Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Placement of Application Components in Industrial Environments

Josip Ledić

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Karoline Saatkamp
Commenced:	December 12, 2018
Completed:	June 11, 2019

Acknowledgement

This work wouldn't have been possible without the help of a number of persons.

Firstly, I want to thank my supervisor Karoline Saatkamp for having enough patience with me and guiding me in the right direction whenever I lost focus.

I also want to thank Michael Wurster, Kálmán Képes and all other IAAS employees who never hesitated to give me technical support during the development of the prototype.

I really learned a lot during my time at the IAAS, which dates back to the beginning of my master's studies, being a former participant of an IAAS development project, having taken my technical specialization exams there, then being a student assistant and finally having written my thesis there as well. Therefore I thank all the people that gave me the opportunities to make these experiences.

Finally, I want to thank my family and friends who supported me throughout my time at university and gave me every reason to continue my journey whenever there were traces of doubt in my mind.

Abstract

Industrial companies are currently competing for higher degrees of automation and adaptability in their factories. In the course of the fourth industrial revolution factories have become smarter by the introduction of technologies such as cyber-physical systems, Cloud Computing, 5G, and the Internet of Things. With the introduction of systems into their factories that combine these technologies such as Driverless Transport Systems, industrial companies are confronted with a previously unknown web of complexity that emerges from the interconnectivity between the various application components of these systems. The amount of collected data and the need for data analysis in industrial environments grows steadily and results in a new accentuated role for the IT. Due to this trend, system architects are now often faced with large and heterogeneous environments when introducing a new system into an existing smart factory and have to look attentively at the way they embed new applications into the existing IT-infrastructure.

To reduce the cost in general, the goal is to reuse existing computation resources for new applications where possible. Finding optimal placement locations for multiple application components in a pool of resources that ranges from traditional options like on-device computing capabilities to emerging options like the edge cloud, is an optimization problem that is often described as the Application Component Placement (ACP) problem.

This thesis addresses these issues by presenting a conceptual approach to solving this problem for an industrial use case by making use of the concepts of the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard and extending the capabilities of the OpenTOSCA ecosystem by providing a prototypical implementation of an ACP-solving algorithm.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

ACP Application Component Placement. 18

DBMS Database Management System. 19

DTS Driverless Transport Systems. 17

IasC Infrastructure-as-Code. 25

SCP Set Cover Problem. 42

TOSCA Topology and Orchestration Specification for Cloud Applications. 26

1 Introduction

The first chapter introduces the motivation and goals behind this thesis presented in Section 1.1, followed by Section 1.2 on page 19 which outlines the structure of the following chapters in this work. Finally, a running example throughout this thesis and its related use cases are presented in Section 1.3 on page 19 and Section 1.4 on page 20, which are used to explain the conceptual approach presented in this work.

1.1 Motivation

We are currently in the midst of the Cloud Computing era which, in recent years has made storage and computing cheaper and more flexible for companies. Companies are transitioning their industrial facilities from a more traditional approach to a smart approach that makes use of emerging paradigms like the Internet of Things. Because the level of automation in industrial environments and therefore the amount of generated and collected data is rising, the number of applications in the IT-landscape of companies that analyze and process this data is growing as well. Such *Industry 4.0* environments with a high degree of automation and interconnectivity of systems rely on webs of application components for their realization. A convenient option is to host these applications in the enterprise cloud, if feasible. Nonetheless, many of these industrial applications in smart factories are latency sensitive and therefore have real-time requirements that can't be met by traditional cloud offerings and call for being hosted in close proximity to their place of usage i.e., the edge of the network.

In the case of a Driverless Transport Systems (DTS) in a smart factory for example, components for image processing, engine control, and sensor data need to interact with each other in an efficient manner even though they have different requirements and may be hosted in different locations. On top of that, additional components for the inventory management may interact with these components as well e.g., databases or components for the management of digital twins of items in the inventory.

Each of these components could theoretically be deployed on-device (a mobile compute unit, in the case of a DTS), in the edge cloud, in the private cloud or in the public cloud, but when considering the constraints of each component e.g., the latency-sensitivity of an engine control component, it becomes clear that these requirements can hardly be met by a machine hosted in a remote data center. A popular alternative in the past was to host such components on-device and to transfer any collected data to a company server in intervals rather than continuously in real-time.

But advancements in the research on 5^{th} Generation networks (5G) proposing speeds of *10Gbit/s* and latencies of no more than a few *ms* have made Edge Cloud Computing and computation offloading another viable option for applications with real-time requirements without having to rely on mobile on-device computing resources [YWJ+15], [GJF+16], [SCZ+16], [CJLF16]. Therefore, nowadays the goal has become to increase the reliability, flexibility and cost-efficiency of industrial

applications, as well as to increase the degree of automation of operating these applications as much as possible, by embracing the favorable properties of Cloud Computing, while not violating the individual technical constraints and requirements of individual applications.

Figure 1.1: The Continuum of Deployment Locations.

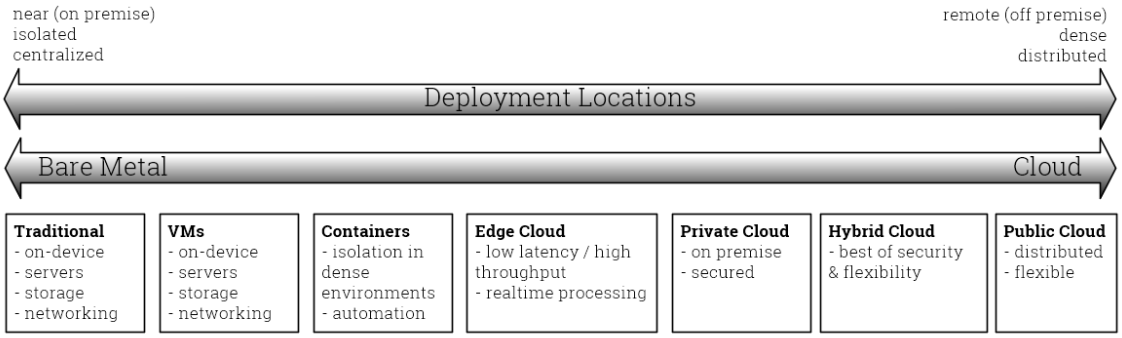


Figure 1.1 shows a continuum of deployment locations, inspired by [Mor18]. It shows the different possible deployment locations on a one-dimensional axis that combines the remoteness, density and distribution of each option. The first half demonstrates the increasing degree of virtualization while the second half demonstrates the Cloud Computing options from the edge of the network to remote data centers. It is fair to assume that the cost of the options depicted on the continuum decreases from left to right. This is mainly due to the higher start-up and maintenance costs of on-premise options compared to cloud offerings, but also due to secondary effects such as e.g., the increase of battery life of systems such as DTS, which can operate for longer periods of time when opting for computation offloading.

Naturally, companies want to make use of that, but the problem of finding optimal placement locations for each application component can be very challenging for humans as well as algorithms, especially post modeling phase. During the modeling phase of a smart factory i.e., when designing the whole infrastructure from scratch and considering all software and hardware components in advance, the problem of Application Component Placement (ACP) is easier to deal with. In reality, this case is not that common, as nowadays more and more industrial companies find themselves in a transitory phase from a traditional to a smart implementation of their factories, where providers of smart systems such as DTS are confronted with a heterogeneous, rigid and legacy IT-infrastructure and have to potentially reuse preexisting computation resources, and therefore install their systems and place their application components with regards to the unique idiosyncrasies of every other factory. Therefore, there is a need for a solution that can assist industrial users when confronting the problem of ACP and that lets them express preferences while doing that. This thesis presents such a concept of an algorithmic approach to the ACP that is implemented prototypically as part of the OpenTOSCA ecosystem.

1.2 Structure

The structure of this thesis is made up of six chapters.

The first chapter began with the motivation and is concluded by the upcoming sections that present the running example throughout this thesis followed by the different use cases this work revolves around.

Chapter 2 on page 25 introduces necessary technical fundamentals as well as background information on TOSCA and OpenTOSCA.

In Chapter 3 on page 35 the state of the art in the field of application placement and other related work is presented.

Chapter 4 on page 39 contains the concept for the placement of application components onto running instances that was developed as part of this thesis.

Chapter 5 on page 49 explains all the implementation details of the prototypical implementation of this concept along with the environment in which the prototype was tested initially.

Finally, Chapter 6 on page 65 concludes this work with a summary and an outlook.

1.3 Running Example

The main use case chosen for this thesis relates to the placement of the application components of a DTS in an industrial environment, such as a smart factory.

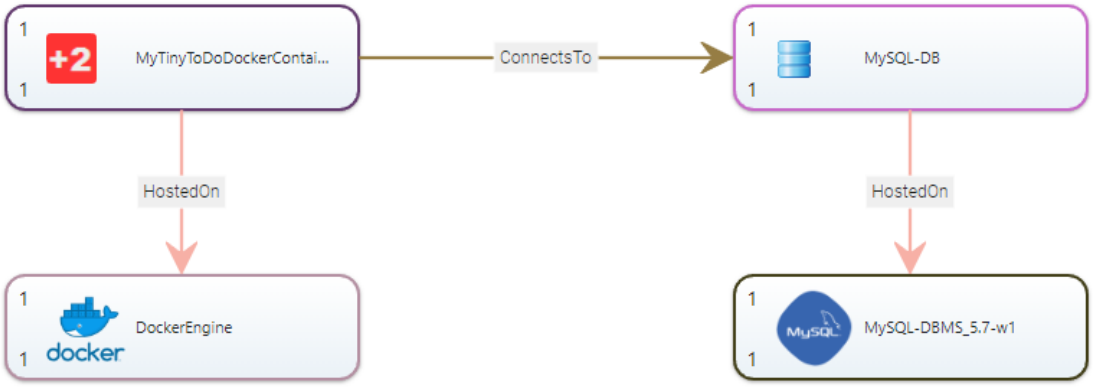
The application topology shown in Figure 1.2 on the following page as well as the topology of a host shown in Figure 1.3 on the next page will act as technical dummys throughout this work. They were created with the *Topologymodeler*[1] of the *Eclipse Winery*[2] project which follows the Vino4TOSCA visual notation for application topologies by Breitenbücher et al. [BBK+12].

The *MyTinyToDo*[3] topology is a minimal example of an application that acts as a placeholder for industrial applications throughout this thesis, especially for those introduced in the use cases in the upcoming section. Its contents are not considered as it solely acts as a dummy for real industrial applications in the context of this thesis. It consists of a containerized variant of the application that connects to a MySQL[4] database which relies on a Database Management System (DBMS) and a Docker[5] engine to run on.

The *UbuntuOnOpenStack* topology acts as a placeholder for different placement environments such as the edge cloud, the private cloud and the public cloud which are simulated for the use cases of this thesis which are described in the following sections. Both the *MyTinyToDo* application as well as the *UbuntuOnOpenStack* instances will have distinct annotations to reflect their differences and to meet

[1] https://github.com/eclipse/winery/tree/master/org.eclipse.winery.frontends
[2] https://projects.eclipse.org/projects/soa.winery
[3] https://www.mytinytodo.net/
[4] https://www.mysql.com/
[5] https://www.docker.com/

Figure 1.2: The topology of the MyTinyToDo application.

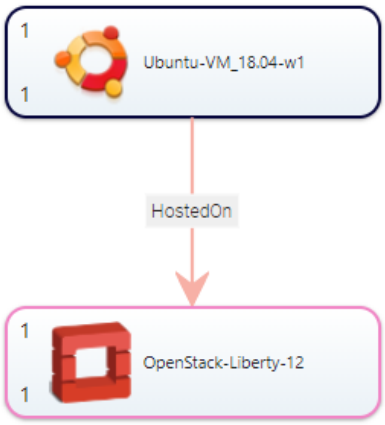


the requirements of the application components explored in the use cases section. The goal will be to place components of the MyTinyToDo application onto instances of UbuntuOnOpenStack. This process is explained in detail in the concept chapter in Section 5.1 on page 49. The proof-of-concept prototype in Chapter 5 on page 49 makes use of these different variants.

1.4 Use Cases for Application Placement in Industrial Environments

To better understand the applicability and the usefulness of the conceptual approach that follows in Chapter 4 on page 39, a number of use cases related to a Driverless Transport System and the computation-offloading of its applications will be presented in this section. These use cases are chosen such that they cover common applications of DTS, are distinct in nature and cover a large range of values for some of the most common properties of components a placement strategy can

Figure 1.3: The topology of the UbuntuOnOpenStack host.



be optimized for, such as cost, bandwidth usage, CPU usage, RAM usage, etc. The application components that are subjects of these use cases will be simulated by the dummy applications that were just presented and used for an initial evaluation of the prototype in Chapter 5 on page 49.

1.4.1 Use Case I: Driverless Transport System (DTS) - Image Processing

A Driverless Transport System, also called Active Shuttle, is used in a production environment to transport items autonomously. To achieve this, the Active Shuttle uses a live camera that is paired with an image processing application which analyzes the footage produced by the camera system. For example, the Active Shuttle can identify objects and find the correct object to pick up out of multiple candidates or it can notice different angles of orientation of an object and position itself in respect to the object it wants to pick up.

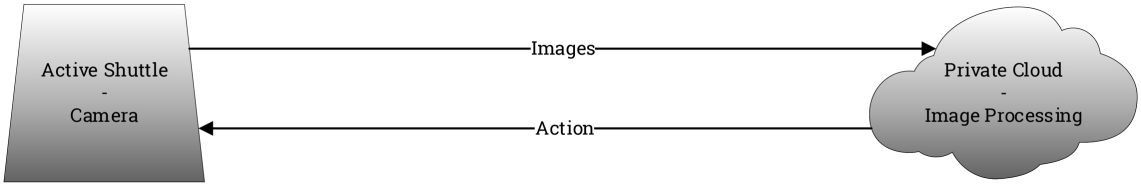
Due to the computationally heavy nature of the image processing application, it shouldn't run on the mobile computer of the Active Shuttle but in the centralized enterprise cloud, since the goal is to minimize the energy consumption of the Active Shuttle so that the productive time-windows of the system can be maximized. A low latency is not of highest priority for this use case but the applications should run reliably to avoid outages of the Active Shuttle. Due to the high bandwidth required, the high sensitivity of the data and the non-feasibility of encrypting and decrypting raw video footage, the public cloud is not an option since it would be too costly, too unsecure and too slow.

A suitable placement strategy for this use case and its image processing component would be a strategy that places this component in the private cloud which is responsive enough as well as secure and reliable.

The Active Shuttle is shown in Figure 1.4 and the schematic view of this use case is shown in Figure 1.5 on the next page.

Figure 1.4: A group of Active Shuttles transporting boxes of goods.

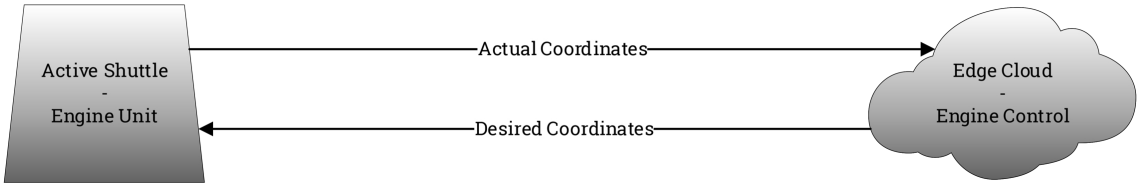
Figure 1.5: Schematic view of the image processing use case.



1.4.2 Use Case II: Driverless Transport System (DTS) - Engine Control

The driving and engine control application of the Active Shuttle requires real-time processing and can't be deployed anywhere where the latency would be too high to perform driving tasks reliably. For safety reasons the DTS stops immediately when the connection times out due to high latency or when it loses the connection to application. This faulty behavior is to be avoided to maximize the productivity of the system. The schematic view of this use case is shown in Figure 1.6.

Figure 1.6: Schematic view of the engine control use case.

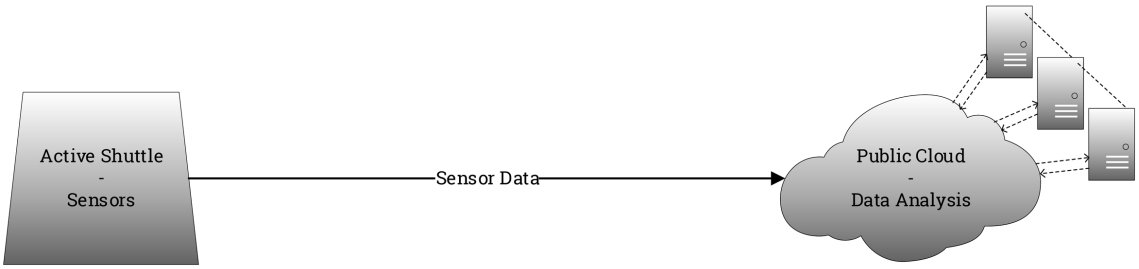


Due to the latency-sensitive requirements of the engine control application, edge computing capabilities are needed to satisfy these requirements. A suitable strategy for this use case would be a strategy which favors low latency and reliability over cost or bandwidth concerns. Therefore the desired hosting location for this component is the edge cloud.

1.4.3 Use Case III: Driverless Transport System (DTS) - Sensor Data

In the last use case sensor data for the provenance of items that are transported by the DTS have to be analyzed. These sensors are equipped with a GPS module as well as a temperature and humidity sensor. The Active Shuttle collects data from these sensors permanently and transfers it to the cloud every now and then. The schematic view of this use case is shown in Figure 1.7 on the next page.

