a HTTP POST request to the */tasks* resource. Essentially, this request consists of the application and transformation identifiers, providers information and specification of inputs including the access links. In case the application was found via searching the repository, the provider information is available as it can be taken directly from the response. However, if the consumer issues the request knowing only the application's details the provider information can be omitted. In such case the default provider is used in the prototype. Listing 4.6 shows the overall structure of a transformation request. As was discussed in Section 3.6, inputs can

**Listing 4.6** Structure of the transformation request

```
{
   "appID": "string",
   "transformationID": "string",
   "resultsEndpoint": "string",
   "providers": [ ... ],
   "inputParams": [ ... ],
   "inputFiles": [
      { "format": "string", "link": "string" }
   ],
   "inputFileSets": [
      { "format": "string", "count": 0, "linkToArchive": "string" }
   ]
}
```

be provided in a push or pull-based manner. In the prototype we use a combination of both. Input parameters are included inside the request, whereas file-based input types are provided as links. This approach has several advantages. Firstly, the consumer does not need to upload files multiple times in case the same or similar requests have to be issued. Secondly, this approach fits nicely into the picture of communication with the TraDE Middleware which already has a data model consisting of cross-partner data objects and their data elements which can be accessed via links. After receiving the request, the inputs are downloaded into the corresponding temporary folders created for the transformation

task. The process of handling transformation tasks might take a long time, thus it is more convenient to implement an asynchronous communication. For this reason the request has to include an information about respective endpoints for providing the output results which can either be pushed to the consumer or pulled by the consumer. In the latter case the endpoint is used for sending a notification that the task is completed. We choose to use the push-based approach as it also requires less effort from the TraDE Middleware's side as it does not need to implement polling. When the task is finished, the results are uploaded to the given address.

# 5 Case Study

In this chapter we demonstrate how the introduced concepts can be applied to a real world example. More specifically, we analyze how parts of the motivational example discussed in Section 1.2.6 can be simplified by modeling them implicitly as data transformations and invoking them using the introduced framework for handling data transformation applications.

Recall the choreography modeling example shown in Figure 1.2 from Section 1.2.6 describing the KMC simulation using the OPAL software. From the conceptual point of view, one specific part of this choreography is not a part of the actual simulation process, but rather changing the representation of existing data values. Figure 5.1 highlights the participant *OpalVisual* which transforms the snapshots data into an .mp4 video and creates a plot of the saturation data. The results of its execution do not influence the simulation and

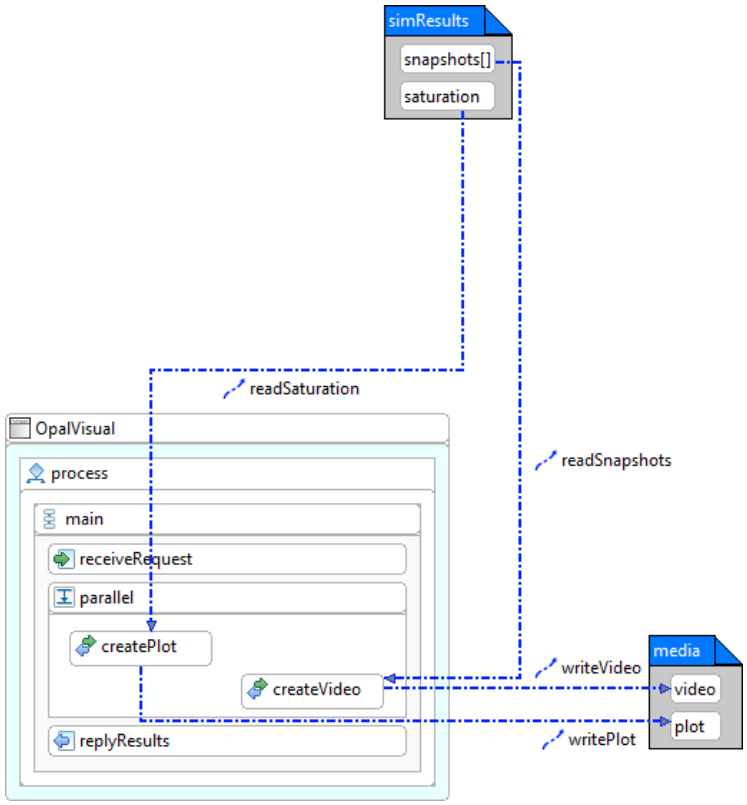


**Figure 5.1:** A highlighted OpalVisual participant in the OPAL simulation choreography that is solely responsible for the data transformation

even if this participant will be removed from the choreography the simulation can still be considered successfully completed. In fact, when following this explicit modeling approach if other representations are required then either the *OpalVisual* participant needs to be extended with additional activities or another participant has to be created depending on the representation's semantics, e.g. whether it is a visualization or not. This is a perfectly valid approach which, however, leads to the increase of visual clutter in the choreography model making it less readable and understandable.

Switching to an implicit modeling of these transformations might help to make the model more concise without making it less comprehensive. However, this approach implies having means to represent a transformation in the model without using the notions of participants and activities. In TraDE modeling concepts, the connectors between data elements represent the data flow. One way to provide such means is to enhance the notion of the data flow by introducing different types of connectors. For instance, if a data element belonging to some cross-partner data object in order to be transferred to its next destination has to be transformed a special type of connector can be used. Figure 5.2 shows an example of enhanced data flow connectors linking respective data elements belonging to "simResults" and "media" cross-partner data objects. The representation of snapshots data is transformed into a video and a diagram of saturation data is generated. The *OpalVisual* participant is removed from the model making it less verbose. To provide a reader with the