Figure 1.7: Schematic view of the sensor data use case.



Due to the relatively small amount and textual format of the collected data, the bandwidth required for this use case is very low. Latency sensitivity is not given, since the data is transferred in intervals rather than real-time. The data is not security relevant and can be analyzed in a distributed manner. Reliability requirements are low since the data can be transferred during the next interval if the connection is lost.

Therefore, a suitable strategy for the application in this use case would be a strategy that offloads it to a cheap and flexible option, such as the public cloud.

2 Fundamentals

Having presented different use cases for ACP in industrial environments, the next step is to provide some background information as well as knowledge about previous approaches to solving this problem. This work is about providing a conceptual solution for the placement of application components onto existing IT infrastructure in industrial environments as well as to provide a prototypical implementation of it by enhancing the existing OpenTOSCA[1] ecosystem with such a feature. Therefore, it is assumed that the infrastructure consisting of a number running instances of hosts had to be set up using OpenTOSCA in the first place. This is important since knowledge about running instances is critical for being able to place any application. By making this assumption this work can concentrate on the actual step of application placement. Nonetheless an instance model and methods for obtaining knowledge about the states of instances such as hardware crawling and monitoring are a prerequisite for the step of placing components onto running instances and may be considered in future work.

To be able to understand the concept and the prototype of this thesis, the OASIS standard TOSCA as well as the OpenTOSCA ecosystem have to be introduced first. Before that, a quick overview of how management of enterprise applications is done by organizations today is presented in Section 2.1. Section 2.2 on the next page then presents the TOSCA standard and Section 2.3 on page 29 describes the TOSCA runtime that has been developed by the University of Stuttgart, called OpenTOSCA.

2.1 Automation and Management of Enterprise Application Deployment

The landscape of current enterprise applications is often times a web of complex systems. It is no rarity for companies today to have hundreds if not thousands of custom applications deployed. The Cloud Security Alliance in their 2017 trends report even went as far as stating that "every company is a software company" today[2]. This is especially true for many industrial companies where the degree of automation in industrial environments is increased continuously.

These applications are heterogeneous, distributed and modular. On the one hand this enables applications to be cloud-native and benefit from elasticity, horizontal scaling and redundancy [BMQ+07], but on the other hand, since they often depend on each other and have been developed at different points in time with different technologies by different people, the maintenance costs and error proneness of these systems are high. Furthermore such applications are often not documented adequately and management is often done completely manually or via bash scripts [Bin15].

[1] https://www.opentosca.org/

[2] https://downloads.cloudsecurityalliance.org/assets/survey/custom-applications-and-iaas-trends-2017.pdf

The ubiquity of cloud computing offerings in today's industrial IT landscapes has driven the need for manageability of applications that live in multi-tenant environments such as the cloud. Administering and operating such a complex web of applications is costly and difficult but with the right tools it can be automated and managed more efficiently. Descriptive and reproducible approaches such as Infrastructure-as-Code (IasC) have turned out to be a good solution for this problem. Because of the rise of agile and lean techniques that shorten development timeframes and as continuous integration becomes more pervasive with time, the need for IasC tooling grows even more [ABN+17].

Vendors with large cloud offerings like Amazon, Microsoft and Google have written standards for this reason that allow infrastructure design in an IasC manner. One example being Amazon's AWS CloudFormation[3], which allows defining sets of AWS resources and relations between them in a JSON format. Such tools are used increasingly in production as the growing complexity of operating and managing enterprise and industrial applications has forced companies to adopt modern DevOps methodologies and to make use of the benefits of this approach. Apart from proprietary vendor formats and solutions there are also open specifications like the OASIS TOSCA standard that implement similar ideas. TOSCA, an industrial standard for IasC, which is used by many software solutions in this domain, has been developed concurrently to these vendor specific standards and is described in more detail in the following section.

2.2 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) by the Organization for the Advancement of Structured Information Standards (OASIS)[4] is about "enhancing the portability and operational management of [...] cloud applications..." [5] [BBKL13b], [BBLS12]. The TOSCA standard is a YAML (2016) and XML (2013) based descriptive language that can be used to define topologies of cloud applications, the relationships between their components as well as the processes that provision, instantiate and manage them throughout their lifecycles. The important parts of the TOSCA model will be presented in the following subsection, to better understand the concept presented in this thesis.

2.2.1 TOSCA Model

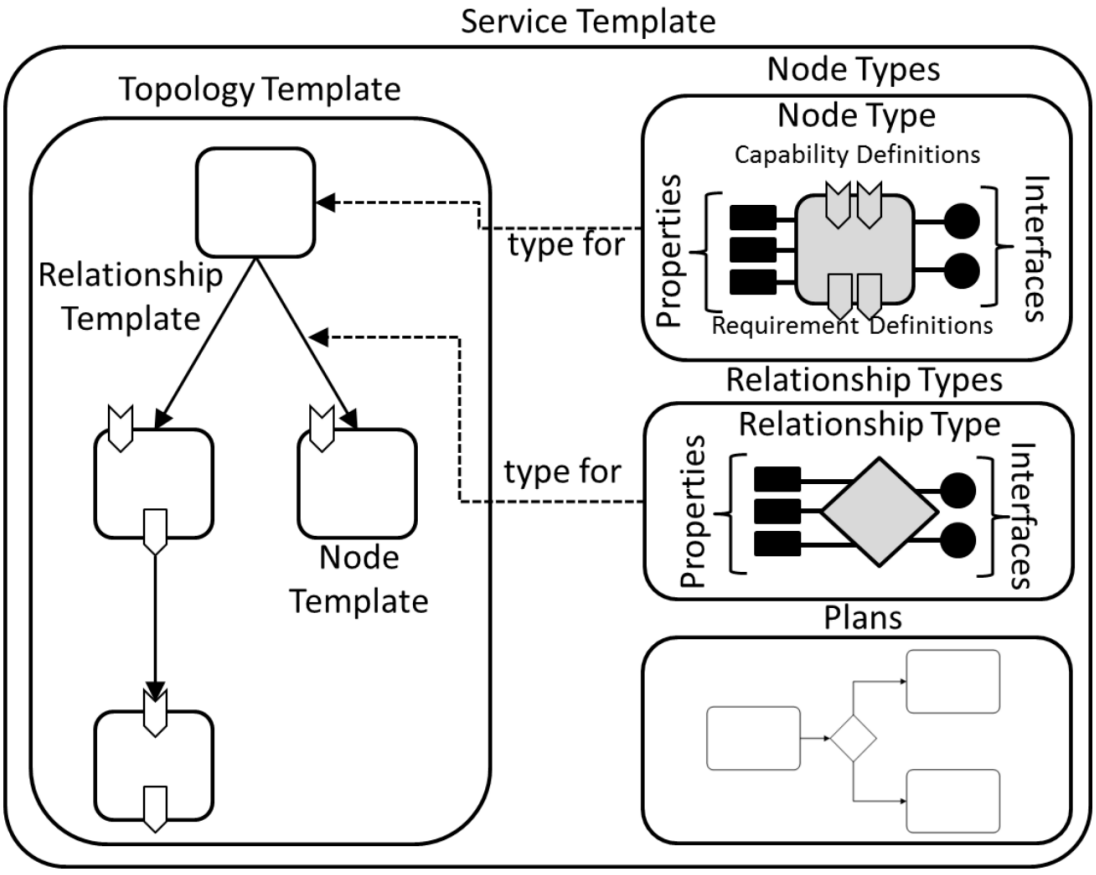
The TOSCA metamodel allows the definition of so-called application topology models in a descriptive manner. The application topology of a web service describes the set of components (nodes) that an application consists of as well as the *Relationships* they have to each other. In TOSCA this definition of a web service's structure is referred to as the *TopologyTemplate* of a service. *Plans* define the models of the processes that are used to manage a service during its lifetime [OAS14]. The *TopologyTemplate*, a directed graph, can be divided into a set of *NodeTemplates* (representing nodes in the graph) and a set of unidirectional connections between them, called *RelationshipTemplates*. These nodes (NodeTemplates) have their respective types that are called *NodeTypes*. Node Types can be thought of classes or types and NodeTemplates can be thought of objects from an object-oriented

[3] https://en.m.wikipedia.org/wiki/AWS_CloudFormation

[4] https://www.oasis-open.org/

[5] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

Figure 2.1: Structural Elements of a ServiceTemplate and their Relations, from [OAS14]



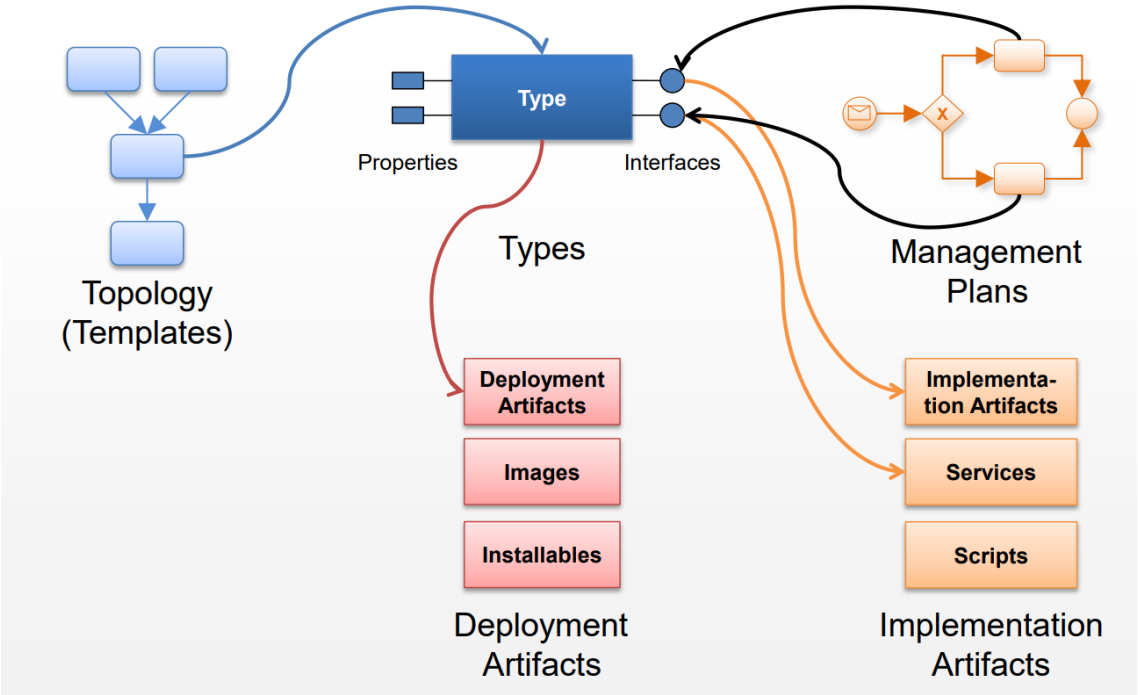
perspective. On top of the classical understanding of classes and objects the TOSCA standard allows NodeTypes to also specify usage constraints. These entities contain a “Properties” field that allows plain text, key-value or XML-based properties to be defined that are used to describe certain aspects of an entity. To model the requirements and capabilities that a component has, the Node Type definition can be extended with *Capability Definitions* and *Requirement Definitions*. These two are specialized properties of Node Types that can be used to model dependencies of certain components like the need for at least 16GB of RAM on a potential host system.

In this thesis, the capitalization of terms such as “Property”, “Requirement” or “Capability” signalizes that it refers to the TOSCA-centric meaning of the word. Analogous to the distinction between NodeTypes and NodeTemplates there are *RelationshipTypes* and *RelationshipTemplates* in TOSCA. They can be used to describe relationships between components such as a ”hostedOn”relation between an application component and an operating system node. The combination of the aforementioned entities represents a *ServiceTemplate*, which is the main construct of a web service in the sense of the TOSCA model. Figure 2.1 shows these structural elements of a ServiceTemplate.

2.2.2 TOSCA CSAR

The **C**loud **S**ervice **AR**chive (CSAR) is described by the TOSCA standard as an archived file containing the different elements that make up a ServiceTemplate. It contains model information needed to be able to interpret the content as well as artifacts needed for the provisioning and management of the service, such as bash scripts, SQL statements or mountable images. All the components mentioned in Section 2.2.1 on page 26 are bundled together with all the necessary artifacts required for the management operations into a single *.csar* file. Figure 2.2 shows this concept.

Figure 2.2: The parts of a CSAR, from [Wai]

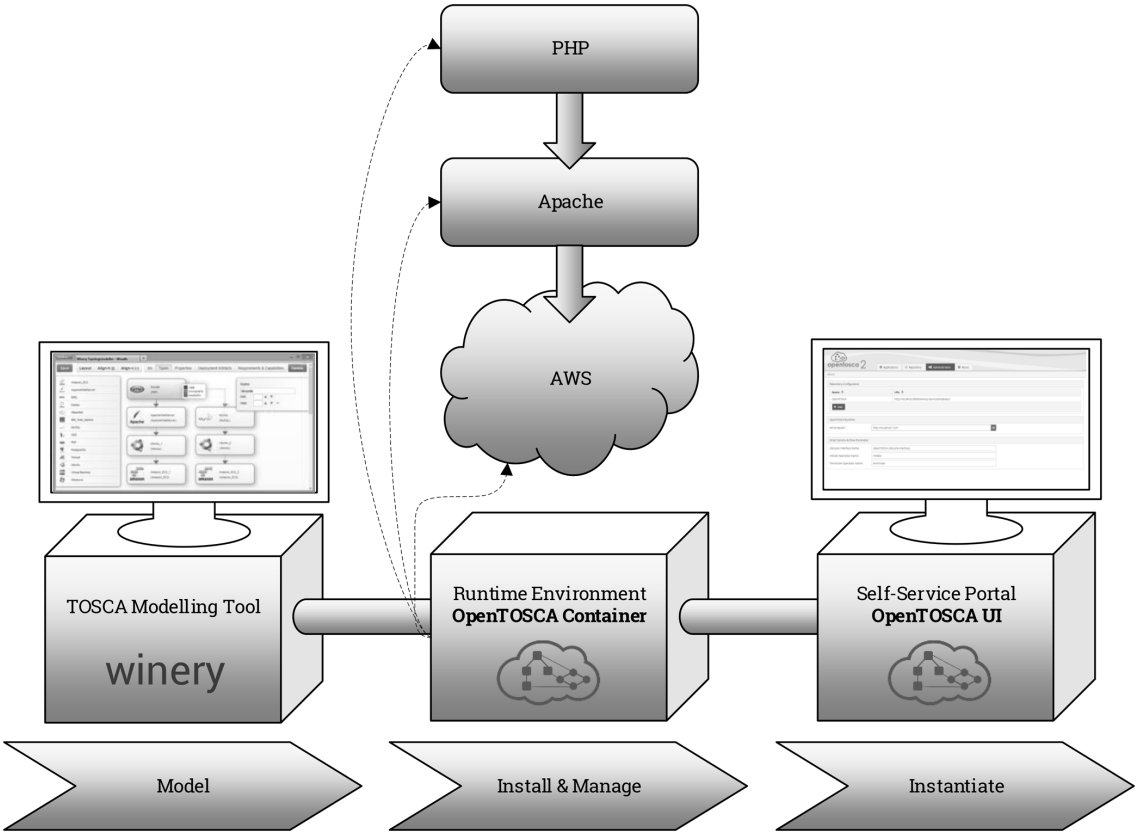


This file is supported by every TOSCA runtime such as OpenTOSCA. It enables users to save their defined services in a portable manner together with meta information about these services that allow the interpretation of the CSAR by the TOSCA runtime. For example, in OpenTOSCA the user can upload CSAR archives of a service and instantiate it through the OpenTOSCA UI.

2.3 OpenTOSCA

The OpenTOSCA[6] ecosystem for TOSCA-based cloud applications was developed by the Institute of Architecture of Application Systems (IAAS) and the Institute for Parallel and Distributed Systems (IPVS) of the University of Stuttgart. It's partially founded by the German government and various affiliated projects. It supports the majority of the TOSCA specification and consists of multiple components shown in Figure 2.3.

Figure 2.3: The three components of the OpenTOSCA ecosystem, from [RRR+16]



2.3.1 Winery

The Eclipse Winery project is a web-based graphical tool for modeling TOSCA applications, their topologies as well as for defining reusable TOSCA types like Node Types and Relationship Types or adding different artifacts to ServiceTemplates. An instance of Winery requires a so-called winery-repository that has a REST-interface which allows the changes from the graphical tool to be reflected in the file system. It uses the TOSCA standard as storage format and supports the CSAR file type for importing and exporting ServiceTemplates [KBBL13]. It was initially developed by the

[6] https://www.opentosca.org/

University of Stuttgart and later introduced to the Eclipse Foundation[7]. There is a fork of Eclipse Winery called OpenTOSCA Winery that is used for preparing OpenTOSCA contributions to the Eclipse version and has a different set of features.

2.3.2 OpenTOSCA Container

The OpenTOSCA container is the actual runtime for TOSCA-based cloud applications [BBH+13]. It is written in Java and based on the OSGi[8] framework to establish a modularized component model architecture of bundles that guarantees extensibility, as well as Apache Maven[9] for management of the software project. It supports the provisioning of the services that can be modeled in the Winery by interpreting the contents of the respective CSAR file. It has a REST-interface that allows various operations on the contents of a CSAR as well as on the running instances of the container itself. Figure 2.4 on page 32 shows an overview of the API and the various resources that can be addressed and accessed through it. The underlying services are used in this thesis to gather information about running instances that inform the placement algorithm.