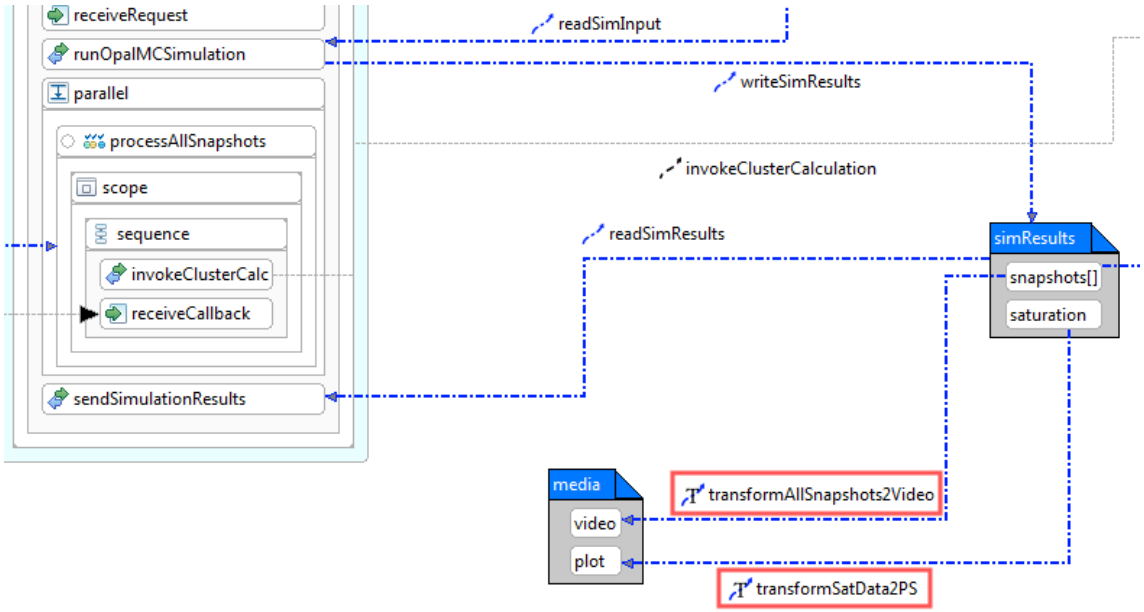


**Figure 5.2:** The implicit data transformation modeling using enhanced data flow connectors

better insight, Figure 5.3 demonstrates the complete OPAL choreography with implicit data transformation modeling applied. We focus on the *opal3dAnimatedLoopFile* application responsible for the transformation of snapshots into a video (transformAllSnapshots2Video data flow in Figure 5.2) due to its more complex organization which allows us to demonstrate more aspects of the process. This application is written in Python programming

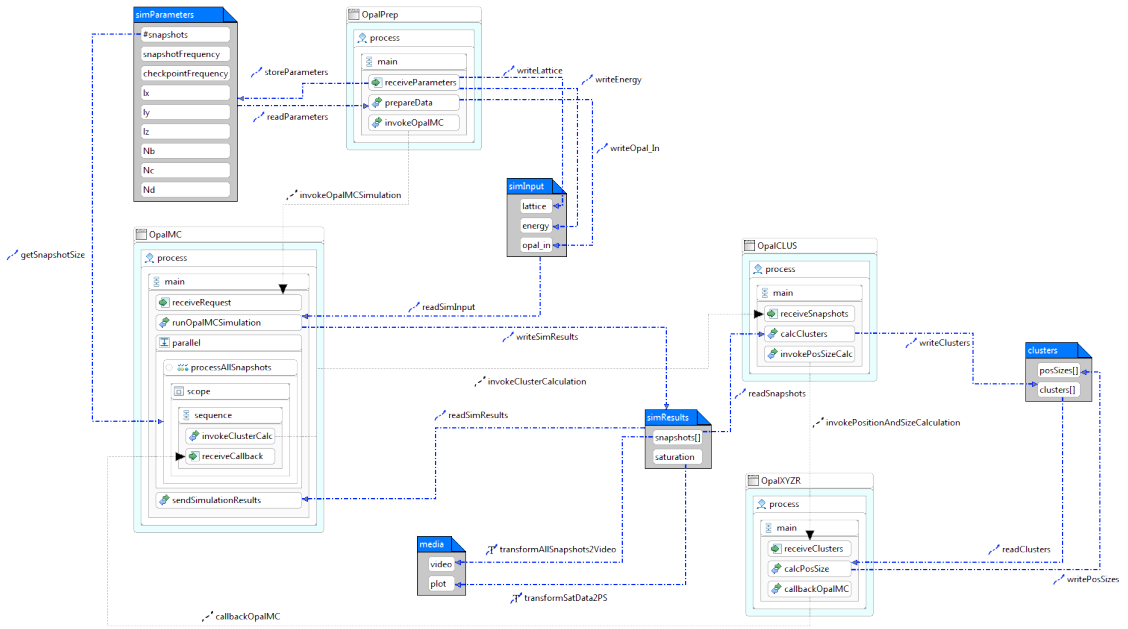


**Figure 5.3:** The modified OPAL simulation choreography with the implicit data transformation modeling

language and relies on several external modules including: (i.) numpy[1], a scientific computing package for Python; (ii.) matplotlib[2], a plotting library; (iii.) and ffmpeg[3], a multimedia framework for working with audio and video.

First, we need to create an application package specification for the *opal3dAnimatedLoopFile* application. Listing 5.1 shows the excerpt from the specification with the application's description. The next step is to describe the supported transformations. In this case, the

**Listing 5.1** The specification of general information about the application

```
"appInfo": {
   "appName": "opal3dAnimatedLoopFile",
   "appVersion": "1.0.0",
   "appPublisher": "Michael Hahn",
   "appDesc": "Transformation of the snapshots into video",
   "appDevelopers": ["Michael Hahn"],
   "appLicense": "Apache License 2.0",
   "tags": [ "opal", "simulation", "video"]
}
```

application supports only one transformation which takes two mandatory input parameters and transforms a set of .dat files into an .mp4 video. One important thing is that the file names are not used in the invocation command and the application expects them to be

[1]NumPy developers, NumPy: http://www.numpy.org/index.html
[2]The Matplotlib development team, Matplotlib: http://matplotlib.org/
[3]FFmpeg team, FFmpeg: https://ffmpeg.org

found next to the executable. The first parameter specifies a string prefix which is used in files' names. The second defines how many files will be used for the transformation. As a result of the transformation, one video file is produced next to the executable. Additionally, the name and qname specifications need to be provided. Listing 5.2 shows the specification of the transformation for *opal3dAnimatedLoopFile*. The input files are modeled as the

**Listing 5.2** The application's transformation specification

```
"transformations": [
   {
      "name": "snapshots-to-video",
      "qname": "simtech.opal.snapshots2video",
      "inputParams": [
         {
            "inputName": "prefixName",
            "alias": "$prefixName",
            "isOptional": false,
            "type": "string"
         },
         {
            "inputName": "numberOfFilesToAnimate",
            "alias": "$numberOfFilesToAnimate",
            "isOptional": false,
            "type": "integer"
         },
      ],
      "inputFileSets": [
         {
            "inputName": "snapshots",
            "alias": "$inputsShapshots",
            "isOptional": false,
            "format": "text/plain",
            "requiredPath": "{r}/",
            "fileSetSize": "$numberOfFilesToAnimate"
         }
      ],
      "outputFiles": [
         {
            "outputName": "opalClusterSnapshotsVideo",
            "alias": "$outputVideo",
            "format": "video/mp4",
            "accessPath": "{r}/"
         }
      ]
   }
]
```

InputFileSet. In the prototype we use the {*r*} qualifier in front of the path string to specify that the rest of the string is the path relative to the application's folder. Another option to distinguish between relative and absolute paths is to extend the model by adding new properties.

We omit the discussion of a complete specification of the dependencies due to its size and overall similarity of the process. One thing to notice is that we need to include any additional dependency apart from the aforementioned modules. For instance, the proper Python distribution is itself a dependency. Listing 5.3 shows how the aforementioned

**Listing 5.3** The specification of the application's dependencies

```
"dependencies": {
   "softDeps": [
      ...,
      {
         "depName": "numpy",
         "alias": "$numpy",
         "depDesc": "fundamental package for scientific computing with Python",
         "depVersion": ">1.13.0",
         "commands": ["sudo apt-get -y install python-numpy"]
      },
      {
         "depName": "matplotlib",
         "alias": "$matplotlib",
         "depDesc": "plotting library for the Python programming language",
         "depVersion": ">2.0.0",
         "commands": ["sudo apt-get -y install python-matplotlib"]
      },
      {
         "depName": "ffmpeg",
         "alias": "$ffmpeg",
         "depDesc": "Install FFmpeg storing animated plots as videos",
         "depVersion": ">3.0",
         "commands": [
            "sudo apt-get -y install ppa-purge software-properties-common",
            "sudo ppa-purge -y ppa:mc3man/trusty-media",
            "sudo add-apt-repository -y ppa:mc3man/trusty-media",
            "sudo apt-get update",
            "sudo apt-get -y install ffmpeg"
         ]
      }
   ]
}
```

external modules and libraries are specified. In the prototype we use the version string only as an additional information which has no influence on the execution of commands. All dependencies are installed from remote repositories meaning that no dependency-related files are provided. The remaining part is to specify the invocation. A command to invoke this application requires only two input parameters and is shown in Listing 5.4.

As a next step, the Dockerfile is generated based on the application package specification relying on the previously discussed mappings. Listing 5.5 demonstrates the generated Dockerfile. It consists of instructions specifying the general application information and creation of system folders. Moreover, there is a separate RUN instruction for every dependency, and instructions for setting the working directory to the folder containing the application's executable and specification of a default command.

**Listing 5.4** The specification of the application's dependencies

```
"invocations": {
   "invocationsCLI": [ {
      "invName": "opal3dAnimatedLoopFile CLI invocation",
      "invDesc": "Specify the prefix of the snapshots and the number of snapshots to use",
      "command": "python opal3dAnimatedLoopFile.py $prefixName $numberOfFilesToAnimate"
   } ]
}
```

**Listing 5.5** The resulting Dockerfile generated based on the application package specification

```
FROM ubuntu:14.04
LABEL maintainer="Michael Hahn"
#system folders
ENV APPHOME /app
COPY app ${APPHOME}
WORKDIR ${APPHOME}
RUN chmod -R a+x *
#dependencies
...
#depName: numpy
RUN sudo apt-get -y install python-numpy
#depName: matplotlib
RUN sudo apt-get -y install python-matplotlib
#depName: ffmpeg
RUN sudo apt-get -y install ppa-purge software-properties-common;sudo ppa-purge -y
     ppa:mc3man/trusty-media;sudo add-apt-repository -y ppa:mc3man/trusty-media;sudo
     apt-get update;sudo apt-get -y install ffmpeg
#Set working directory and default command
WORKDIR ${APPHOME}
CMD ["/sbin/init"]
```

One aspect we want to focus on is the default command's definition. According to official recommendations [Doc17a], Docker containers should be as clean and modular as possible. The more strict rule is to have one process per container meaning that it is dedicated to a specific application. While this approach suits our needs, we cannot directly use the invocation command as the entry point for applications. Before running the application, inputs need to be copied into required locations inside the container to be able to execute the application. This can happen when the container is running which leads us to the requirement to run the container without actually running the application. Another possibility is to use Docker volumes, i.e. run containers with the temporary volumes attached. For the prototype, we use the "/sbin/init" command which is the equivalent of only running the operating system inside the container. After the image is created using the Dockerfile, it can be used for the invocation of applications using the inputs obtained from transformation requests. We use pull-based approach of getting the inputs based on the links provided in the request. As soon as inputs are downloaded and mapped to the transformation's inputs, the copying into respective locations inside the container takes place. After these preparations are finished the application is invoked and the resulting outputs are pushed to the TraDE Middleware.

# 6 Conclusion and Future Work

In general, computer-based experiments are complex due to multiple factors including heterogeneity of scientific software and its dependencies, diverse environments, needs to wrap the software, and complicated configuration processes. While service choreographies help to reduce this complexity by structuring the communication processes and the TraDE Middleware adds the flexibility by introducing the data flow transparency, the process still remains generally complex. Furthermore, the technical aspects of the data flow such as data transformation of its elements along the path could add unnecessary complexity to models of experiments. The issue of orchestrating computer-based experiments is solved by workflow enactment systems relying on powerful yet technically complex orchestrating languages like BPEL or BPMN. On the other hand, an explicit modeling and executing the data transformation applications independently for every involved participant is not practical as it requires to manually wrap them as web services and implies that the modeler has expertise in orchestrating languages. Having a way to handle data transformation applications in a standardized manner can simplify the process of modeling and executing scientific service choreographies with the TraDE concepts applied. Moreover, the models without additional visual clutter are easier to understand. As we have shown in Chapter 5, an implicit modeling of data transformations in the context of TraDE Middleware makes service choreographies more concise without loosing their meaning.