It consists of multiple components such as the *Application Bus*, the *Plan Builder* and the *Service Invoker*. The Application Bus provides a number of APIs that provide the core functionality of the Application Bus Engine to other components. The Application Bus Engine collects the information that is needed for the invocation of operations, for example, determining the endpoint of the invoked application, and determines the plugin that is capable of performing the invocation [Ope]. The Plan Builder is generating the Build Plans and Termination Plans that are able to install, deploy and provision the parts of a TopologyTemplates. It derives an executable BPEL process from the abstract control flow that it constructs from the topology of an application in a given CSAR and the type of plan to be created [Ope]. The Service Invoker (aka Management Bus) provides the service invocation interface and connects to other components of the OpenTOSCA container. It is a mediator between a plan and the SI-Engine that invokes the actual services, such as the Endpoint Service and the Instance Data Service [Ope].

Figure 2.5 on page 33 shows an architectural overview of the OpenTOSCA container.

2.3.3 OpenTOSCA UI

The OpenTOSCA UI is a modern Angular-based[10] successor to the OpenTOSCA Vinothek project. It is a self-service portal that allows the provisioning of cloud applications via a graphical web-application that hides technical details of the container [BBKL14]. The user can see running instances in detailed views of certain services and initiate management actions that were defined as management plans during the design phase.

Figure 2.6 on page 33 shows the interaction between the OpenTOSCA UI and the OpenTOSCA container. A user would usually upload a CSAR that defines a ServiceTemplate that can be interpreted by the TOSCA runtime. This prepares the ServiceTemplate for provisioning immediately and results

[7] https://projects.eclipse.org/projects/soa.winery
[8] https://www.osgi.org/
[9] https://maven.apache.org/
[10] https://angular.io/

in an error if the CSAR that was uploaded is not well-defined. In the next step the user would provide certain properties that are required for the instantiation of a certain component, for example, the credentials to a cloud provider. After the user confirms the provisioning the OpenTOSCA UI sends a HTTP request to the REST interface of the OpenTOSCA container with payload that contains the properties that were specified. When the instantiation was initialized by the container, the UI gets refreshed with the according information about the instance and an entry in the running instances list.

Figure 2.4: An overview of the container API, from [Ope].

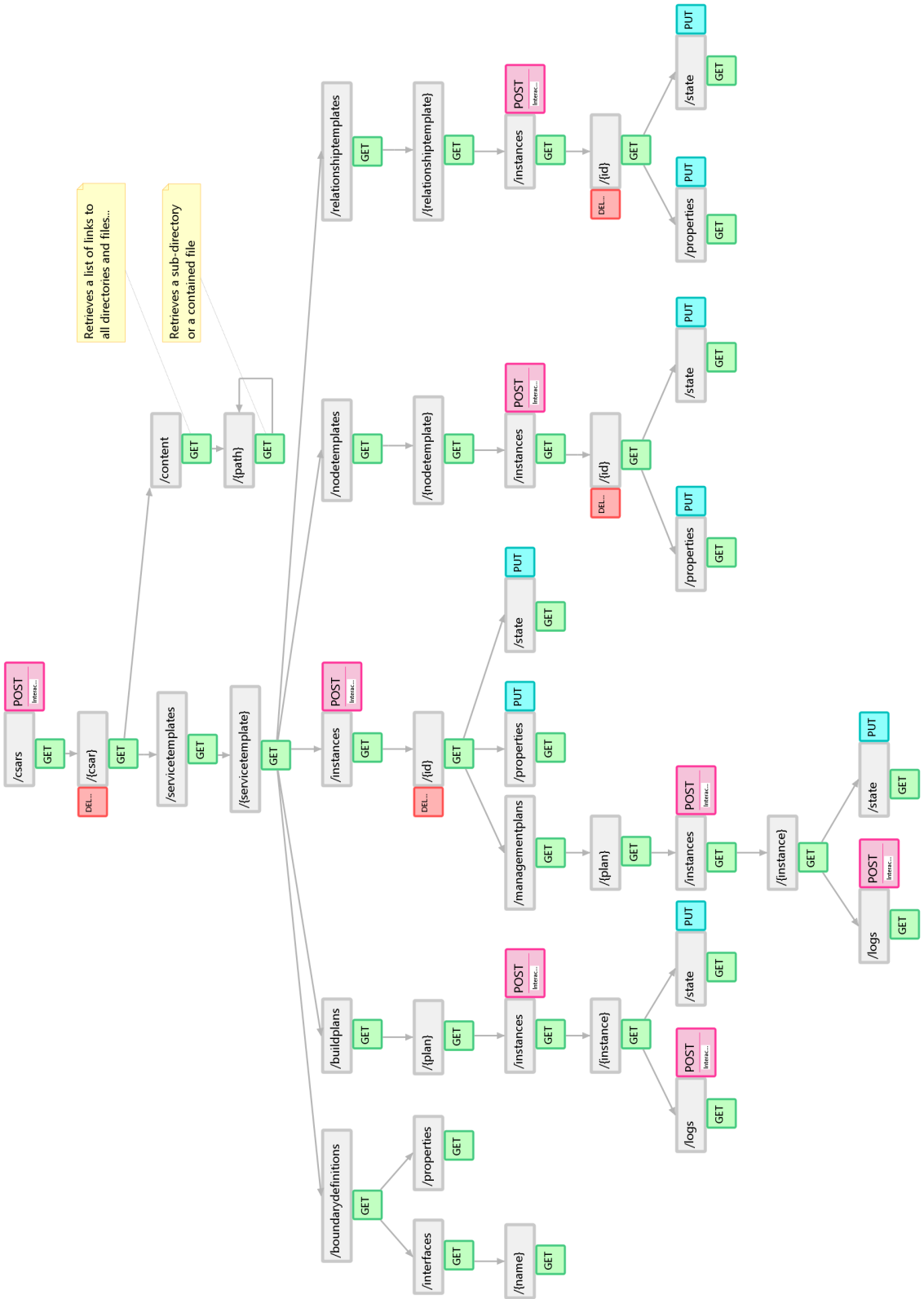


Figure 2.5: An overview of the container architecture, from [Ope].

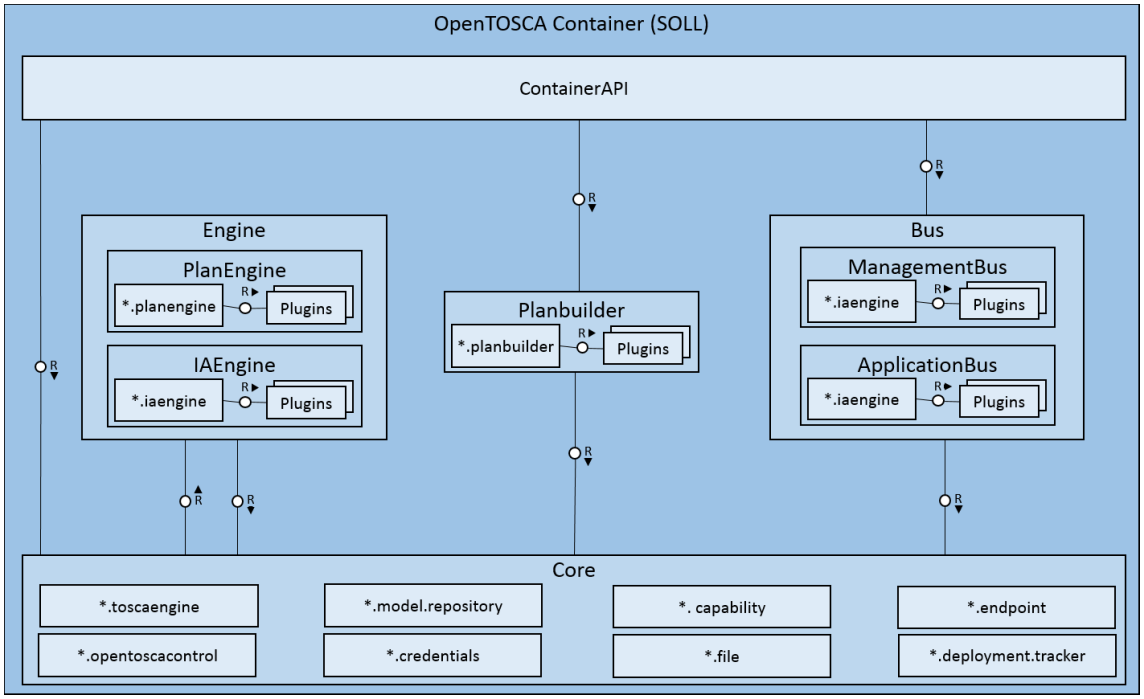
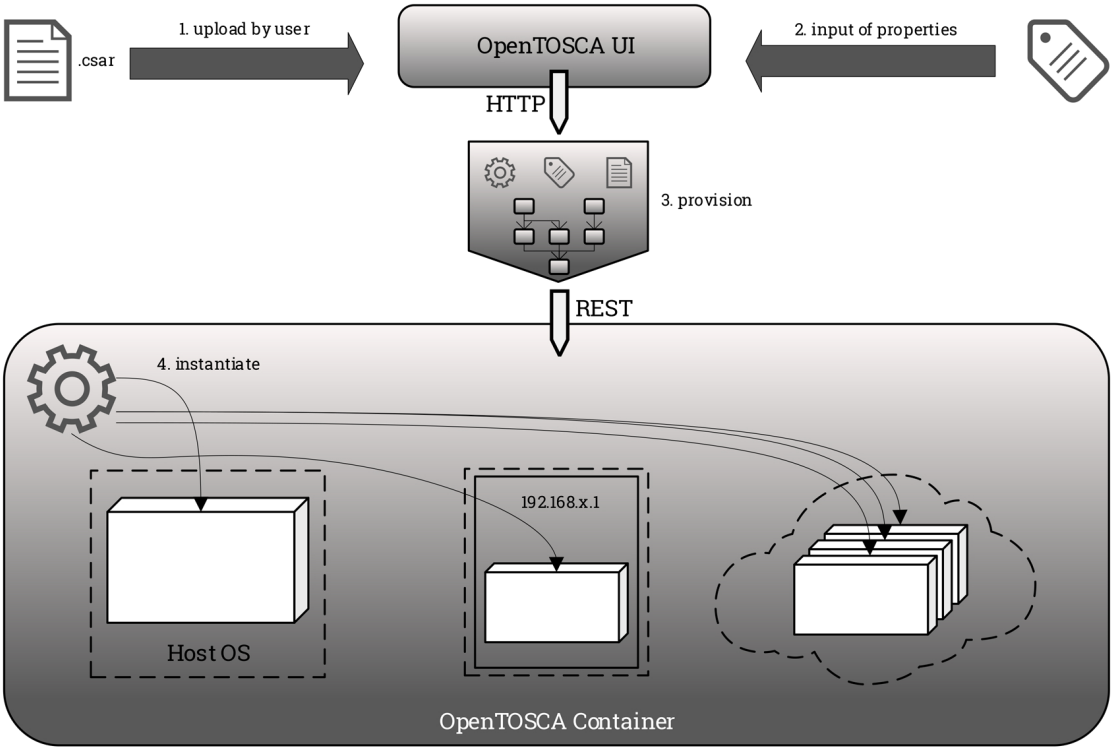


Figure 2.6: The workflow of instantiating an application in the OpenTOSCA ecosystem.



3 Related Work

There is already a large amount of work on the topics discussed in this thesis. This chapter gives a quick overview of related work the concept of this thesis is based on and aims to deliver a glimpse into the thought process that went into the narrowing down and development of it. Firstly, methodologies for obtaining information about running instances are presented, followed by a brief summary of previous approaches to solving the ACP problem.

3.1 Obtaining Information about Running Instances and their States

Large and complex nets of enterprise applications require a model for keeping track of the states of their running instances, as well as relationships between them, to be able to adjust the infrastructure to the quickly changing demands that organizations today have to adapt to [Bin15]. Such an instance model is required to avoid the tedious and costly task of identifying existing instances in the IT-infrastructure manually and to be able to analyze, adapt and optimize the IT-infrastructure [PX13]. The two main approaches for sourcing the information for such an instance model that were identified during the research phase of this thesis are crawling and monitoring. The next two subsections will present earlier work on these approaches, both done in the context of TOSCA and OpenTOSCA respectively.

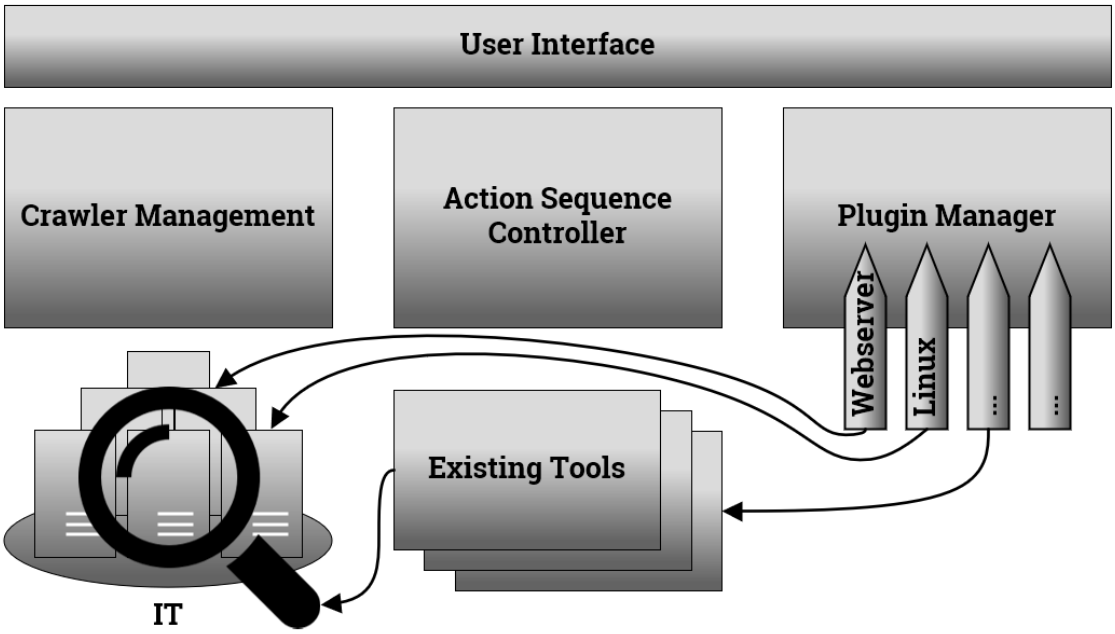
3.1.1 Crawling of IT-Instance Models

Binz presents an automated way of crawling IT-instance models using TOSCA [BBKL13a], [Bin15]. His idea is to use a crawler that generates component topologies and can be extended by plugins for idiosyncratic aspects of different components. Such a plugin can either be written in a way to obtain information about the logic and relationships between components or about the actual state of components. The focus on extensibility is explained by the need for being able to address the heterogeneity of IT-systems. The plugins that make up the crawler have to *pull* i.e., extract the data from the components rather than the components *pushing* the data to the crawler, as the components shouldn't have knowledge about the crawler and as it would require a manipulation of the existing setup.

His concept of a crawler follows a top-down approach, meaning that it starts from the root nodes of an application and goes deeper towards the platforms it is hosted on and the machine it is deployed on. Like this a topology that follows the TOSCA specification can be generated by the crawler. It consists of multiple components with different tasks. The first component is an action sequence controller that decides the order and locations in which the different plugins are executed. The second component is a management component that stores data about iterations of the crawler, and

the last component is a plugin management component that manages the injection of plugins and is extended by a deduplication component and the component that generates the topologies. Figure 3.1 shows an overview of this architecture.

Figure 3.1: Architecture of a crawling concept, from [Bin15].



The plugin nature of this concept allows the retrieval of any kind of information about running instances that may inform the placement algorithm during the placement operation.

3.1.2 Monitoring

Another way of obtaining information about the state of running instances is monitoring. Monitoring represents a whole field on its own, and is only touched upon briefly in this subsection. In an earlier master's thesis different monitoring systems were compared and as a result of this evaluation the TICK Stack[1] was integrated into the existing OpenTOSCA ecosystem [Wie18] as part of a prototype.

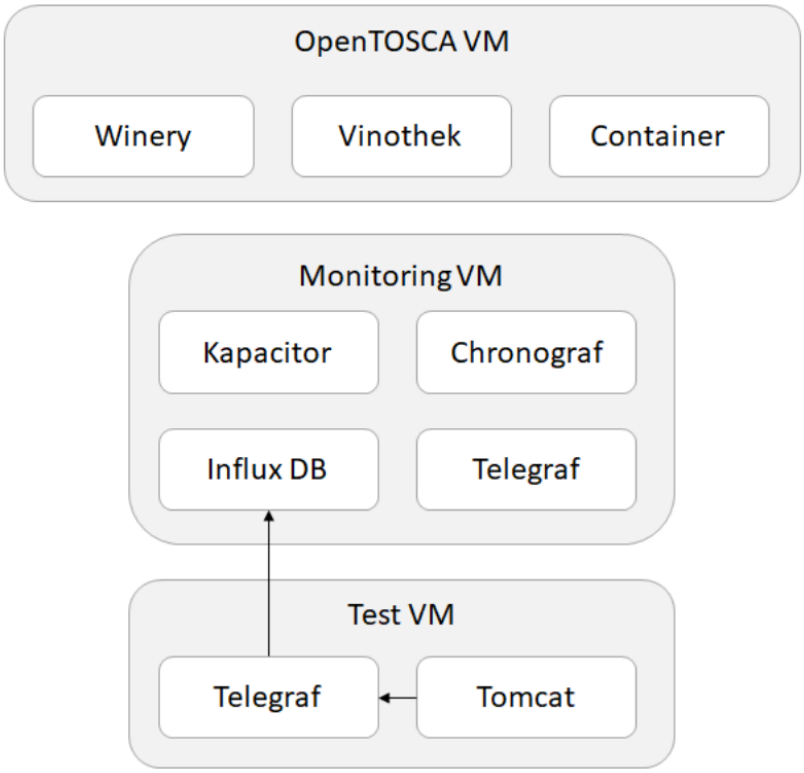
The TICK Stack is based on Telegraf[2], InfluxDB[3], Chronograf[4] and Kapacitor[5].