In this thesis we conceptualize a framework which allows addressing this problem and provide its prototypical implementation. In Chapter 2 we discuss the background concepts and research topics which are relevant for this research. Next, in Chapter 3 we introduce the concepts for handling the data transformation applications and design the framework supporting these concepts. We first start with a set of assumptions underlying our work, focusing on the black-box approach towards data transformation applications which do not require user interaction and use files as the main unit of interchange. We then discuss various interaction models between the actors involved in the process of interaction with such applications. Next, we derive the extensible conceptual model of data transformation applications and describe the application packaging details. As a starting point of the framework design, we present a generic framework which we then gradually improve by refining its parts. This process results in the unification of concepts in the refined framework. As the last parts we explain the interaction scenarios in more details and discuss how the framework can be integrated with the TraDE Middleware. Finally, Chapter 4 describes the details of the framework's prototypical implementation. The already mentioned case study discussed in Chapter 5 analyzes a real world use case and discusses how the presented concepts can be applied to one of its parts. In the next couple of paragraphs we discuss the possible directions and concepts for future work.

**General Ideas** The concepts presented in this thesis are made extensible in order to support future modifications and enhancements. While we mostly discussed the file-based communication, other input types might be added into the conceptual model thus requiring improvements in every related component. Moreover, we discussed only the CLI-based invocation and adding other types of invocations is another interesting modification. Likewise, the data transformation software requiring user interaction is another kind of applications which might be considered for adding into the prototype. In general, the publishing process can be improved by supporting different options such as uploading files separately in a two-step publishing process, e.g. in combination with the authorization. The API can be more fine-grained to support partial application updates and versioning. Any scenarios involving file transfer can leverage from streaming. This might be particularly helpful in cases when large files are used in data transformations.

**Repository Enhancements** Several aspects related to the repository can be improved. For instance, more complex search implementation can be introduced using dedicated search engines like Apache Solr[1] or Elasticsearch[2]. The composite transformations search is another interesting enhancement which requires a lot of effort not only from the repository's perspective, but also from the task manager's side. Additionally, an implementation of the repository's optimizer component might be useful in cases when a large number of applications is published. Discovering the associative rules for dependencies and creating separate base images which can be used for generation of the Dockerfiles might significantly reduce the image building times making the overall deployment more convenient.

One significant point is adding the support of other provisioning mechanisms which requires generating different provisioning specifications as well as modifying the structure of provisioning-ready application packages and corresponding metadata in the database. A particular example of supporting another provisioning technology is to implement the generation of TOSCA [OAS13] topologies based on application package specifications. This makes provisioning-ready packages platform-independent and allows using the OpenTOSCA [Bin+13] ecosystem for the automated provisioning of data transformation applications as well as the management of their life cycle. Furthermore, the ecosystem provides a GUI-based topology modeling software, Winery [Kop+13], which allows to graphically specify and visualize the topology (e.g., components and their dependencies) of an application.

**Task Manager Enhancements** As with the repository, adding the support for other provisioning engines requires having a corresponding deployers implemented. Another interesting direction is supporting the invocation of different kinds of applications, e.g. web services. In the presented concepts we focused mainly on automated transformations scenarios due to the need to integrate the framework with the TraDE Middleware. However,

[1] Apache Software Foundation, Apache Solr: https://lucene.apache.org/solr/
[2] Elasticsearch BV, Elasticsearch: https://www.elastic.co/

the GUI-based interaction methods are also an interesting approach especially for use cases when the framework is used independently of other systems.

# Bibliography


[ACD10] A. A. Almonaies, J. R. Cordy, T. R. Dean. “Legacy system evolution towards service-oriented architecture.” In: *International Workshop on SOA Migration and Evolution*. 2010, pp. 53–62 (cit. on p. 29).

[Alt+04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock. “Kepler: an extensible system for design and execution of scientific workflows.” In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE. 2004, pp. 423–424 (cit. on p. 14).

[ASV13] A. Afanasiev, O. Sukhoroslov, V. Voloshinov. “MathCloud: publication and reuse of scientific applications as RESTful web services.” In: *International Conference on Parallel Computing Technologies*. Springer. 2013, pp. 394–408 (cit. on pp. 36, 38).

[Bar47] M. S. Bartlett. “The use of transformations.” In: *Biometrics* 3.1 (1947), pp. 39–52 (cit. on p. 15).

[BBW08] B. Baliś, M. Bubak, M. Wegiel. “LGF: A flexible framework for exposing legacy codes as services.” In: *Future Generation Computer Systems* 24.7 (2008), pp. 711–719 (cit. on p. 37).

[Bel+15] P. Belmann, J. Dröge, A. Bremges, A. C. McHardy, A. Sczyrba, M. D. Barton. “Bioboxes: standardised containers for interchangeable bioinformatics software.” In: *Gigascience* 4.1 (2015), p. 47 (cit. on p. 46).

[Ber14] D. Bernstein. “Containers and cloud: From lxc to docker to kubernetes.” In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84 (cit. on pp. 40, 42).

[Bin+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. “OpenTOSCA - A Runtime for TOSCA-based Cloud Applications.” In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, 2013, pp. 692–695. DOI: 10.1007/978-3-642-45005-1_62 (cit. on p. 108).

[Bis+99] J. Bisbal, D. Lawless, B. Wu, J. Grimson. “Legacy information systems: Issues and directions.” In: *IEEE software* 16.5 (1999), pp. 103–111 (cit. on p. 28).

[BK16] R. Bawa, I. Kaur. “Algorithmic Approach for Efficient Retrieval of Component Repositories in Component based Software Engineering.” In: *Indian Journal of Science and Technology* 9.48 (2016) (cit. on pp. 27, 70).