[1] https://www.influxdata.com/time-series-platform/
[2] https://influxdata.com/time-series-platform/telegraf/
[3] https://www.influxdata.com/products/influxdb-overview/influxdb-2-0/
[4] https://influxdata.com/time-series-platform/chronograf/
[5] https://influxdata.com/time-series-platform/kapacitor/

Telegraf is an agent that gets deployed on an instance and can collect different metrics such as system stats (CPU, RAM, networking) as well as listen to events and observe message queues. Kapacitor is the real-time stream processing engine that orchestrates incoming data of multiple Telegraf instances and Chronograf is the interface to the InfluxDB, a time series database capable of storing this data.

The way the monitoring system was integrated was that a separate monitoring VM hosting the TICK Stack was set up alongside the OpenTOSCA VM that runs the OpenTOSCA container, the OpenTOSCA UI and the Eclipse Winery. A test VM hosting an Apache Tomcat[6] instance that serves as a dummy application was then supplemented by a Telegraf agent that reported information about the test VM to the monitoring VM. An overview of this setup is shown in Figure 3.2.

Figure 3.2: Overview of the monitoring system architecture from [Wie18].



Because crawling and monitoring present whole fields of research on their own and the prototypes of these works aren't trivial to setup, using real instance data sourced by these systems was outside the scope of this thesis, therefore instance information is only simulated for the prototype of this thesis. The details of the simulated example that the prototype was initially tested against are presented in Chapter 5 on page 49.

[6] http://tomcat.apache.org/

Nonetheless, the presented related work on crawling and monitoring demonstrated that obtaining information about running instances is possible with TOSCA and the OpenTOSCA ecosystem. Therefore it is safe to assume that a crawling and monitoring system can exist in the OpenTOSCA ecosystem and can provide information about running instances which could potentially feed the decision process of the component placement algorithm that is presented in this thesis.

The remaining part of the related work chapter is concerned with the ACP problem.

3.2 Approaches to the Application Component Placement Problem

There have been many approaches of trying to solve the problem of ACP – known to be NP-hard [URS07] – efficiently, ranging from mathematical optimization frameworks like the ACP solver by Xiaoyun Zhu et al. [ZSB+08], [JCL11], over centralized [KSST05], [TSSP07], [CSW+08], [ASLW14], [BML+17] and distributed [BMT14] algorithmic approaches. This section presents the approach to solving this problem that inspired the ideas in this work, before presenting the conceptual approach discussed in this thesis in Chapter 4 on the next page.

3.2.1 Dynamic Application Placement under Constraints

Kimbrel et al., a group of IBM researchers, presented a heuristic algorithm to solve the problem of satisfying the requirements of applications in an environment where available system resources can change dynamically [KSST05]. Their algorithm was implemented into the IBM Websphere[7] environment as part of the Websphere component known as the *placement controller*. The way it works is that the placement controller is aware of the preexisting mapping of applications onto instances in a given Websphere cell. It then can periodically or upon need execute the placement algorithm to readjust the mapping i.e., the placement of components onto running instances. The placement controller reacts to changes in loads by relying on monitoring data. It also contains a workload predictor and historical information to make better decisions. Their algorithm assumes the following problem formulation:

- There are m servers (instances) $1, ..., m$,
- with memory capacities $\Gamma_1, ..., \Gamma_m$,
- and service capacities (no. of requests served per unit of time) $\Omega_1, ..., \Omega_m$.
- There are n application components with memory requirements $\gamma_1, ..., \gamma_n$.

Before the placement all servers (instances) the values of their densities Ω_i/Γ_i and the densities of application components ω_{jt}/γ_j are ordered in decreasing order. Then the two with highest densities get matched if the capabilities of the server satisfy the memory demands of the application and the application gets removed from future consideration. The available memory and service capacities are recomputed after each iteration. This is repeated until the initial placement candidate is found.

The concept of this thesis is based on a similar idea and is presented in detail in the upcoming chapter.

[7] https://www.ibm.com/cloud/websphere-application-platform

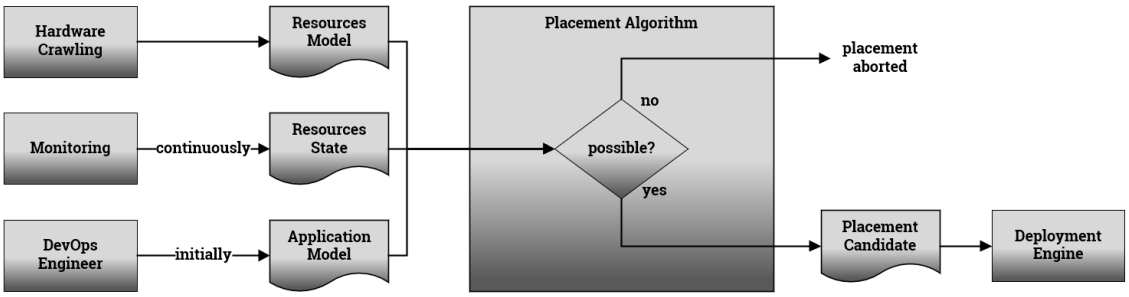
4 A Concept for the Placement of Application Components

Having introduced the ACP problem briefly, as well as different ways it was tackled by earlier work, this chapter starts with an overview of the general idea and design behind this concept. After that, the underlying ACP problem is formalized and the algorithmic approach that is used in this work is presented, along with different potential strategies of flavors the algorithm could be adapted to. Finally, a bridge from the theoretical concept to the OpenTOSCA ecosystem is made, to introduce the prototypical implementation of the concept discussed in Chapter 5 on page 49.

4.1 General System Design

The system design of the concept is shown in Figure 4.1. It is inspired by earlier work on automated ACP by Zhu et al. [ZSB+08] who used a similar design for their ACP solver.

Figure 4.1: A simplified view of the general system design of the conceptual solution.



A crawling and monitoring system is assumed which in an ideal case would feed into the resource or instance model of the given environment. Ideally the instance model would already exist before any placement operation is done or it would be generated by the crawler in a preliminary step. The monitoring system should continuously update the state of the resources so that there is a single source of truth that represents the amount of available resources or the loads of a certain instance at a given time. It is also assumed that the application model was initially designed by a human who expressed preferences over e.g., the placement locations of individual components. In the case of this thesis the resource model is only simulated but in a real use case knowledge about the given IT-infrastructure should preferably be known before the application is modeled. These three sources are then fed to the placement algorithm which performs the actual placement operation and returns a placement candidate if one is found or aborts the placement if no candidate could be found. This candidate is represented as a map between NodeTemplates of the components of applications that

have to be placed and the NodeTemplateInstances of their chosen hosts. To avoid a coercive user experience the placement candidate is then presented to the user who can override the decisions of the algorithm. After that, the final TopologyTemplate that contains all the selected matches between components and instances is constructed by making use of the topology editing capabilities of the Eclipse Winery. This returns a well-defined CSAR that can then be deployed normally using the existing workflow of the OpenTOSCA ecosystem.

4.2 Formalizing the problem

Chapter 3 on page 35 showed that ACP solutions from related work or the industry itself are in general based on some kind of theoretical model, such as a collection of sets or graph-based models. Therefore this section describes the formal model that was used to better reason about the underlying problem of this thesis and that enabled the engineering of an appropriate algorithm which addresses the problem. Many publications in the field of ACP use their custom and/or proprietary models to describe their solutions and concepts. The previously described TOSCA standard makes no exception to this observation, as demonstrated earlier in the fundamentals chapter. Even though all of them have their respective idiosyncrasies that get more perceivable the more one gets into the details, these models still follow very similar ideas that can be summarized as follows:

There is a set of components $C = \{c_1, c_2, ..., c_i\}$ with its corresponding subsets of software components $S = \{s_1, s_2, ..., s_j\}$ and hardware components $H = \{h_1, h_2, ..., h_k\}$, where $S \subset C \supset H$ and $S \cup H = C$. Between components are connections $R_{\text{mn}} = (r_m, r_n)$ that are often described as relationships. These connections can exist between software components where they describe software dependencies (e.g., a webserver relying on a database) as well as between hardware components (e.g., a physical connection) or between software and hardware components (e.g., an operating system being hosted on a hardware node).

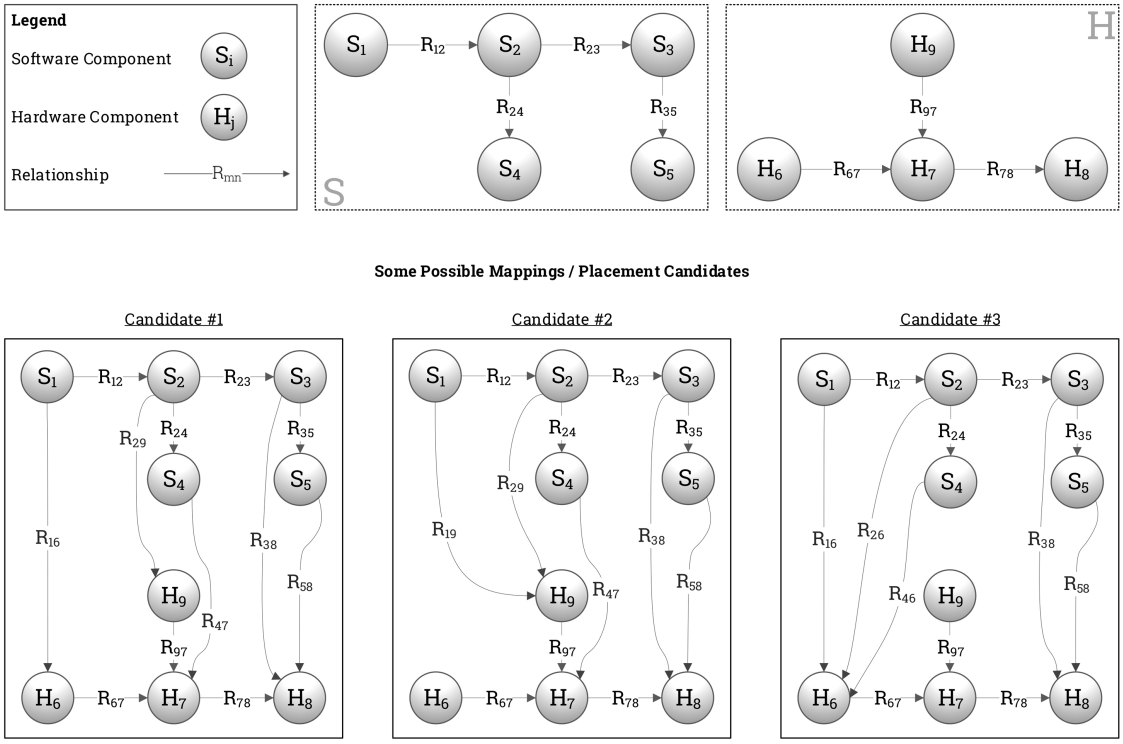
Components and connections have certain properties attached to them that carry necessary information for the mapping between software and hardware components or in other words for the placement of software components onto hardware components.

A common abstraction of software and hardware components is required in theory because the boundary between software and hardware is not a hard one but can rather be moved on the continuum that exists between bare-metal, hypervisors, virtual machines, operating systems, PaaS offerings, and so on. This is especially true today where the boundary between hardware and software gets blurred increasingly. Hence, it's up to the user to decide what qualifies as hardware and where software starts in a certain use case. In practice this distinction can be achieved by attributing different properties to hardware and software components and by assigning different types to them, so that they can be distinguished from each other clearly. One way this can be done, and the way that it is done in the prototype of this thesis, is to only regard VMs that qualify as operating system nodes as potential hosts for application components and to disregard their successors i.e. the machines that they are hosted on, as they aren't required for the placement process. Therefore, whenever the word hardware is used in this work, it can stand for any kind of node with hosting capabilities, such as a VM.

As previously stated, the goal is to place **all software components** onto hardware components. This implies that not all hardware components have to be used, especially if the goal is cost efficiency. Rather than occupying fresh host instances, one may want to reuse hosts that still have enough computational resources to host another application.

Figure 4.2 shows a graphical representation of a set of software and hardware components as well as different candidates that could result from matching the two. For example, in *candidate #1* all hardware components are utilized, whereas in *candidate #2* the hardware component H_6 is not used as a destination by any of the software components. Achieving this reasonably well relies on solving an optimization problem.

Figure 4.2: Possible Candidates



In principle all quantifiable and qualitative properties of software components (requirements) have to be met by those of hardware components (capabilities) for a match to become possible. As long as the model allows assigning properties to components that can later be compared and matched algorithmically, one can find appropriate placement candidates in reasonable time for realistic problem sizes with the correct approach.

These properties are not limited to the ones that are predetermined by the nature of an application (like the requirements of a JRE of a Java application) but can rather be chosen freely by the system architect according to their needs or for example, represent system stats that were gathered through crawling and monitoring.

In fact, any metric that carries useful information for its user can be used to reach conclusions about the placement problem. For reasons of scope, this work will only concentrate on a couple of these properties, but similar techniques to those presented in this work can be used for any

quantifiable metric. Since a component often will have many different properties or metrics that describe its requirements or capabilities, these metrics can be regarded as values on distinct axes which corresponds to a multi-dimensional vector. The multi-dimensionality of these attributes that have to be optimized for, provides a complex problem that has to be tackled in the right manner. On a side note, since optimizing for multiple dimensions at once is very difficult, dimensionality-reducing cost functions and greedy approaches that make use of one dimensional approximate costs, seem to be reasonable approaches for future work.

Having made this assumption, it should be said that the goal of this work is not to provide ground-breaking algorithmic methods that can improve the time-complexity compared to existing solutions. The goal is to define a conceptual approach that works reasonably well, provides a higher degree of automation for its users and allows the expression of user preferences for the treatment of individual components by the placement algorithm.

It should provide some additional value for the user compared to traditional methods of placing application components, especially in those cases where the problem sizes aren't too large and placement decisions are still often done manually by humans based on historical data and experience of the system architects.

4.3 Algorithmic Approach

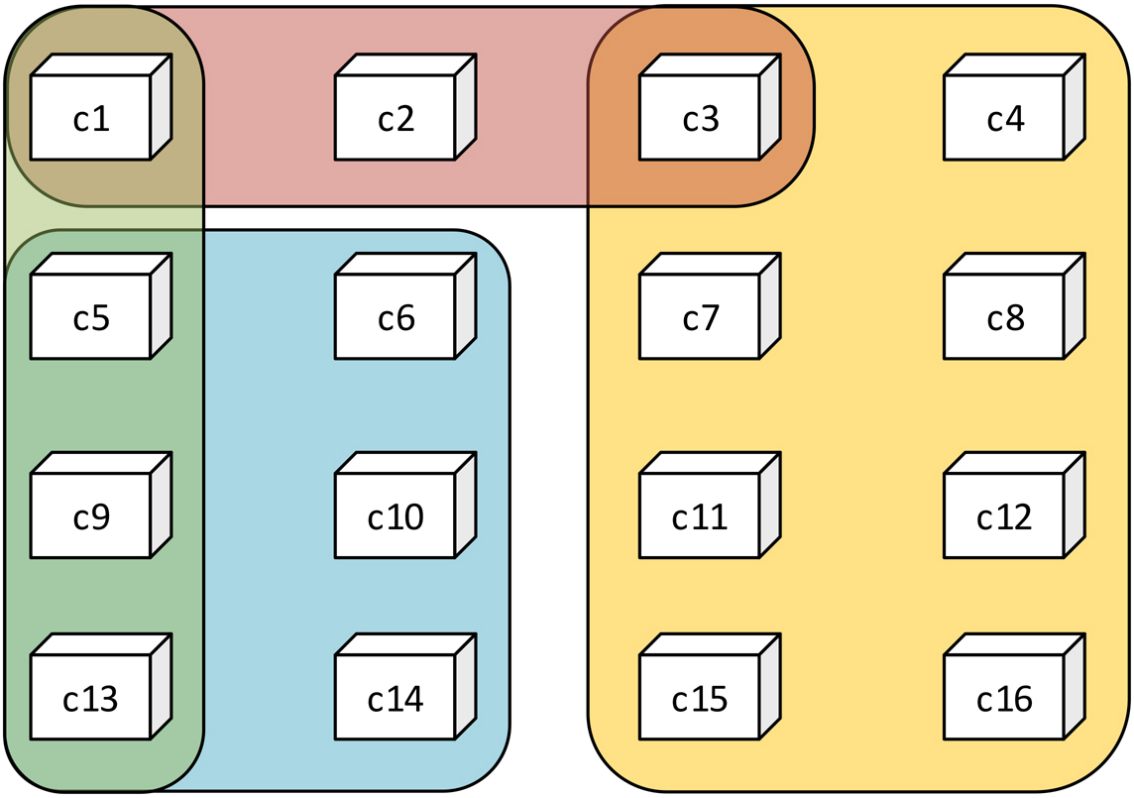
Optimizing the placement of subsets of software components onto subsets of hardware components for multiple attributes like e.g., CPU load, RAM space and network load, at the same time, is a NP-hard problem and can't be achieved in polynomial time. Another assumption in this work in that this class of problems represents a variation of the well-known Set Cover Problem (SCP) and can be reduced to it. It's one of Karp's 21 NP-complete problems, demonstrated to be the case in 1972 [Kar72]. The following subsection will demonstrate the suitability of the Set Cover Problem for the goal of this work. Even though the practical part of this thesis i.e., the prototypical implementation that is described in Chapter 5 on page 49, does not mirror the theoretical concepts discussed in this section in a 1:1 fashion, it was still heavily influenced by the formal discussion of the SCP that follows.

4.3.1 The Set Cover Problem

For easier reasoning about we can re-use the sets that were defined in Section 4.2 on page 40. Instead of dividing C into S and H though, we can ignore this distinction here, as the Set Cover Problem is a generalization of a whole family of such optimization problems and is easier to understand this way.

Definition 4.3.1
Let $C = \{c_1, c_2, ..., c_n\}$ be the universe of n components, $K = \{K_1, K_2, ..., K_j\}$ be the collection of subsets of C, and $w : K \rightarrow \mathbb{Q}+$ be a cost function. Find a minimum cost collection of subsets of K that covers all elements of C.

Figure 4.3: A possible solution of a Set Cover Problem.



In the case of mapping software components onto hardware components this means that all software components have to be covered by a hardware component such that the cost (however this cost function is specified) is minimal. To be able to visualize how the problem of application placement can be reduced to the Set Cover Problem, we have to make a few distinctions first. In our use case the idea of a subset of components K_j represents a hardware component that covers this set of software components. This means that we don't need to take the hardware components into consideration as elements of C when doing this, as the resulting set of subsets $\{K_1, ..., K_m\}$ contains precisely the representations of the m hardware components we want to deploy onto. Furthermore the Set Cover Problem allows for the overlapping of such subsets. Semantically, for the use case of this work, this would mean that multiple hardware components can host a certain software component. Obviously this is just a limitation of the physical reality of the problem but does not violate the validity of this observation in any significant way. Figure 4.3 shows a visual abstraction of a solution to a Set Cover Problem.

In the case of Figure 4.3, the colored rounded rectangles are the subsets of C (the hardware components) that host the software components $C = \{c1, ..., c16\}$. As you can see, the green subset is not required to cover all software components as the red and the blue subsets also provide the capabilities required by the components $\{c1\}$ and $\{c5, c9, c13\}$ respectively. We would therefore have to choose the subsets $\{red, yellow, blue\}$ to cover all software components.

Theoretically, in a more complex example with many more components there could be billions of possible subsets that are redundant in terms of fulfilling the requirements of the software components through their capabilities. Choosing the optimal set of subsets (e.g., the ones with the minimal cost) would therefore be impossible to do in reasonable time in these cases – as mentioned earlier – especially with brute force approaches. To address this impossibility there are some greedy strategies that make the Set Cover Problem more approachable through algorithmic approaches.

4.3.2 A Greedy Approach to the Set Cover Problem

A common approach for solving problems like these are variations of so called greedy approximation algorithms [Bla98]. Greedy algorithms are used in optimization problems and follow some sense of intuition or heuristic. They make the compromise of selecting the locally optimal choice at each iteration and thereby approximating the global optimum. In many problems like the Set Cover Problem such approaches don't always produce optimal solutions as they don't consider all the options but they are are part of a family log(n) approximate algorithms that are used to solve such problems [Hoc82]. The idea is to make iterations of optimal steps at each step of the iterator to find a solution eventually that, approximates the globally optimal solution reasonably well.

In the case of this work, a greedy approach would remove the already covered software components after each iteration of the algorithm from the universe of components C which would remove them from consideration during the following iterations of the algorithm. In other words, if we take a look at Figure 4.3 on page 43 again, if we imagine that the green subset is the result of the very first iteration of the algorithm, it would certainly be part of the final solution, even though we can easily see that it shouldn't be part of an optimal one, since selecting the red, blue and yellow ones leaves us with only a single redundancy ($c3$). Nonetheless, this seems like the most reasonable approach for the goal of this thesis, as the decision making of application placement is still done mostly manually in current industrial environments and this would be a step forward into the automation of such processes. The critical part of getting the best heuristic for such a greedy algorithm is therefore to define a strategy that determines a smart order in which the software components are selected for placement.

Such a strategy could be as simple as one that chooses the destination for software components solely on the least potential amount of computational residue during each of the algorithm's iterations.

4.3.3 Least Residue Strategy

Each of these software components may have different requirements e.g., different amounts of free RAM required for operation. The cost function $Cost(S_i)$ in this case could be defined as the sum of residual GB of RAM for each subset S_j. The following example is used to demonstrate this strategy. Assuming that the algorithm follows the least computational residue principle at each iteration, it would always choose the subset of components whose cost effectiveness is smallest. The elements of the chosen subset would be added to a resulting set I, until I equals the set of all components U and no more components are left.

Example: Let $\{s1, s2, s3, s4$ and $s5\}$ be software components that have to be placed on a set of hardware components.

The set of subsets corresponds to the hardware components with the necessary capabilities that satisfy the requirements of the software components. For simplification purposes lets assume different constant costs for the subsets at this point to focus on the algorithm itself. In a real use case the cost would have to be recalculated at every step.

$S_1 = \{s1, s2\}, Cost(S_1) = 3$

$S_2 = \{s1, s3, s4\}, Cost(S_2) = 2$

$S_3 = \{s2, s3, s5\}, Cost(S_3) = 4$

In this case, hardware component 1 which corresponds to subset S_1 has the theoretical capabilities that could satisfy the requirements of the components $\{s1, s2\}$. The other two subsets follow analogously. The algorithm would go through all subsets and find the subset S_i whose ratio of $Cost(S_i)/|S_i - I|$ is minimal. This tries to place as many components as possible while minimizing cost (in this case computational residue) at each step. The components in S_i would then be added to I. The pseudo code of this greedy algorithm is presented in Algorithm 4.1.

Algorithm 4.1 Pseudo code of the greedy algorithm, inspired by [Ste06]

```
I ← ∅
while I ≠ U do
    find Si in S1, S2, ..., Sk with Min(C(Si)/|Si − I|)
    I ← I + Si
end while
return I
```

A Walkthrough

First iteration:

$I = \{\}$

- $S_1 = Cost(S_1)/|S_1 - I| = 3/2$
- $S_2 = Cost(S_2)/|S_2 - I| = 2/3$
- $S_3 = Cost(S_3)/|S_3 - I| = 4/3$

Since S_2 is the minimum in this iteration we add its elements to I.

Second iteration:

$I = \{s1, s3, s4\}$

- $S_1 = Cost(S_1)/|S_1 - I| = 3/1$
- $S_3 = Cost(S_3)/|S_3 - I| = 4/2$

Since S_1 is the minimum in this iteration we add its elements to I.

Third Iteration:

$I = U = \{s1, s2, s3, s4, s5\}$

Since I now contains all components we are done at this point. The subsets that the algorithm selected are hardware components containing the software components that shall be placed onto them.

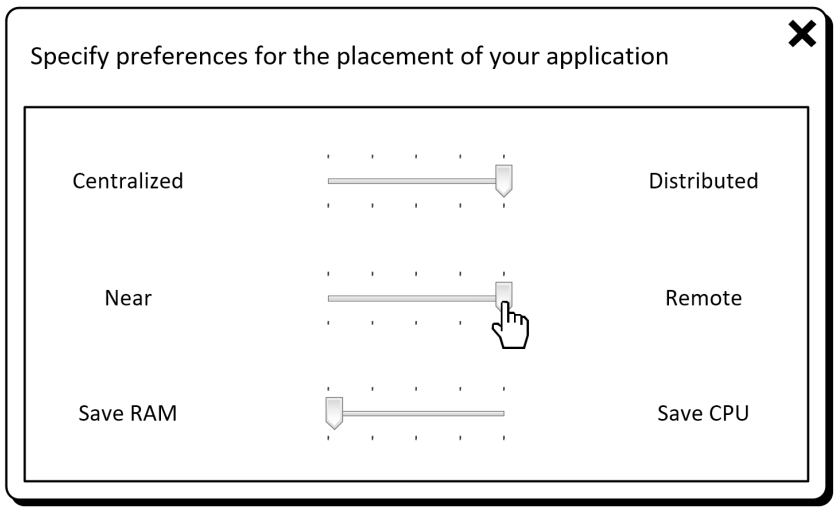
4.3.4 Other Strategies and Heuristics

As part of this work, a prototypical implementation of such an algorithm was implemented in the OpenTOSCA container and will be described in detail in the Implementation chapter. This prototype is limited to the Least Residue Strategy described in Section 4.3.3 on page 44 since the implementation of a generic version of the *Log(n) Approximate Greedy Algorithm* that would support interchangeable strategies and heuristics is beyond the scope of this thesis.

In an ideal case though, the user of this system would be able to specify preferences in terms of providing different heuristics or strategies, and different flavors of cost functions, to make this algorithm adaptable to different user preferences for different use cases in production.

A conceptual mockup of such a potential feature is shown in Figure 4.4. It could be added to the placement dialog of the UI, which is described in more detail in the Implementation chapter.

Figure 4.4: Preferences for the algorithm a user could express through the UI



Even though this dynamicity is not implemented in the prototype, it is certainly an interesting aspect for future work. The compromise that this work makes instead, is that it offers the user an option via the UI to override the recommendations of the algorithm manually, if so desired. By doing that the user still has full control over the actual matching between the components and the instances for cases when the placement candidate returned by the algorithm doesn't seem to be optimal.

Nonetheless, other interesting strategies and heuristics were explored conceptually during this work, including the following ones, that may have been interesting for some of the use cases of this thesis:

As-distributed-as-possible: A strategy with the goal of spreading out the software components onto as many hardware components as possible. Such a strategy would maximize the resilience of the resulting application and distribute computational load optimally.

As-centralized-as-possible: A strategy with the goal of providing latency-sensitive applications the most performance possible. This would maximize the throughput of these components.

Preference for a certain resource: A strategy like this would allow the user to prioritize the conservation of a certain resource e.g., to try to rather waste CPU cores than RAM. I.e., this would be a special case of the least residue strategy.

4.3.5 Expressing Preferences via Requirements and Capabilities

As described earlier, the TOSCA specification supports RequirementTypes, CapabilityTypes and instances of these types called Requirements and Capabilities. These can be assigned to NodeTypes and NodeTemplates respectively. This means that apart from a generic and strategy-aware greedy algorithm, user preferences and strategies may also be expressed through Requirements and Capabilities.

The idea is to assign equally named Requirements and Capabilities to NodeTemplates that have to be placed and to NodeTemplates of their potential hosts, respectively. To achieve that efficiently, lists of NodeTemplateInstances that have a certain Capability are created first. These lists are then hashed into a map where the key is the name of a CapabilityType and the value is the list containing the IDs of all NodeTemplateInstances that have this Capability. By creating these lists first, the algorithm only ever looks at pairs of NodeTemplates with matching Requirements and Capabilities.

On the one hand this means that the user can limit the placement of certain components to a certain subset of instances that have the necessary Capabilities and thereby express a preference or strategy, on the other hand this increases the performance of the algorithm and reduces error proneness by guiding the placement process according to the user's preference.

Another way the complexity of the operation gets reduced is by assuming that the NodeTemplateInstances were marked as potential hosts in some way at modeling time. This can be a simple property that marks a NodeTemplate as an operating system node, for example. By doing this, the algorithm only looks at operating system nodes, instead of considering all of their successors and neighbors in a TopologyTemplate.

An example of such a pair of Requirements and Capabilities would be a NodeTemplate with the Requirement **"ReqLocationOnDevice"** accompanied by another NodeTemplate of a running instance with the Capability **"CapLocationOnDevice"**. This would, for example, guarantee that this specific application would be placed *on-device* assuming that the operating system NodeTemplate of the TopologyTemplate that represents the computational resources of the device has been enriched with this Capability and deployed with OpenTOSCA beforehand.

4.4 From Placement Matches to a Deployable CSAR

After the algorithm returns a placement candidate and the user confirms the selection via the UI, a topology completion task has to be performed, similar to previous work on splitting and matching of topologies by Saatkamp et al. [SBKL17]. This means that the ServiceTemplate that was selected for placement is edited by adding NodeTemplates of the same NodeType the matched instance has to the existing TopologyTemplate and drawing a ”hostedOn”RelationshipTemplate onto it. To identify the correct instances, the NodeTemplates of the added instances have to have a property that contains the IP address of the running instance, as well as a property of “state” with the value of “running”. This step is necessary because the OpenTOSCA container doesn't support dynamic editing of running instances. The ServiceTemplates are combined into a single one that doesn't have any open requirements anymore and can be deployed normally.

The OpenTOSCA container doesn't support the full TOSCA model either, which is why a separate instance of the Eclipse Winery – called *OpenTOSCA container repository* in the context of the container – was added as a supplementary component, so that the OpenTOSCA container can connect to it to delegate TOSCA modeling tasks. This is a compromise that is used in the OpenTOSCA ecosystem for now, to make use of the full TOSCA modeling capabilities of the Eclipse Winery. The connection to the container repository from the container is achieved by the WineryConnector, a class that was created specifically for this purpose. The OpenTOSCA container may have full TOSCA model interoperability in the future which would make this step unnecessary.

After the completion task is done, the container repository exports the newly merged CSAR which then is ready to be installed in the OpenTOSCA container and instantiated via the OpenTOSCA UI. The whole flow of actions across the different TOSCA tools that is required for the end-to-end operation of placing application components is described in more detail in the implementation chapter that follows.

5 A Prototypical Implementation of the ACP Concept

A goal of this work was to provide a prototype that would enable the placement of application components onto existing instances as discussed in Chapter 4 on page 39, using a TOSCA runtime such as OpenTOSCA.

This prototype was implemented in form of extensions to the existing OpenTOSCA ecosystem. Because the idea of ACP onto existing instances relies on the assumption that the instances were instantiated using OpenTOSCA and enhanced with respective Capabilities beforehand, such an environment had to be created in preparation for the prototype of this thesis.

This step of preparing the instances to fit the use cases of this thesis is described in the next section. After that the actual prototype is presented in more detail.

5.1 Modeling the TOSCA-Types for the Use Cases

To achieve this and to model the two types of nodes (application components vs. hardware instances), different ServiceTemplates were created, using the Eclipse Winery tool, to serve as minimal examples for the concept and as subjects for the prototypical implementation afterwards. This section presents the important parts of the two ServiceTemplates and their different variants that were created to demonstrate how the demands of the use cases presented in the introductory chapter could be satisfied accordingly.

Ubuntu on OpenStack

The *UbuntuOnOpenStack* ServiceTemplate used in this work is a modified version of an equally named ServiceTemplate from the OpenTOSCA/tosca-definitions-internal Github repository[1]. An overview of its topology was shown earlier in Figure 1.3 on page 20. The Ubuntu-VM NodeType that is part of this TopologyTemplate was duplicated and enriched by different CapabilityTypes for the different use cases, namely by a *CapLocationPrivateCloud*, a *CapLocationEdgeCloud* and a *CapLocationPublicCloud* CapabilityType that is used to inform the matching algorithm that it is a suitable host for applications that should be placed in the private cloud, edge cloud, and public cloud respectively.

[1] https://github.com/OpenTOSCA/tosca-definitions-internal

To validate the least-residue-strategy of the greedy placement algorithm of the prototype a key-value property called “AmountOfRAM” was added to the Ubuntu-VM NodeType as well. In the case of an instance it can be understood as the amount of **available** RAM this instance can provide for its applications.

The TOSCA standard supports key-value properties for Requirements and Capabilities as well but for the example of this thesis they were deliberately expressed as **NodeTemplate Properties** instead. The reasoning behind this choice is explained in more detail in the implementation details part that follows.

MyTinyToDo

The *MyTinyToDo* ServiceTemplate mocks the application components of the Driverless Transportation System that are to be placed onto the instances. An overview of its topology was shown earlier in Figure 1.2 on page 20. The *DockerEngine* and *MySQL-DBMS* NodeTypes that are part of it were enhanced with RequirementTypes that match the CapabilityTypes of the instances, to inform the algorithm that they have open requirements and want to be placed onto a running instance with matching Capabilities. To match the “AmountOfRAM” Properties of the instances, the DockerEngine and MySQL-DBMS NodeTemplates were given this property as well. In the case of an application it can be understood as the amount of RAM **required** for operation.

5.1.1 Overview of the Setup

To better understand the environment in which the prototype was tested, this subsection gives a quick summary of the various instances of the ServiceTemplates described in Section 5.1 on page 49 that were created for the purpose of testing the prototype. For simplicity the OpenStack NodeTemplate is omitted in the following summary as we are only interested in the NodeTemplateInstances that can become the targets of *hostedOn* RelationshipTemplates in the processes of placement i.e., in this case, the Ubuntu-VMs. Section 5.1.1 shows a tabular summary of the instances of this ServiceTemplate that were created including their respective Capabilities and Properties to simulate the existing IT-infrastructure of an industrial company.

Table 5.1: An overview of the VMs that were instantiated as the initial setup.