[BN06] E. W. Biederman, L. Networx. “Multiple instances of the global linux namespaces.” In: *Proceedings of the Linux Symposium*. Vol. 1. 2006, pp. 101–112 (cit. on p. 41).

[Boe15] C. Boettiger. “An introduction to Docker for reproducible research.” In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79 (cit. on pp. 44, 57).

[BS03] P. Binkele, S. Schmauder. “An atomistic Monte Carlo simulation of precipitation in a binary system.” In: *Zeitschrift für Metallkunde* 94.8 (2003), pp. 858–863 (cit. on p. 16).

[Can+06] G. Canfora, A. R. Fasolino, G. Frattolillo, P. Tramontana. “Migrating interactive legacy systems to web services.” In: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. IEEE. 2006, 10–pp (cit. on p. 35).

[Can+08] G. Canfora, A. R. Fasolino, G. Frattolillo, P. Tramontana. “A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures.” In: *Journal of Systems and Software* 81.4 (2008), pp. 463–480 (cit. on p. 35).

[CD+00] S. Comella-Dorda, K. Wallnau, R. C. Seacord, J. Robert. *A survey of legacy system modernization approaches*. Tech. rep. Carnegie-Mellon univ pittsburgh pa Software engineering inst, 2000 (cit. on p. 29).

[Cha04] D. Chappell. *Enterprise Service Bus: Theory in Practice*. Theory in practice. O’Reilly Media, 2004. ISBN: 9781449391096 (cit. on p. 78).

[CS14] R. Chamberlain, J. Schommer. “Using Docker to support reproducible research.” In: *DOI: https://doi. org/10.6084/m9. figshare* 1101910 (2014) (cit. on p. 44).

[Dec+07] G. Decker, O. Kopp, F. Leymann, M. Weske. “BPEL4Chor: Extending BPEL for modeling choreographies.” In: *Web Services, 2007. ICWS 2007. IEEE International Conference on*. IEEE. 2007, pp. 296–303 (cit. on pp. 14, 15).

[Dec+09] G. Decker, O. Kopp, F. Leymann, M. Weske. “Interacting services: from specification to execution.” In: *Data & Knowledge Engineering* 68.10 (Apr. 2009), pp. 946–972 (cit. on p. 17).

[Dee+09] E. Deelman, D. Gannon, M. Shields, I. Taylor. “Workflows and e-Science: An overview of workflow system features and capabilities.” In: *Future generation computer systems* 25.5 (2009), pp. 528–540 (cit. on p. 14).

[Del+05] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, P. Kacsuk. “GEMLCA: Running legacy code applications as grid services.” In: *Journal of Grid Computing* 3.1-2 (2005), pp. 75–90 (cit. on p. 37).

[DK76] F. DeRemer, H. H. Kron. “Programming-in-the-large versus programming-in-the-small.” In: *IEEE Transactions on Software Engineering* 2 (1976), pp. 80–86 (cit. on p. 11).

[DKB08] G. Decker, O. Kopp, A. Barros. “An Introduction to Service Choreographies.” In: *Information Technology* 50.2 (2008), pp. 122–127. DOI: 10.1524/itit.2008.0473 (cit. on p. 14).

[DPP05] C. Diamantini, D. Potena, M. Panti. “Wrapping Legacy Code for a Service Oriented Knowledge Discovery Support system.” In: *Proc. of IADIS Int. conf. WWW/Internet 2005*. 2005, pp. 19–22 (cit. on p. 36).

[DRK14] R. Dua, A. R. Raja, D. Kakadia. “Virtualization vs containerization to support paas.” In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE. 2014, pp. 610–614 (cit. on p. 43).

[DSM16] M. Dabhade, S. Suryawanshi, R Manjula. “A systematic review of software reuse using domain engineering paradigms.” In: *Green Engineering and Technologies (IC-GET), 2016 Online International Conference on*. IEEE. 2016, pp. 1–6 (cit. on p. 26).

[EE01] H. Enderton, H. Enderton. *A Mathematical Introduction to Logic*. Elsevier Science, 2001. ISBN: 9780080496467 (cit. on p. 67).

[Elk16] M. M. Elkhouly. “Software Component as a Service (SCaaS).” In: *Australian Journal of Basic and Applied Sciences* 10.16 (2016), pp. 45–51 (cit. on p. 38).

[Erl16] T. Erl. *Service-oriented Architecture: Analysis and Design for Services and Microservices*. The Prentice Hall Service Technology Series from Thomas Erl Series. Prentice Hall, 2016. ISBN: 9780133858587 (cit. on p. 26).

[eSc17] I. I. C. on eScience. *IEEE International Conference on eScience*. https://escience-conference.org. 2017 (cit. on p. 14).

[Fel+15] W. Felter, A. Ferreira, R. Rajamony, J. Rubio. “An updated performance comparison of virtual machines and linux containers.” In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE. 2015, pp. 171–172 (cit. on pp. 40, 42, 43).

[Fin09] E. L. Fink. “The FAQs on data transformation.” In: *Communication Monographs* 76.4 (2009), pp. 379–397 (cit. on p. 15).

[Fou17] C. Foundry. *Warden Documentation*. http://docs.cloudfoundry.org/concepts/architecture/warden.html. 2017 (cit. on p. 42).

[Fow10a] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 9780131392809 (cit. on p. 93).

[Fow10b] M. Fowler. *Richardson Maturity Model*. 2010. URL: https://www.martinfowler.com/articles/richardsonMaturityModel.html (cit. on pp. 90, 92).

[FSB16] T. C. Fanelli, S. C. Simons, S. Banerjee. “A Systematic Framework for Modernizing Legacy Application Systems.” In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 678–682 (cit. on p. 29).

[Gla+08] T. Glatard, J. Montagnat, D. Emsellem, D. Lingrand. “A Service-Oriented Architecture enabling dynamic service grouping for optimizing distributed workflow execution.” In: *Future Generation Computer Systems* 24.7 (2008), pp. 720–730 (cit. on p. 37).

[GLH15] S. García, J. Luengo, F. Herrera. *Data preprocessing in data mining*. Springer, 2015 (cit. on p. 15).