NodeTemplateInstance	Capability	Key-Value Property	Related Use Case
Ubuntu-VM #1	CapLocationPrivateCloud	{ “AmountOfRAM”: “32” }	DTS - Image Processing
Ubuntu-VM #2	CapLocationEdgeCloud	{ “AmountOfRAM”: “8” }	DTS - Engine Control
Ubuntu-VM #3	CapLocationPublicCloud	{ “AmountOfRAM”: “8” }	DTS - Sensor Data
Ubuntu-VM #4	CapLocationPublicCloud	{ “AmountOfRAM”: “16” }	DTS - Sensor Data

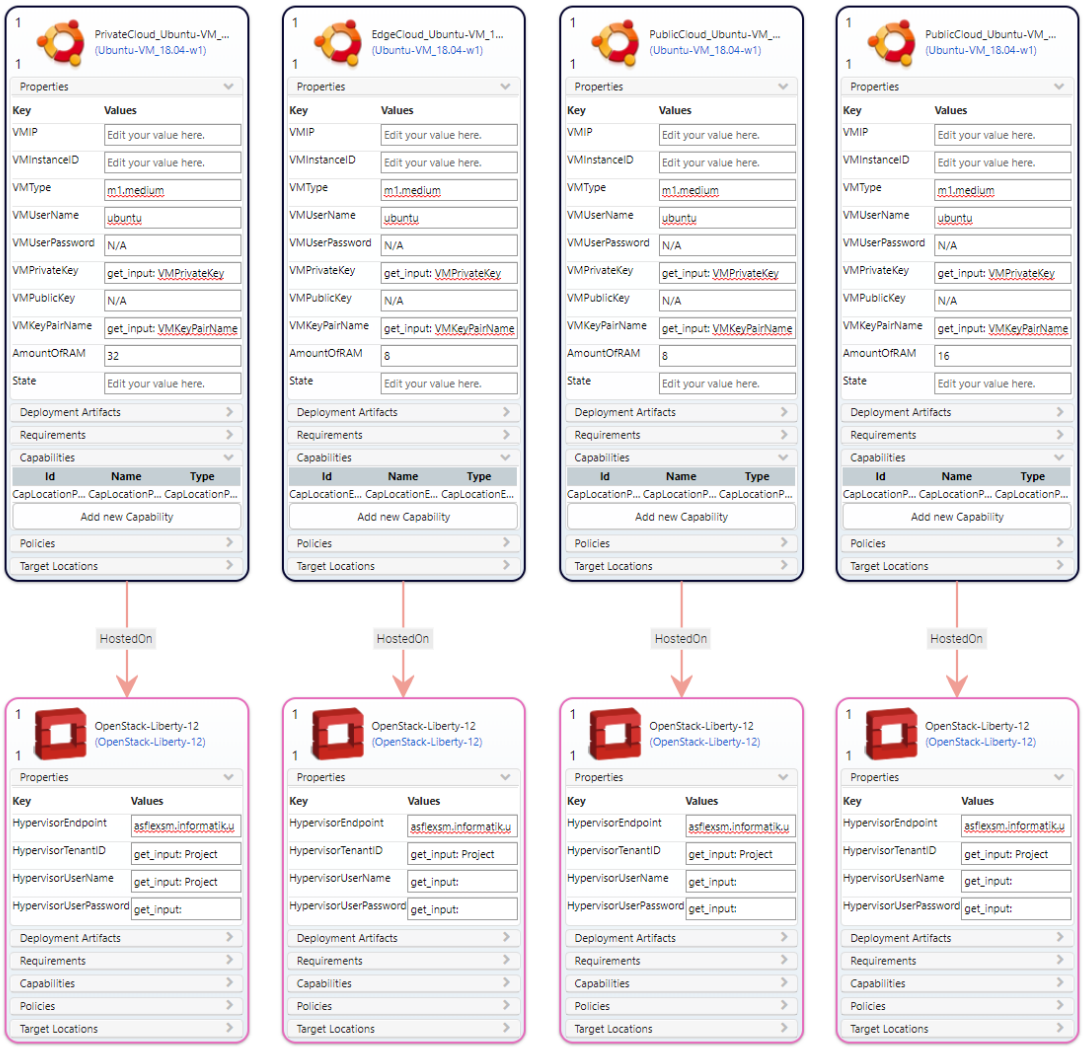
The rows of this table can be interpreted as follows:

- There is one VM that simulates the environment of the private cloud with 32GB of available RAM, and it is the intended destination for the image processing component.
- There is one VM that simulates the environment of the edge cloud with 8GB of available RAM, and it is the intended location for the engine control component.

- There are two VMs that simulate the environment of the public cloud with 8GB and 16GB of available RAM respectively, both with the Capabilities to host the sensor data analysis component.

A view of this TopologyTemplate is shown in Figure 5.1

Figure 5.1: An overview of the topology of the instances that were created for this prototype.



To complement these instances, two copies of the MyTinyToDo application were saved in a single ServiceTemplate and exported as a CSAR file. Each of the MyTinyToDo copies contains two leaf nodes with different Requirements and Properties, which results in the following 4 NodeTemplates that have to be placed, shown in Section 5.1.1 on the following page. The other NodeTemplates are omitted, as we are only interested in those with open requirements.

The rows of this table can be interpreted as follows:

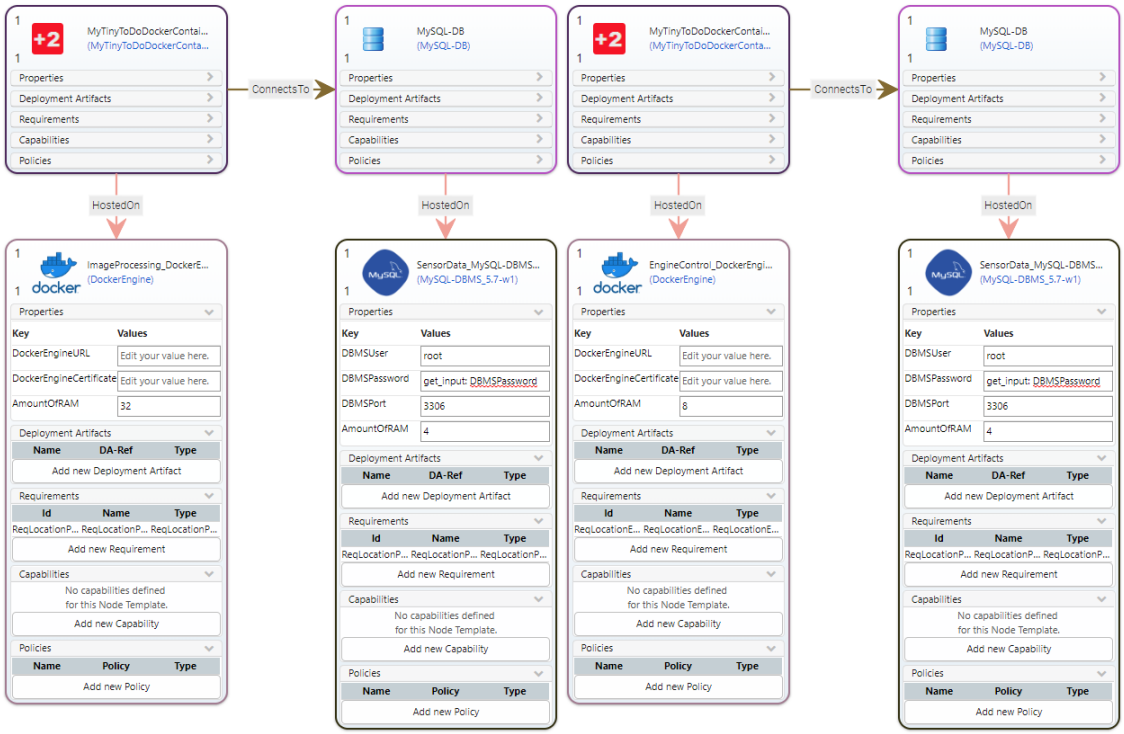
Table 5.2: An overview of the NodeTemplates that have to be placed.

NodeTemplate	Requirement	Key-Value Property	Related Use Case
DockerEngine #1	ReqLocationPrivateCloud	{ “AmountOfRAM”: “32” }	DTS - Image Processing
DockerEngine #2	ReqLocationEdgeCloud	{ “AmountOfRAM”: “8” }	DTS - Engine Control
MySQL-DBMS #1	ReqLocationPublicCloud	{ “AmountOfRAM”: “4” }	DTS - Sensor Data
MySQL-DBMS #2	ReqLocationPublicCloud	{ “AmountOfRAM”: “4” }	DTS - Sensor Data

- There is one DockerEngine NodeTemplate that simulates the image processing component that requires 32GB of RAM and has the Requirement that tells the algorithm that it has to be deployed in the private cloud.
- There is one DockerEngine NodeTemplate that simulates the engine control component that requires 8GB of RAM and has the Requirement that tells the algorithm that it has to be deployed in the edge cloud.
- There are two MySQL-DBMS NodeTemplates that simulate the sensor data analysis components that each require 4GB of RAM, both with the the Requirement that tells the algorithm that they have to be deployed in the public cloud.

An overview of this TopologyTemplate is shown in Figure 5.2.

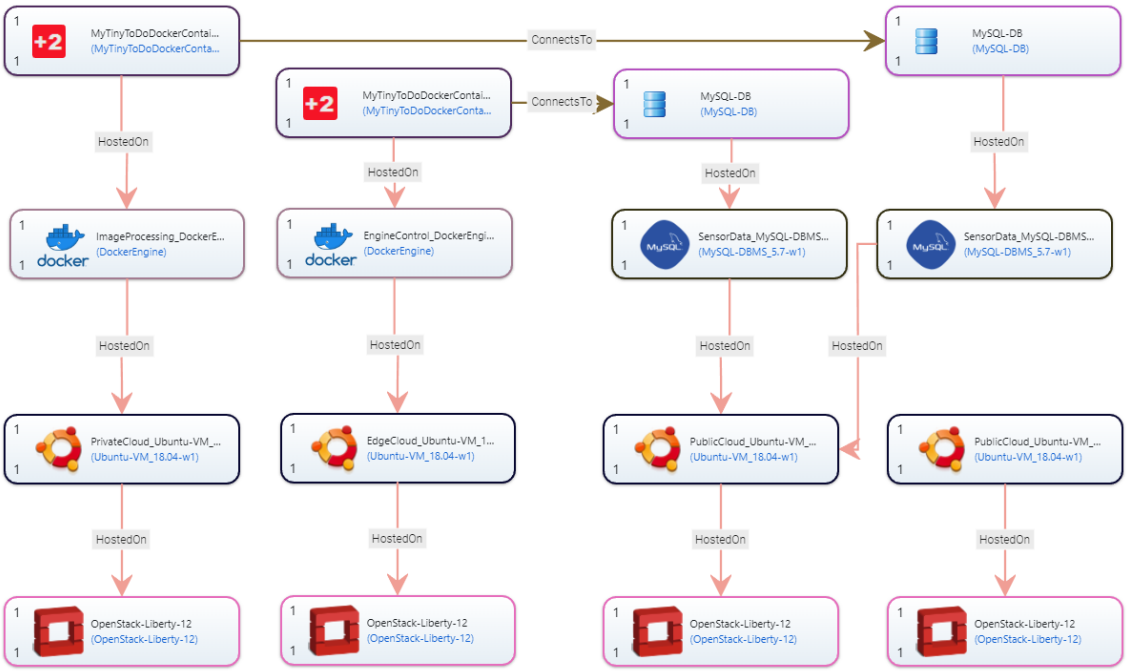
Figure 5.2: An overview of the topology of the applications that have to be placed.



Since the algorithm implements the least-residue-strategy explained in Section 4.3.3 on page 44 no unnecessary resources should be wasted. Because both sensor data analysis components (represented by the MySQL-DBMS NodeTemplates) require 4GB of RAM and the Ubuntu instance #3 offers

exactly 8GB of available RAM via its Properties, the last Ubuntu instance should not be part of the final placement candidate, as its resources are not required in an optimal solution. Therefore Figure 5.3 shows the desired topology of the placement candidate in this example.

Figure 5.3: An overview of the desired topology of the optimal placement candidate.



5.2 Implementation Details

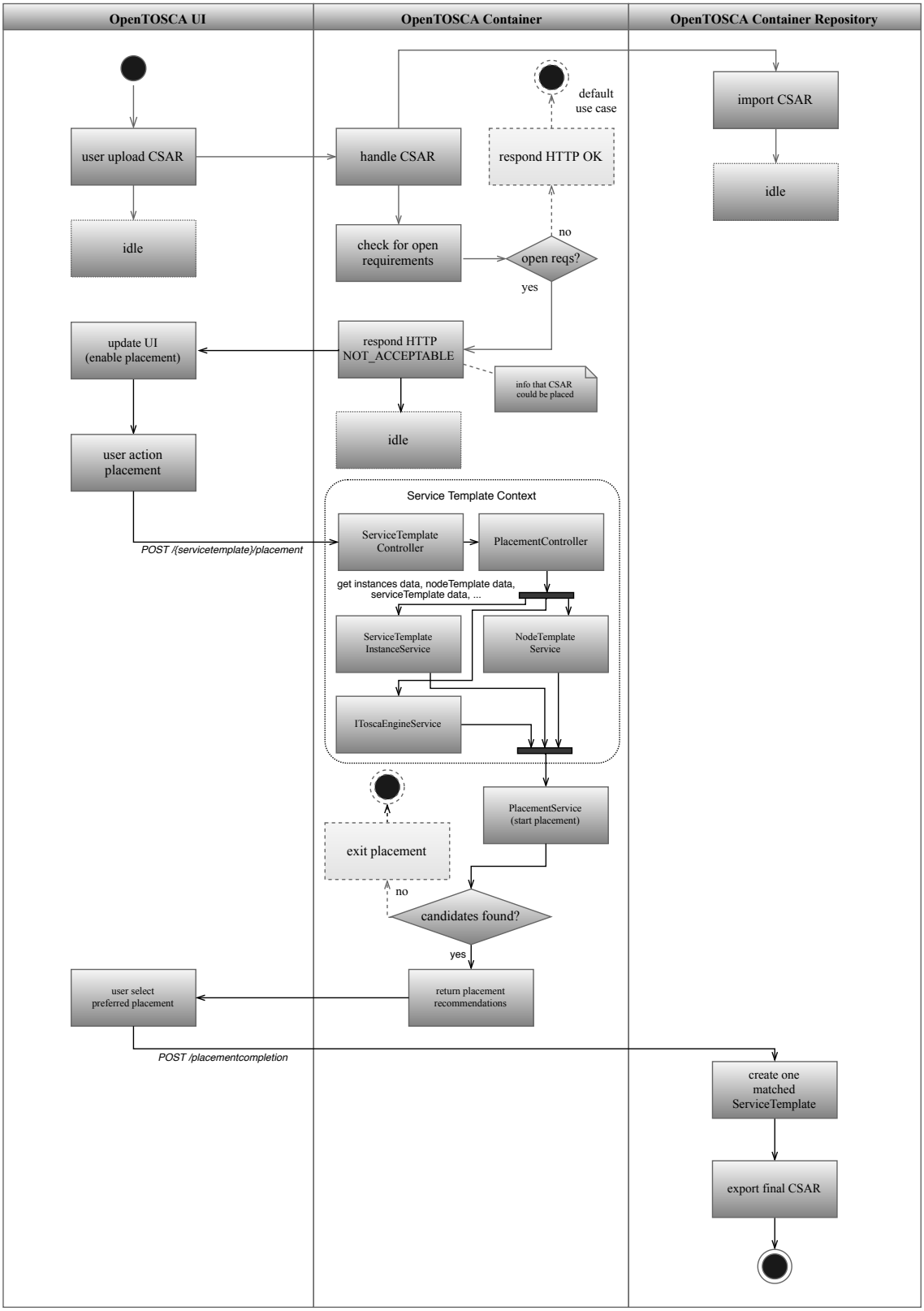
The sophistication of the prototype and the ServiceTemplates used in this example is limited to fit into the scope of this thesis. Therefore some features described in the concept chapter were simplified or changed for the prototypical implementation, to be extended by future contributors and to adjust to the idiosyncrasies of the existing OpenTOSCA feature set. Existing interfaces and methods were used whenever possible to maximize the reusability of existing methods and to avoid code duplicates.

The placement functionality on the backend – inside the OpenTOSCA container – was introduced in a plug-in manner by adding a new bundle to the existing Maven project. The functionality on the frontend – inside the OpenTOSCA UI – was provided by creating new Angular components where necessary and by extending existing HTML templates, reducer functions and event handlers. This approach provides a loosely coupled addition to the existing system that can be extended and / or replaced in the future.

This chapter describes these changes to the OpenTOSCA UI, the OpenTOSCA Container as well as the Eclipse Winery in detail. The structure of this section loosely follows the chronological order of actions required for placing an application starting from the CSAR upload in the OpenTOSCA UI. This means that the changes made to each OpenTOSCA tool will be described by going

through each of the steps from uploading a CSAR until the placement of an application is achieved. Where necessary, an overview of the original feature set will be provided to create a contrast, before explaining the changes that were done as part of this thesis. An overview of the end-to-end choreography that results from the interaction between the different OpenTOSCA tools for placing an application is depicted in the diagram shown in Figure 5.4 on the facing page.

Figure 5.4: The choreography between the different OpenTOSCA tools when placing an application.



5.3 Additions and Changes to the OpenTOSCA UI

This section goes beyond the brief description of the interaction between the OpenTOSCA UI and the container that was provided in Chapter 2 on page 25 to understand which changes had to be made to the existing code. As mentioned earlier, when the user uploads a CSAR via the UI, it gets sent to the REST API of the OpenTOSCA container where it is processed further. Since a CSAR that requires placement onto existing instances is not well-defined by nature, meaning that it is not instantiable before getting matched with instance nodes, it de facto has open requirements.