[GML00] G. C. Gannod, S. V. Mudiam, T. E. Lindquist. “An architecture-based approach for synthesizing and integrating adapters for legacy software.” In: *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE. 2000, pp. 128–137 (cit. on p. 32).

[GMW10] D. Garlan, R. Monroe, D. Wile. “Acme: An architecture description interchange language.” In: *CASCON First Decade High Impact Papers*. IBM Corp. 2010, pp. 159–173 (cit. on p. 32).

[Gol73] R. P. Goldberg. “Architecture of virtual machines.” In: *Proceedings of the workshop on virtual computer systems*. ACM. 1973, pp. 74–112 (cit. on p. 40).

[Guo+05] H. Guo, C. Guo, F. Chen, H. Yang. “Wrapping client-server application to Web services for Internet computing.” In: *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*. IEEE. 2005, pp. 366–370 (cit. on p. 35).

[Hah+17a] M. Hahn, U. Breitenbücher, O. Kopp, F. Leymann. “Modeling and execution of data-aware choreographies: an overview.” In: *Computer Science - Research and Development* (2017). ISSN: 1865-2042. DOI: 10.1007/s00450-017-0387-y (cit. on pp. 16–18).

[Hah+17b] M. Hahn, U. Breitenbücher, F. Leymann, A. Weiß. “TraDE - A Transparent Data Exchange Middleware for Service Choreographies.” In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*. Vol. 10573. Lecture Notes in Computer Science. Cham: Springer International Publishing AG, Oct. 2017, pp. 252–270. ISBN: 978-3-319-69462-7. DOI: 10.1007/978-3-319-69462-7_16 (cit. on pp. 21–23).

[HKL16] M. Hahn, D. Karastoyanova, F. Leymann. “A Management Life Cycle for Data-Aware Service Choreographies.” In: *Proceedings of the 23rd International Conference on Web Services (ICWS 2016)*. IEEE Computer Society, 2016, pp. 364–371. DOI: 10.1109/ICWS.2016.54 (cit. on p. 15).

[HM84] E. Horowitz, J. B. Munson. “An expansive view of reusable software.” In: *IEEE Transactions on Software Engineering* 5 (1984), pp. 477–487 (cit. on p. 24).

[Hos+16] A. Hosny, P. Vera-Licona, R. Laubenbacher, T. Favre. “AlgoRun: a Docker-based packaging system for platform-agnostic implemented algorithms.” In: *Bioinformatics* 32.15 (2016), pp. 2396–2398 (cit. on pp. 45, 53).

[HS95] M. A. Hernández, S. J. Stolfo. “The merge/purge problem for large databases.” In: *ACM Sigmod Record*. Vol. 24. 2. ACM. 1995, pp. 127–138 (cit. on p. 74).

[HS98] M. A. Hernández, S. J. Stolfo. “Real-world data is dirty: Data cleansing and the merge/purge problem.” In: *Data mining and knowledge discovery* 2.1 (1998), pp. 9–37 (cit. on p. 74).

[HT02] H. Haddad, H. Tesser. “Reusable subsystems: domain-based approach.” In: *Proceedings of the 2002 ACM symposium on Applied computing*. ACM. 2002, pp. 971–975 (cit. on p. 30).

[HT03] A. J. Hey, A. E. Trefethen. “The data deluge: An e-science perspective.” In: (2003) (cit. on p. 13).

[Hua+03] Y. Huang, I. Taylor, D. W. Walker, R. Davies. “Wrapping legacy codes for grid-based applications.” In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003, 7–pp (cit. on pp. 36, 37).

[HW12] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN: 9780133065107 (cit. on p. 86).

[HX06] H. M. Haddad, Y. Xie. “Wrapper-based framework for domain-specific software reuse.” In: *Journal of information science and engineering* 22.2 (2006), p. 269 (cit. on pp. 31, 32).

[Ini17] O. A. Initiative. *Open API Initiative (OAI)*. https://www.openapis.org/. 2017 (cit. on p. 89).

[Jan07] N. W. Jankowski. “Exploring e-science: an introduction.” In: *Journal of Computer-Mediated Communication* 12.2 (2007), pp. 549–562 (cit. on p. 13).

[Joh01] G. Johnson. *The World: In Silica Fertilization; All Science Is Computer Science*. Ed. by T. N. Y. Times. [Online; posted 25-March-2001]. 2001. URL: http://www.nytimes.com/2001/03/25/weekinreview/the-world-in-silica-fertilization-all-science-is-computer-science.html (cit. on p. 13).

[Joy15] A. M. Joy. “Performance comparison between linux containers and virtual machines.” In: *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. IEEE. 2015, pp. 342–346 (cit. on p. 43).

[Juh+09] E. Juhnke, D. Seiler, T. Stadelmann, T. Dörnemann, B. Freisleben. “LCDL: an extensible framework for wrapping legacy code.” In: *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*. ACM. 2009, pp. 648–652 (cit. on pp. 30, 53).

[Kim+17] B. Kim, T. A. Ali, C. Lijeron, E. Afgan, K. Krampis. “Bio-Docklets: Virtualization Containers for Single-Step Execution of NGS Pipelines.” In: *bioRxiv* (2017), p. 116962 (cit. on p. 46).

[Kiv+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori. “kvm: the Linux virtual machine monitor.” In: *Proceedings of the Linux symposium*. Vol. 1. 2007, pp. 225–230 (cit. on p. 43).

[Kop+13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery - A Modeling Tool for TOSCA-based Cloud Applications.” In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, 2013, pp. 700–704. DOI: 10.1007/978-3-642-45005-1_64 (cit. on p. 108).

[KSB17] G. M. Kurtzer, V. Sochat, M. W. Bauer. “Singularity: Scientific containers for mobility of compute.” In: *PloS one* 12.5 (2017), e0177459 (cit. on p. 45).

[LCJ03] B.-J. Lee, T. Choi, T. Jeong. “X-CLI: CLI-based management architecture using XML.” In: *Integrated Network Management VIII*. Springer, 2003, pp. 477–480 (cit. on p. 32).