Open Requirements

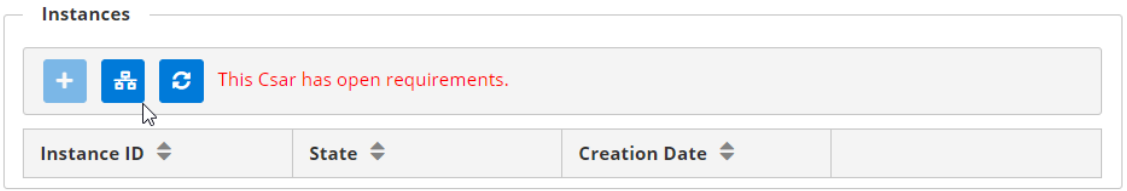
An application can have open requirements e.g., when the target environment of an application is not known in advance i.e., at modeling time [SBK+18]. The OpenTOSCA container is able to detect open requirements in ServiceTemplates as the CSAR gets uploaded. The container detects open requirements if the number of outgoing RelationshipTemplates is smaller than the number of Requirements.

Originally the container would reject such CSARs and return an error to the OpenTOSCA UI following a deletion of the uploaded CSAR. Since open requirements are always present in the use case of application placement, this behavior had to be changed.

With changes to the CsarController's *handleCsar()* function, the container now accepts CSARs with open requirements and returns a flag inside its HTTP response to the UI that informs it that a placement of the application can be initiated.

After receiving this response, the updated UI then enables the additional placement button next to the instantiation button (that now gets disabled in the case of open requirements) to avoid the instantiation of uncompleted ServiceTemplates. This feature is demonstrated in the screenshot shown in Figure 5.5.

Figure 5.5: The new placement button is enabled when a CSAR has open requirements.



Since the UI only gets this information once, namely when the CSAR gets uploaded, the *placeability* of a CSAR had to be kept in the state of the web application to correctly reflect which CSAR is instantiable and which one is placeable, as this would otherwise get lost during a page refresh.

This functionality was achieved by storing references to each placeable CSAR inside an array of objects that consist of the *CsarId* as well as a *"placementPossible"* flag inside the *localStorage* of the browser. Whenever the detailed view of a CSAR is opened by the user, the *onInit()* lifecycle

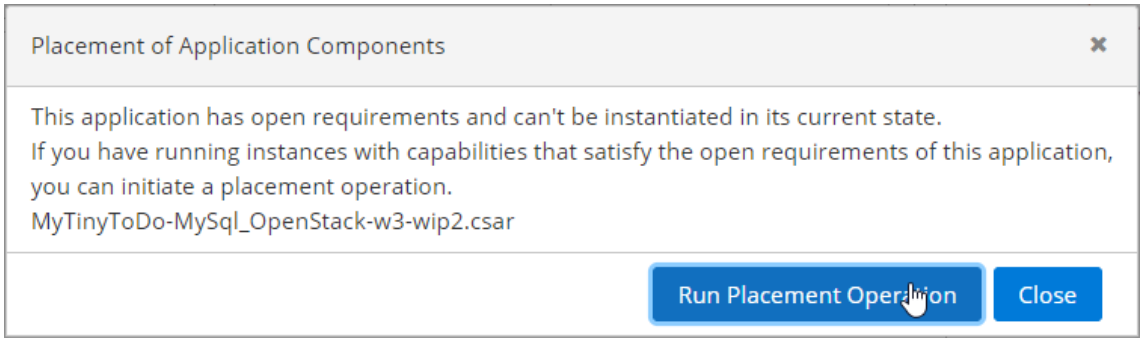
method of this Angular component now takes care of loading the respective array of placeable CSARs from localStorage and dispatching an update action to the Redux[2] reducer function that updates the inner application state of the currently viewed CSAR accordingly.

With this change, a CSAR with open requirements can be kept inside the UI permanently and is always distinguished correctly from CSARs without open requirements, as long as the localStorage of the browser doesn't get emptied by a third party. A more persistent option that reflects the placeability of a CSAR and that is based on a single source of truth should be favored over this approach in the future.

Initiating the Placement Algorithm from the UI

When the user clicks the placement button by interacting with the placement dialog shown in the screenshot in Figure 5.6, another HTTP request to the REST API of the container is sent with the intent of trying to place this application's components onto existing instances.

Figure 5.6: The placement dialog offering the initiation of a placement operation to the user.



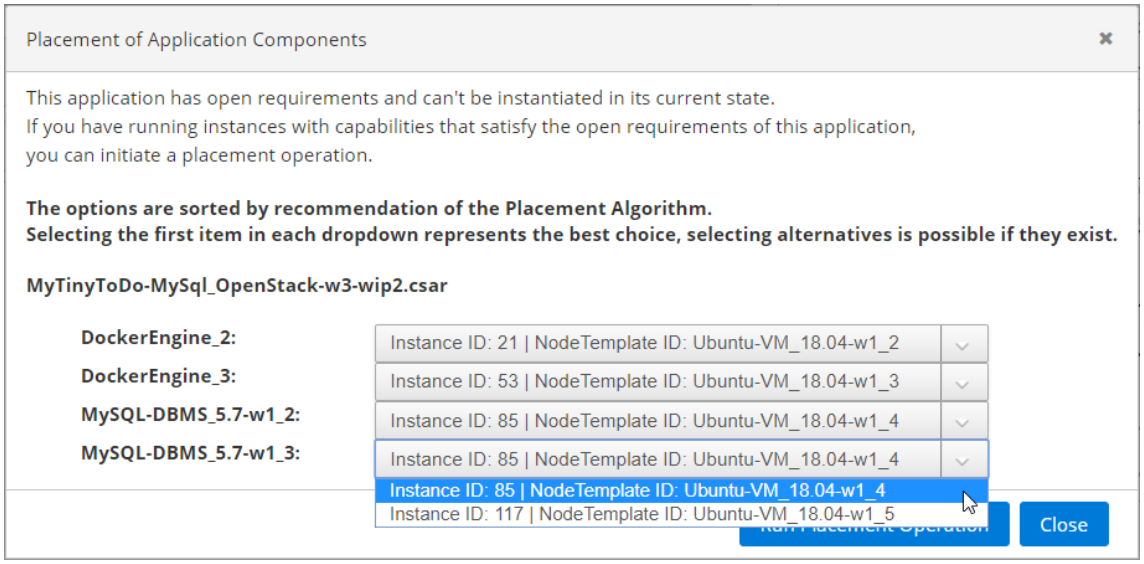
If the container finds a suitable placement candidate, it is returned to the OpenTOSCA UI along with all alternative placement locations mapped to each NodeTemplate that has to be placed. Like this, the user retains control about the final decision of placement and can inspect the solution of the algorithm before the instantiation of the NodeTemplates that have to be placed.

Figure 5.7 on the following page shows the set of dropdown lists, containing alternative placement locations for each NodeTemplate that the user can choose from.

Since the least residue strategy is only applied on the backend (as will be explained shortly) any placement locations the user might choose manually from these dropdown lists, may lead to cases where the "AmountOfRAM" property of an instance has a smaller value than the sum of the same property's values of the NodeTemplates the user chooses to place on it. Therefore future work may consider the implementation of a feature that keeps track of the amount of resources of each instance throughout the whole end-to-end process. The following section describes the additions and changes that were done to the OpenTOSCA container as part of the prototype of this thesis.

[2] https://redux.js.org/

Figure 5.7: Dropdown list with placement locations for each NodeTemplate that has to be placed.



5.4 Additions and Changes to the OpenTOSCA Container

Originally the container did not support any CSARs that had open requirements, and would reject them on upload. Now with the changes to the **CsarController** inside the *org.opentosca.controller.api* bundle these CSARs don't get deleted anymore and a response to the OpenTOSCA UI is sent containing the information that a placement operation can be initiated for this CSAR.

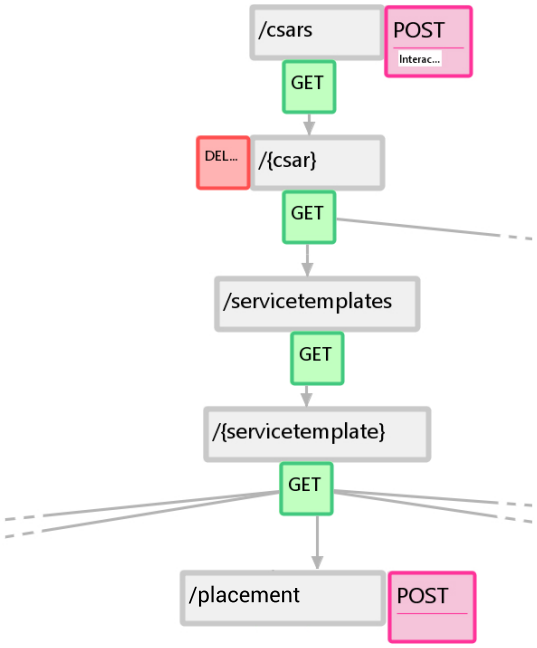
The existing container REST API, that was shown earlier in Figure 2.4 on page 32 was extended by a POST method at */csars/{csar}/servicetemplates/{servicetemplate}/placement*. This change is demonstrated in the diagram in Figure 5.8 on the facing page, that shows an excerpt of the container API. Sending a PORT request to this path activates the PlacementController that was added to the container API bundle to handle these requests which then calls the PlacementService inside the new bundle *org.opentosca.placement* that was added to the existing OSGi framework and root Maven project respectively, to handle the placement operation itself.

The various *Services* in the OpenTOSCA container support the retrieval of created instances as well as the retrieval and modification of *ServiceTemplateInstance* and *NodeTemplateInstance* Properties in line with the TOSCA standard. This is necessary for the PlacementService to be able to distinguish different instances of the same ServiceTemplates and NodeTemplates respectively, since they get assigned an unique ID on instantiation.

To limit the search space during the search for viable host instances for the costly placement algorithm the prototype makes use of an existing helper method that only returns NodeTemplateInstances which qualify as **operating system nodes**. With this assumption and limitation the algorithm doesn't have to visit all nodes and its performance is increased significantly.

As mentioned earlier in the OpenTOSCA ecosystem, Properties are used to describe idiosyncrasies of certain NodeTemplates at modeling time. For example, they can be expressed in terms of key-value pairs of strings such as:

Figure 5.8: The added path for initiating a placement operation via a HTTP POST request.



{ "location": "onPremise"}.

As mentioned in Chapter 4 on page 39 the matching of Requirements and Capabilities in this prototype is done between NodeTemplate Requirements and Capabilities rather than between one NodeTemplateInstance's Capabilities and one NodeTemplate's Requirements. The latter option allows for mathematical operations on the values of Requirements and Capabilities that reflect some resource availability e.g., the subtraction of a certain amount of available RAM from a NodeTemplateInstance's Capability, after having placed a NodeTemplate with a certain amount of required RAM onto it.

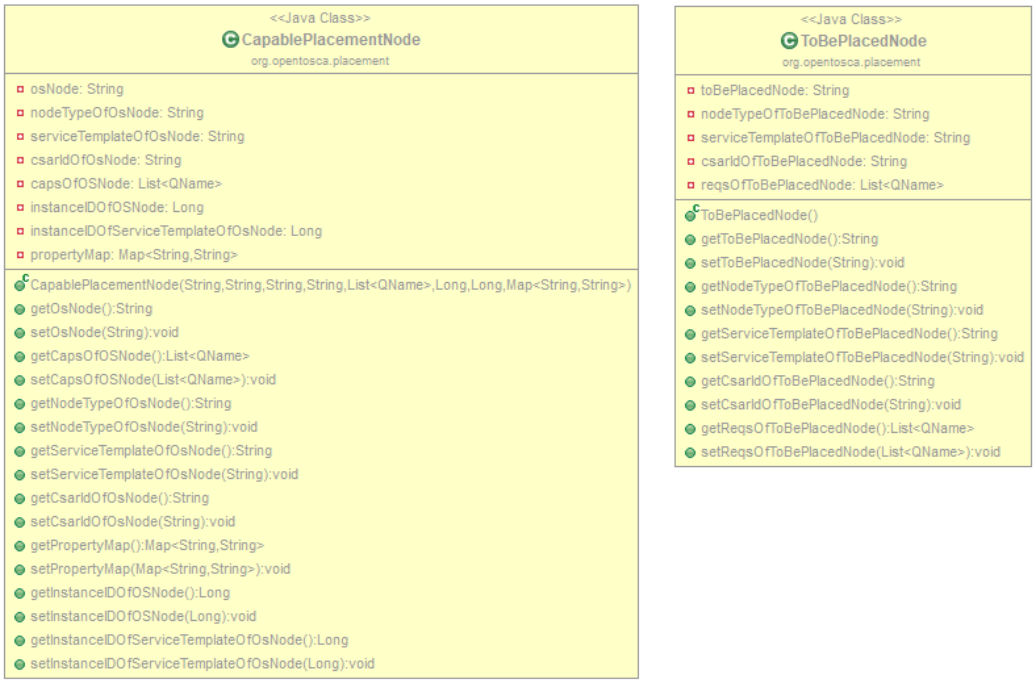
This allows keeping track and adjusting of values of Capabilities dynamically during the operation of the algorithm and beyond that. On top of that, this represents a reasonable approach to constructing a cost function. Since this is also possible via a NodeTemplateInstances's Properties rather than its Capabilities and since the OpenTOSCA container already supports the retrieval and modification of properties, this option was chosen over implementing methods for the retrieval of Requirement Properties and Capability Properties. The running example throughout this thesis has only one such property named ÄmountOfRAM". Obviously, this simplifies the complexity of the cost function implemented in this thesis, compared to cost functions where a multitude of such properties would have to be considered at the same time.

5.4.1 PlacementController

The PlacementController lives inside the context of its ServiceTemplate by default due to the HATEOAS [Fie00] maturity level of the OpenTOSCA container's REST API. This allows the retrieval of all necessary information that the PlacementService needs for finding suitable placement

candidates such as the CsarId, the ServiceTemplateId as well as its Requirements, Capabilities and Properties. It handles incoming placement requests and calls the PlacementService with all the information it requires. To achieve that, the PlacementController searches for all NodeTemplateInstances as well as all NodeTemplates of the ServiceTemplate that have to be placed e.g., that have open requirements. For each NodeTemplateInstance that represents an operating system and that has some Capabilities, an Object of the class *CapablePlacementNode* is created. For each NodeTemplate with open requirements an Object of the class *ToBePlacedNode* is created. Figure 5.9 shows an UML diagram of these two classes which contain all the required attributes the PlacementService needs, to perform the matching and to identify each instance precisely. These Objects are then added to their respective ArrayLists shown in Listing 5.1 and handed to the PlacementService which implements the actual placement operation.

Figure 5.9: UML class diagram of the classes of ToBePlacedNode and CapablePlacementNode



Listing 5.1 Excerpt from *PlacementController.java*

```
1  List<CapablePlacementNode> listOfCapablePlacementNodes = new ArrayList<CapablePlacementNode>();
2  List<ToBePlacedNode> listOfToBePlacedNodes = new ArrayList<ToBePlacedNode>();
```

5.4.2 PlacementService

The PlacementService contains the actual algorithmic approach that was implemented in this prototype. In the first step, it compares the Capabilities of the CapablePlacementNodes with the Requirements of the ToBePlacedNodes and matches them by comparing the names of their Requirements and Capabilities assuming that they were named equally following their different

prefixes of “Req” and “Cap”. This results in a HashMap of Lists of CapablePlacementNodes for each Capability shown in Listing 5.2. This map corresponds to the collection of subsets of the universe discussed in Section 4.3 on page 42 and therefore presents the NodeTemplate and NodeTemplateInstance data to the algorithm in a compatible format.

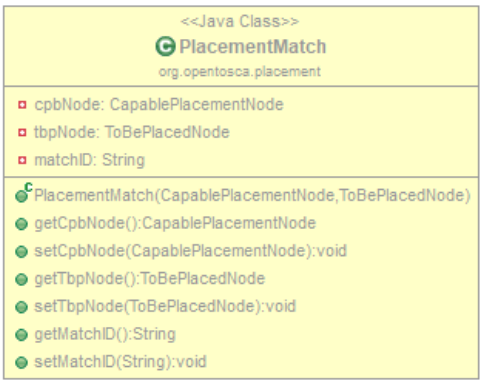
Listing 5.2 Excerpt from *PlacementService.java* I

```
1  /**
2  *  This map contains all nodes with a certain capability in the form of:
3  *  { key: value } ==> { capName: List<CapablePlacementNode> }
4  */
5  Map<String, List<CapablePlacementNode>> capNamesToCapableNodes
6    = new HashMap<String, List<CapablePlacementNode>>();
```

In the next step it compares the Properties of each NodeTemplate that has to be placed to those of each CapablePlacementNode in a for-loop. It only continues if there are Properties with matching keys and then calculates the computational residue out of the values of these Properties that would result from a match between these two nodes. In the use case that was constructed for the initial testing of this prototype, this Property was the AmountOfRAM property. The residue is kept in a local variable that resides outside of this for-loop and it is updated only if a smaller residue is found between the NodeTemplate and another CapablePlacementNode. A chosenHost variable keeps track of the currently optimal choice and it also resides outside of this for-loop.

If a better match is found i.e., one that produces less residue, the previously chosenHost is added to the list of alternative hosts and the new, locally optimal host is reassigned to the chosenHost variable. The result is a list of PlacementMatches that represents the placement candidate as well as a second list that contains potential alternative matches which are presented to the user via the UI if the user wants to change the decisions made by the algorithm. An UML class diagram of the class PlacementMatch is shown in Figure 5.10.

Figure 5.10: UML class diagram of the class PlacementMatch



Listing 5.3 on the next page shows a code excerpt of the important parts of the PlacementService that demonstrates the implementation of the behavior that was just described.

Listing 5.3 Excerpt from *PlacementService.java* II

```
1     public PlacementCandidate findPlacementCandidate(final List<CapablePlacementNode> cpbNodes,
2     final List<ToBePlacedNode> tbpNodes) {
3
4     List<PlacementMatch> results = new ArrayList<PlacementMatch>();
5     List<PlacementMatch> alternativeMatches = new ArrayList<PlacementMatch>();
6     Map<String, List<CapablePlacementNode>> capNamesToCapableNodes
7       = new HashMap<String, List<CapablePlacementNode>>();
8
9     capNamesToCapableNodes.forEach((cap, nodesWithThisCap) -> {
10      tbpNodes.forEach(tbpNode -> {
11        tbpNode.getReqsOfToBePlacedNode().forEach(req -> {
12
13          // Get the important part of the Requirement name for matching with Capabilities
14          List<String> reqStrings
15            = splitPrefixFromReqOrCap(removeNumbersFromString(req.getLocalPart()));
16          String reqName = reqStrings.get(1);
17
18          nodesWithThisCap.forEach(cpbNode -> {
19            // only look at cases where capabilities and requirements match
20            ...
21              // 1. calculate residue of properties
22              // 2. find out intersection of properties
23              // 3. if there are common properties
24              if (intersect != null) {
25                intersect.iterator().forEachRemaining(propertyKey -> {
26                  // if enough resources are available
27                  ...
28                  if (amountCap >= amountReq) {
29                    // first iteration
30                    if (chosenHost == null) {
31                      residue = amountCap - amountReq;
32                      chosenHost = cpbNode;
33                    } else if (chosenHost != null) {
34                      Integer myResidue = amountCap - amountReq;
35                      if (myResidue < residue) {
36                        residue = myResidue;
37                        // add as alternative host since it got beaten by the latest cpbNode
38                        alternativeHost = chosenHost;
39                        chosenHost = cpbNode;
40                      } else {
41                        alternativeHost = cpbNode;
42                      }
43                    }
44                    cpbNode.getPropertyMap().put(propertyKey, residue.toString());
45                  }
46                });
47              }
48            }
49          });
50          // create PlacementMatch if one was found
51          ...
52        });
53      });
54    });
55    return new PlacementCandidate(results, alternativeMatches);
56  }
```

5.5 Additions and Changes to the Eclipse Winery

The OpenTOSCA container depends on an instance of the Eclipse Winery, also called OpenTOSCA container repository, for advanced TOSCA model manipulation functionalities, as mentioned earlier in Section 4.4 on page 48. The last step from the PlacementCandidate to a deployable CSAR is to inject NodeTemplates of the NodeTemplateInstances into the Topology of the uploaded CSAR that contains the application components that have to be placed. The Winery REST API doesn't follow the HATEOAS approach but rather organizes the URLs in the following way:

/<type>/<encoded namespace>/<encoded id>.

The type can be a ServiceTemplate, a NodeTemplate, a NodeType, etc. It is uniquely identified by providing its namespace and id after that.

For this last step a POST method was added to the existing REST API of the Winery that resides behind the ServiceTemplateResource and the **placement/completion** path. It is inspired by similar methods which include TopologyTemplate manipulation that were implemented as part of earlier work by Saatkamp et al. [SBKL17], [SBK+18], [SBKL19].

Firstly the NodeTemplates of the chosen hosts in the placement candidate are retrieved from the repository of the Winery where they were stored initially when the CSARs of the instances were uploaded. They are copied over to the TopologyTemplate that contains the application components and *hostedOn* connections from the NodeTemplates that have to be placed, are drawn to the chosen hosts.

Another important step is that a couple of Properties of the newly added NodeTemplates of the instances have to be set before the CSAR gets exported. Firstly, the **"{ State: Running }"** property has to be set which tells the OpenTOSCA container that the instances that the application components will be hosted on are already existent and running. Secondly, the Property that contains the IP address of the Ubuntu VM has to be set, so that the VM can be found by the OpenTOSCA container eventually. Finally, the CSAR that results from this operation can be exported utilizing the existing CSAR-export functionality of the Winery. This CSAR can then be deployed via the OpenTOSCA UI in the usual manner, completing the end-to-end workflow of ACP in the OpenTOSCA ecosystem.

6 Conclusion and Outlook

This thesis presented a concept of an algorithmic approach that tries to tackle the Application Component Placement (ACP) problem onto running instances. It showed that the placement of TOSCA modeled applications onto running instances in the context of the OpenTOSCA container is possible.

The running example throughout this thesis that revolves around the integration of a Driverless Transport System (DTS) into the existing infrastructure of a smart factory was introduced in Chapter 1 on page 17. Dummy applications were used as placeholders for the characteristic application components of DTS such as an image processing component, an engine control component and a sensor data analysis component. Dummy instances of Ubuntu VMs on an OpenStack instance were set up to simulate different placement locations these components can be offloaded to. The applications and instances were enhanced with TOSCA-conform Requirements and Capabilities, namely ones that represent a desired placement location such as the edge cloud, the private cloud and the public cloud, as well as a *NodeTemplate Property*, namely an "AmountOfRAM" property, that was used to define an application component's amount of required RAM and an instances amount of available RAM.

By making use of an approximate greedy algorithm with a heuristic that follows a least residue strategy and by considering the amount of available resources as the subject of a cost function, it is ensured that a reasonably optimized solution can be provided by the prototypical proof-of-concept implementation that was created as part of this work.

The prototype was implemented in terms of extensions to the existing OpenTOSCA ecosystem. As the goal was to provide a proof-of-concept implementation that represents a minimal end-to-end implementation, changes were made to three different components, namely the OpenTOSCA container, the OpenTOSCA UI and the Eclipse Winery (OpenTOSCA container repository).

The OpenTOSCA UI was enhanced by a feature that treats uploaded CSARs with open requirements differently than well-defined ones, by enabling an additional placement button for these cases. A couple of HTTP methods were implemented in the OpenTOSCA UI to communicate with the other OpenTOSCA tools. Firstly, a method that enables the initiation of the placement operation inside the OpenTOSCA container and secondly a method that sends the chosen placement candidate to the Winery for topology completion. The overruling of the placement candidate by the user via the OpenTOSCA UI was made possible through the provided placement dialog, removing any source of coercion that may be caused by blindly applying the solution returned by the placement algorithm.

The OpenTOSCA container was enhanced by an additional OSGi bundle that offers the placement operation for ServiceTemplates with open requirements. This was done by having implemented a version of the aforementioned algorithm which can be initialized from the context of an uploaded CSAR in the OpenTOSCA UI via the new resource path that was added to the existing REST API of the OpenTOSCA container.

The Eclipse Winery was extended by a method that converts the map between NodeTemplates that have to be placed and running instances into a deployable CSAR by completing the CSAR that was selected for placement with the respective NodeTemplates of the running instances and pointing to them by providing their IP addresses.

The proof-of-concept prototypical implementation was tested initially using four application components and four instances of VMs, demonstrating the capabilities and acceptable performance of the algorithmic solution for reasonably small problem sizes.

Outlook

Future work should consider substituting the mocked availabilities of resources with real data, by combining the concepts of monitoring, crawling and placement, potentially even considering the modeling of the underlying networking capabilities of certain resources. This would be an important step forward from the experimental stage of this prototype to a more generally applicable solution. Defining a clear set of rules for how placement preferences can be expressed by users at modeling time should be another goal of future work, to minimize the amount of assumptions the prototypical implementation had to make.

Since the proof-of-concept implementation of this thesis only uses a single resource that the algorithm optimizes for (AmountOfRAM), future work should consider enhancing the algorithm with a more sophisticated variant of a cost function that can work with the multi-dimensionality when comparing for a number of such properties concurrently or that makes use of dimensionality-reducing techniques such as Principal Component Analysis to reduce the complexity for the algorithm. This would enable a system where bad placement candidates are being punished more during the process of dimensionality reduction by making use of weighted cost functions, which would imply that the placement candidate that fits the model the best i.e., “that hurts the least” in terms of not being an outlier could be selected at every iteration of the algorithm.

Since the user can override the placement candidate recommended by the algorithm via the UI and potentially choose a placement location for a component that does not have enough available resources, future work could prevent that by extending the instance API of the OpenTOSCA container by a feature that returns the actual amount of available system resources to provide a single source of truth for both the UI and the placement algorithm.

The different strategies discussed conceptually in Chapter 4 on page 39 all have their *raisons d’être* in their respective potential use cases, therefore it may be interesting for future work to make the placement algorithm more generic by enabling the provision of different strategies and heuristics in a plugin manner.

Earlier work has shown that distributed ACP solver algorithms beat the performance of centralized ones. If the need for a more performant variant of the placement algorithm arises, a distributed algorithm could be developed by future work.

Bibliography

[ABN+17] M. Artac, T. Borovssak, E. D. Nitto, M. Guerriero, D. A. Tamburri. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017. DOI: 10.1109/icse-c.2017.162 (cit. on p. 26).

[ASLW14] V. Andrikopoulos, S. G. Sáez, F. Leymann, J. Wettinger. "Optimal distribution of applications in the cloud". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2014, pp. 75–90 (cit. on p. 38).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications". In: *Service-Oriented Computing*. Ed. by S. Basu, C. Pautasso, L. Zhang, X. Fu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 692–695. ISBN: 978-3-642-45005-1 (cit. on p. 30).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. "Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA". In: *On the Move to Meaningful Internet Systems: OTM 2012*. Springer Berlin Heidelberg, 2012, pp. 416–424. DOI: 10.1007/978-3-642-33606-5_25 (cit. on p. 19).

[BBKL13a] T. Binz, U. Breitenbucher, O. Kopp, F. Leymann. "Automated Discovery and Maintenance of Enterprise Topology Graphs". In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE, Dec. 2013. DOI: 10.1109/soca.2013.29 (cit. on p. 35).

[BBKL13b] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. Springer New York, Aug. 2013, pp. 527–549. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 26).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. *Vinothek - A Self-Service Portal for TOSCA*. Mar. 2014 (cit. on p. 30).

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. "Portable Cloud Services Using TOSCA". English. In: *IEEE Internet Computing* 16.03 (May 2012), pp. 80–85. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.43. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2012-09&engl=1 (cit. on p. 26).

[Bin15] T. Binz. *Crawling von Enterprise Topologien zur automatisierten Migration von Anwendungen : eine Cloud-Perspektive*. de. 2015. DOI: 10.18419/opus-3543 (cit. on pp. 25, 35, 36).

[Bla98] P. E. Black. *Dictionary of Algorithms and Data Structures| NIST*. Tech. rep. 1998 (cit. on p. 44).

[BML+17] M. Barshan, H. Moens, S. Latre, B. Volckaert, F. D. Turck. “Algorithms for network-aware application component placement for cloud resource allocation”. In: *Journal of Communications and Networks* 19.5 (Oct. 2017), pp. 493–508. DOI: 10.1109/jcn.2017.000081 (cit. on p. 38).

[BMQ+07] G. Boss, P. Malladi, D. Quan, L. Legregni, H. Hall. “Cloud computing”. In: *IBM white paper* 321 (2007), pp. 224–231 (cit. on p. 25).

[BMT14] M. Barshan, H. Moens, F. D. Turck. “Design and evaluation of a scalable hierarchical application component placement algorithm for cloud resource allocation”. In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. IEEE, Nov. 2014. DOI: 10.1109/cnsm.2014.7014155 (cit. on p. 38).

[CJLF16] X. Chen, L. Jiao, W. Li, X. Fu. “Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing”. In: *IEEE/ACM Transactions on Networking* 24.5 (Oct. 2016), pp. 2795–2808. DOI: 10.1109/tnet.2015.2487344 (cit. on p. 17).

[CSW+08] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguade. “Utility-based placement of dynamic Web applications with fairness goals”. In: *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*. IEEE, 2008. DOI: 10.1109/noms.2008.4575111 (cit. on p. 38).

[Fie00] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000 (cit. on p. 59).

[GJF+16] D. Georgakopoulos, P. P. Jayaraman, M. Fazia, M. Villari, R. Ranjan. “Internet of Things and Edge Cloud Computing Roadmap for Manufacturing”. In: *IEEE Cloud Computing* 3.4 (July 2016), pp. 66–73. DOI: 10.1109/mcc.2016.91 (cit. on p. 17).

[Hoc82] D. S. Hochbaum. “Approximation Algorithms for the Set Covering and Vertex Cover Problems”. In: *SIAM Journal on Computing* 11.3 (Aug. 1982), pp. 555–556. DOI: 10.1137/0211045 (cit. on p. 44).

[JCL11] Z. Jin, J. Cao, M. Li. “A Distributed Application Component Placement Approach for Cloud Computing Environment”. In: *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, Dec. 2011. DOI: 10.1109/dasc.2011.94 (cit. on p. 38).

[Kar72] R. M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on p. 42).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery – A Modeling Tool for TOSCA-Based Cloud Applications”. In: *Service-Oriented Computing*. Springer Berlin Heidelberg, 2013, pp. 700–704. DOI: 10.1007/978-3-642-45005-1_64 (cit. on p. 29).

[KSST05] T. Kimbrel, M. Steinder, M. Sviridenko, A. Tantawi. “Dynamic Application Placement Under Service and Memory Constraints”. In: *Experimental and Efficient Algorithms*. Springer Berlin Heidelberg, 2005, pp. 391–402. DOI: 10.1007/11427186_34 (cit. on p. 38).

[Mor18] J. Morello. *The Continuum of Cloud-Native Topologies*. 2018. URL: https://thenewstack.io/continuum-cloud-native-topologies/ (cit. on p. 18).

[OAS14] OASIS. *TOSCA v1.0*. 2014. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf (cit. on pp. 26, 27).

[Ope] OpenTOSCA. *OpenTOSCA container - Github Repository*. URL: https://github.com/OpenTOSCA/container (cit. on pp. 30, 32, 33).

[PX13] C. Pahl, H. Xiong. "Migration to PaaS clouds - Migration process and architectural concerns". In: *2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*. IEEE, Sept. 2013. DOI: 10.1109/mesoca.2013.6632740 (cit. on p. 35).

[RRR+16] R. Rahim, R. Rahim, R. Rahim, R. Rahim, R. Rahim, R. Rahim, R. Rahim, R. Rahim. "The OpenTOSCA Ecosystem - Concepts & Tools". In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*. SCITEPRESS - Science and Technology Publications, 2016. DOI: 10.5220/0007903201120130 (cit. on p. 29).

[SBK+18] K. Saatkamp, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann. "OpenTOSCA Injector: Vertical and Horizontal Topology Model Injection". In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Springer International Publishing, 2018, pp. 379–383. DOI: 10.1007/978-3-319-91764-1_34 (cit. on pp. 56, 63).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Topology Splitting and Matching for Multi-Cloud Deployments". In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258. ISBN: 978-989-758-243-1 (cit. on pp. 48, 63).

[SBKL19] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Method, formalization, and algorithms to split topology models for distributed cloud application deployments". In: *Computing* (Apr. 2019). DOI: 10.1007/s00607-019-00721-8 (cit. on p. 63).

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. DOI: 10.1109/jiot.2016.2579198 (cit. on p. 17).

[Ste06] T. Stern. *Set Cover Problem (Greedy Approximate Algorithm)*. MIT 18.434 Seminar in Theoretical Computer Science. 2006. URL: http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf (cit. on p. 45).

[TSSP07] C. Tang, M. Steinder, M. Spreitzer, G. Pacifici. "A scalable application placement controller for enterprise data centers". In: *Proceedings of the 16th international conference on World Wide Web - WWW '07*. ACM Press, 2007. DOI: 10.1145/1242572.1242618 (cit. on p. 38).

[URS07] B. Urgaonkar, A. L. Rosenberg, P. Shenoy. "Application Placement on a Cluster of Servers". In: *International Journal of Foundations of Computer Science* 18.05 (Oct. 2007), pp. 1023–1041. DOI: 10.1142/s012905410700511x (cit. on p. 38).

[Wai] T. Waizenegger. *OpenTOSCA documentation. How to build CSARs*. URL: https://www.opentosca.org/documents/howto-build-csars.pdf (cit. on p. 28).

[Wie18] M. Wiedenhöfer. "Cloud Service Monitoring mit TOSCA". MA thesis. IPVS Universität Stuttgart, 2018 (cit. on pp. 36, 37).

[YWJ+15] O. N. C. Yilmaz, Y.-P. E. Wang, N. A. Johansson, N. Brahmi, S. A. Ashraf, J. Sachs. "Analysis of ultra-reliable and low-latency 5G communication for a factory automation use case". In: *2015 IEEE International Conference on Communication Workshop (ICCW)*. IEEE, June 2015. DOI: 10.1109/iccw.2015.7247339 (cit. on p. 17).

[ZSB+08] X. Zhu, C. Santos, D. Beyer, J. Ward, S. Singhal. "Automated application component placement in data centers using mathematical programming". In: *International Journal of Network Management* 18.6 (Nov. 2008), pp. 467–483. DOI: 10.1002/nem.707 (cit. on pp. 38, 39).

All links were last followed on June 10, 2019.

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