[Lin17] LinuxContainers.org. *LXC Documentation*. https://linuxcontainers.org/lxc/documentation/. 2017 (cit. on p. 42).

[LK15] W. Li, A. Kanso. “Comparing containers versus virtual machines for achieving high availability.” In: *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE. 2015, pp. 353–358 (cit. on p. 43).

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000. ISBN: 9780130217530 (cit. on p. 14).

[LZ16] D. Liu, C. Zhang. “Comparison and analysis of methods for migrating legacy systems to SOA.” In: *MATEC Web of Conferences*. Vol. 77. EDP Sciences. 2016, p. 04005 (cit. on p. 29).

[Mas11] M. Masse. *REST API Design Rulebook*. Oreilly and Associate Series. O’Reilly Media, 2011. ISBN: 9781449310509 (cit. on p. 90).

[MC07] P. Mohagheghi, R. Conradi. “Quality, productivity and economic benefits of software reuse: a review of industrial studies.” In: *Empirical Software Engineering* 12.5 (2007), pp. 471–516 (cit. on p. 25).

[McI68] D. M. McIlroy. “Mass-Produced Software Components.” In: *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*. Ed. by J. M. Buxton, P. Naur, B. Randell. NATO Science Committee, 1968, pp. 138–155 (cit. on pp. 23, 24).

[Men+17] P. Menage et al. *CGROUPS*. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. 2017 (cit. on p. 41).

[MKK15] R. Morabito, J. Kjällman, M. Komu. “Hypervisors vs. lightweight virtualization: a performance comparison.” In: *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE. 2015, pp. 386–393 (cit. on pp. 41, 43).

[MMK+94] H. Mili, O. Marcotte, A. Kabbaj, et al. “Intelligent component retrieval for software reuse.” In: *Proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering*. 1994, pp. 101–114 (cit. on pp. 27, 72, 73).

[MMM95] H. Mili, F. Mili, A. Mili. “Reusing software: Issues and research directions.” In: *IEEE transactions on Software Engineering* 21.6 (1995), pp. 528–562 (cit. on pp. 24, 26).

[Mor15] R. Morabito. “Power consumption of virtualization technologies: an empirical investigation.” In: *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE. 2015, pp. 522–527 (cit. on p. 43).

[MS70] R. A. Meyer, L. H. Seawright. “A virtual machine time-sharing system.” In: *IBM Systems Journal* 9.3 (1970), pp. 199–218 (cit. on p. 40).

[MSM+15] F. Moreews, O. Sallou, H. Ménager, et al. “BioShaDock: a community driven bioinformatics shared Docker-based tools registry.” In: *F1000Research* 4 (2015) (cit. on p. 46).

[Mul13] B. Mulloy. *Web API design*. 2013. URL: https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf (cit. on p. 90).

[NEO03] H. B. Newman, M. H. Ellisman, J. A. Orcutt. “Data-intensive e-science frontier research.” In: *Communications of the ACM* 46.11 (2003), pp. 68–77 (cit. on p. 13).

[Nic16] J. Nickoloff. *Docker in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2016. ISBN: 1633430235, 9781633430235 (cit. on p. 96).

[OAS06] OASIS. *Web Services Business Process Execution Language Version 2.0*. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html. 2006 (cit. on p. 14).

[PD93] R. Prieto-Díaz. “Status report: Software reusability.” In: *IEEE software* 10.3 (1993), pp. 61–66 (cit. on p. 24).

[PF16] S. R. Piccolo, M. B. Frampton. “Tools and techniques for computational reproducibility.” In: *GigaScience* 5.1 (2016), p. 30 (cit. on p. 44).

[PG74] G. J. Popek, R. P. Goldberg. “Formal requirements for virtualizable third generation architectures.” In: *Communications of the ACM* 17.7 (1974), pp. 412–421 (cit. on p. 40).

[PP93] A. Podgurski, L. Pierce. “Retrieving reusable software by sampling behavior.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pp. 286–303 (cit. on p. 26).

[PR16] R. Priedhorsky, T. C. Randles. “Charliecloud: Unprivileged containers for user-defined software stacks.” In: (2016) (cit. on p. 45).

[RD00] E. Rahm, H. H. Do. “Data cleaning: Problems and current approaches.” In: *IEEE Data Eng. Bull.* 23.4 (2000), pp. 3–13 (cit. on p. 15).

[RR03] T. Ravichandran, M. A. Rothenberger. “Software reuse strategies and component markets.” In: *Communications of the ACM* 46.8 (2003), pp. 109–114 (cit. on p. 25).

[RS16] S. Rochimah, A. S. Sankoh. “Migration of Existing or Legacy Software Systems into Web Service-based Architectures (Reengineering Process): A Systematic Literature Review.” In: *Migration* 133.3 (2016) (cit. on p. 29).

[SA14] O. Sukhoroslov, A. Afanasiev. “Everest: A Cloud Platform for Computational Web Services.” In: *CLOSER*. 2014, pp. 411–416 (cit. on p. 38).

[SB15] P. Sturgeon, L. Bohill. *Build APIs You Won’t Hate: Everyone and Their Dog Wants an API, So You Should Probably Learn How to Build Them*. Philip J. Sturgeon, 2015. ISBN: 9780692232699 (cit. on pp. 90, 92, 93).

[SERS02] E. Stroulia, M. El-Ramly, P. Sorenson. “From legacy to web through interaction modeling.” In: *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE. 2002, pp. 320–329 (cit. on p. 35).

[SGG07] J. Syed, M. Ghanem, Y. Guo. “Supporting scientific discovery processes in Discovery Net.” In: *Concurrency and Computation: Practice and Experience* 19.2 (2007), pp. 167–179 (cit. on p. 14).

[SK13] M. Sonntag, D. Karastoyanova. “Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows.” In: *Journal of Grid Computing* 11.3 (2013), pp. 553–583. ISSN: 1570-7873. DOI: 10.1007/s10723-013-9268-1. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2013-06&engl=0 (cit. on p. 14).

[Sne00] H. M. Sneed. “Encapsulation of legacy software: A technique for reusing legacy software components.” In: *Annals of Software Engineering* 9.1 (2000), pp. 293–313 (cit. on p. 29).

[Sne06] H. M. Sneed. “Integrating legacy software into a service oriented architecture.” In: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. IEEE. 2006, 11–pp (cit. on p. 34).

[Sne+06] H. M. Sneed et al. “Wrapping legacy software for reuse in a SOA.” In: *Multikonferenz Wirtschaftsinformatik*. Vol. 2. 2006, pp. 345–360 (cit. on p. 34).

[SNLM16] M. G. da Silva Neto, W. C. B. Lins, E. B. P. Mariz. “Services for Legacy Software Rejuvenation: A Systematic Mapping Study.” In: *ICSEA 2016* (2016), p. 197 (cit. on p. 29).

[SS03] H. M. Sneed, S. H. Sneed. “Creating web services from legacy host programs.” In: *Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE International Workshop on*. IEEE. 2003, pp. 59–65 (cit. on p. 34).

[SVA15] O. Sukhoroslov, S. Volkov, A. Afanasiev. “A web-based platform for publication and distributed execution of computing applications.” In: *Parallel and Distributed Computing (ISPDC), 2015 14th International Symposium on*. IEEE. 2015, pp. 175–184 (cit. on pp. 38, 39).

[SVIG07] R. Sears, C. Van Ingen, J. Gray. “To blob or not to blob: Large object storage in a database or a filesystem?” In: *arXiv preprint cs/0701168* (2007) (cit. on p. 67).

[Tay+05] I. Taylor, M. Shields, I. Wang, A. Harrison. “Visual grid workflow in Triana.” In: *Journal of Grid Computing* 3.3-4 (2005), pp. 153–169 (cit. on pp. 14, 36).

[TKR70] R. Tagliacozzo, M. Kochen, L. Rosenberg. “Orthographic error patterns of author names in catalog searches.” In: *Information Technology and Libraries* 3.2 (1970), pp. 93–101 (cit. on p. 68).

[Val+16] T. Vale, I. Crnkovic, E. S. de Almeida, P. A.d.M. S. Neto, Y. C. Cavalcanti, S. R. de Lemos Meira. “Twenty-eight years of component-based software engineering.” In: *Journal of Systems and Software* 111 (2016), pp. 128–148 (cit. on p. 26).

[WBL15] J. Wettinger, U. Breitenbücher, F. Leymann. “Streamlining APIfication by Generating APIs for Diverse Executables Using Any2API.” In: *International Conference on Cloud Computing and Services Science*. Springer. 2015, pp. 216–238 (cit. on pp. 32, 33).

[Wei+17] A. Weiß, V. Andrikopoulos, M. Hahn, D. Karastoyanova. “Model-as-you-go for Choreographies: Rewinding and Repeating Scientific Choreographies.” In: *IEEE Transactions on Services Computing* PP.99 (2017), pp. 1–1. DOI: 10.1109/TSC.2017.2732988. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2017-12&engl=0 (cit. on p. 15).

[Yam15] Y. Yamato. “OpenStack hypervisor, container and Baremetal servers performance comparison.” In: *IEICE Communications Express* 4.7 (2015), pp. 228–232 (cit. on p. 43).

[Zai+15] A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, A. Mavridis, T. Chaikalis, I. Deligiannis, P. Sfetsos, I. Stamelos. “An empirical study on the reuse of third-party libraries in open-source software development.” In: *Proceedings of the 7th Balkan Conference on Informatics Conference*. ACM. 2015, p. 4 (cit. on p. 25).

[Zdu02] U. Zdun. “Reengineering to the web: A reference architecture.” In: *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*. IEEE. 2002, pp. 164–173 (cit. on p. 34).

[ZW95] A. M. Zaremski, J. M. Wing. “Signature matching: a tool for using software libraries.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4.2 (1995), pp. 146–170 (cit. on pp. 27, 70).

[ZW97] A. M. Zaremski, J. M. Wing. “Specification matching of software components.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.4 (1997), pp. 333–369 (cit. on p. 27).

[Alg17] AlgoRun project. *AlgoRun v2.0 Documentation*. http://algorun.org/documentation. 2017 (cit. on p. 45).

[Clo17] Cloudius Systems. *OSv*. http://osv.io/. 2017 (cit. on p. 43).

[Cor17] CoreOS, Inc. *rkt overview*. https://coreos.com/rkt. 2017 (cit. on p. 42).

[cur17] curl project. *curl*. https://curl.haxx.se/. 2017 (cit. on p. 59).

[Doc17a] Docker, Inc. *Best practices for writing Dockerfiles*. 2017. URL: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices (cit. on p. 106).

[Doc17b] Docker, Inc. *What is Docker?* https://www.docker.com/what-docker. 2017 (cit. on pp. 42, 44, 45, 88).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html. 2013 (cit. on p. 108).

[Obj11] Object Management Group. *Business Process Model and Notation Version 2.0*. http://www.omg.org/spec/BPMN/2.0/. 2011 (cit. on p. 14).

[Obj17] Object Management Group. *CORBA*. http://www.corba.org/. 2017 (cit. on p. 25).

[Ope17] OpenStack Foundation. *OpenStack*. https://www.openstack.org/software/. 2017 (cit. on p. 43).

[Ora17a] Oracle Corporation. *MySQL*. https://www.mysql.com/. 2017 (cit. on p. 43).

[Ora17b] Oracle. *Enterprise JavaBeans Technology*. http://www.oracle.com/technetwork/java/javaee/ejb/index.html. 2017 (cit. on p. 25).

[Red17] Redis Labs. *Redis*. https://redis.io/. 2017 (cit. on p. 43).

[Uni17] University of Stuttgart. *Cluster of Excellence SimTech*. http://www.simtech.uni-stuttgart.de/en/index.html. 2017 (cit. on p. 15).

[Wor05] World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0*. https://www.w3.org/TR/ws-cdl-10/. 2005 (cit. on p. 14).

[Xen17] Xen Project. *Xen*. https://www.xenproject.org/. 2017 (cit. on p. 43).

All links were last followed on December 12, 2017.

# List of Figures

# List of Tables

# List of Listings

**Declaration**

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

---

place, date, signature