Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

# Live-Modeling: Enabling Deployments at Modeling Time in OpenTOSCA

Phi Dang

| | |
|---|---|
| **Course of Study:** | Computer Science |
| **Examiner:** | Prof. Dr. Dr. h.c. Frank Leymann |
| **Supervisor:** | Kálmán Képes, M.Sc. |
| **Commenced:** | November 15, 2019 |
| **Completed:** | June 15, 2020 |

## Abstract

Due to the complex nature and heterogeneity of cloud computing, application development has become a convoluted task that requires detailed knowledge and a great amount of time. Currently, TOSCA, an OASIS standard, attempts to alleviate the involved nature of modeling by providing a meta-model for defining cloud applications. While this standard enables interoperability and portability, modeling these cloud services manually can still hinder many developers. Graphical modeling strives to resolve this by enabling users to visually construct models via an intuitive graphical user interface. Despite these efforts, application development is still impeded because users are obligated to interact with both the modeling and runtime systems in order to test and validate these models. However, the users' workflow would be improved by decreasing the required oversight during development. Therefore, we devised a concept that consolidates the modeling and runtime systems where users have the option to deploy cloud applications during modeling time. Additionally, the concept incorporates a TOSCA-based validation system that verifies the configuration of an application model throughout the modeling process. We prototypically implemented our concept within the OpenTOSCA ecosystem by extending Winery's topology modeler with the introduction of a visual feedback loop concerning the state of the deployment. Our prototype provides an alternative approach to managing TOSCA-based applications that creates a streamlined modeling process and makes developing complex applications more accessible.

# Contents

**1 Introduction** **17**
1.1 Motivation . . . 18
1.2 Structure . . . 20

**2 Foundations** **21**
2.1 Cloud Computing . . . 21
2.2 DevOps . . . 22
2.3 TOSCA . . . 26
2.4 OpenTOSCA . . . 31

**3 Related Work** **39**
3.1 Models@Runtime . . . 39
3.2 Graphical Modeling Editors . . . 41
3.3 Formal Verification of TOSCA Models . . . 45
3.4 TOSCA-specific Modeling Tools . . . 46

**4 Use Case Scenario** **49**
4.1 Workflow . . . 49
4.2 Example Topology . . . 51
4.3 Example Use Case . . . 51
4.4 Requirements . . . 54

**5 Concept** **57**
5.1 Prerequisites . . . 57
5.2 Overview . . . 57
5.3 Live-Modeling . . . 59
5.4 Validation System . . . 67
5.5 Use Case Revisited . . . 73

**6 Implementation** **75**
6.1 OpenTOSCA Ecosystem . . . 75
6.2 Live-Modeling . . . 75
6.3 Input Validation System . . . 84
6.4 Remarks . . . 87

**7 Conclusion and Outlook** **89**

**Bibliography** **91**

# List of Figures


1.1 Modeling Workflows . . . . . . . . . . 19

2.1 Service Template Structure . . . . . . . . . . 27
2.2 TOSCA Conceptual Layers . . . . . . . . . . 28
2.3 OpenTOSCA Ecosystem . . . . . . . . . . 31
2.4 Winery Architecture . . . . . . . . . . 32
2.5 Screenshot TOSCA Management UI . . . . . . . . . . 33
2.6 Screenshot Winery Topology Modeler . . . . . . . . . . 34
2.7 OpenTOSCA Container Architecture . . . . . . . . . . 35
2.8 Screenshot OpenTOSCA UI . . . . . . . . . . 37

3.1 CloudMF Demo Provisioning . . . . . . . . . . 40
3.2 CloudMF Models@Runtime Environment Architecture . . . . . . . . . . 41
3.3 AWS CloudFormation Designer User Interface . . . . . . . . . . 43
3.4 Cloudify Composer User Interface . . . . . . . . . . 44

4.1 Current Workflow - Sequence Diagram . . . . . . . . . . 50
4.2 Example Application Topology . . . . . . . . . . 52
4.3 Iterative Modeling Example . . . . . . . . . . 53

5.1 Alternative Workflow Scenario . . . . . . . . . . 58
5.2 Alternative Workflow - Sequence Diagram . . . . . . . . . . 60
5.3 Reconfiguration Transform Example . . . . . . . . . . 63
5.4 Input Validation System Workflow . . . . . . . . . . 72
5.5 Form Validation Examples . . . . . . . . . . 72

6.1 Screenshot Winery Topology Modeler Extended . . . . . . . . . . 76
6.2 State Machine Live-Modeling . . . . . . . . . . 77
6.3 Upload CSAR File - Sequence Diagram . . . . . . . . . . 79
6.4 Deploy Instance - Sequence Diagram . . . . . . . . . . 80
6.5 Transform Instance - Sequence Diagram . . . . . . . . . . 81
6.6 Adapt Instance - Sequence Diagram . . . . . . . . . . 82
6.7 Winery Topology Modeler - Node Template Visualizations . . . . . . . . . . 83
6.8 Screenshot TOSCA Management UI Properties Definition . . . . . . . . . . 84
6.9 Input Validation System User Interface . . . . . . . . . . 86
6.10 Topology Modeler Misconfiguration Warning . . . . . . . . . . 86

# List of Tables

3.1 Comparison Modeling Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.1 Live-Modeling Aspects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.2 TOSCA Simple Profile Constraint Operators . . . . . . . . . . . . . . . . . . . . . 71

6.1 State Machine Live-Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

# List of Listings


2.1 AWS CloudFormation EC2 Example . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.2 Kubernetes YAML Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.1 TOSCA XML Syntax Node Type . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.2 TOSCA XML Service Template Example . . . . . . . . . . . . . . . . . . . . . . 69
5.3 TOSCA XML Node Type Extension . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.1 TOSCA XML Node Type MyTinyToDoDockerContainer Extension . . . . . . . . 85

# List of Algorithms


5.1 Unequality Check Topology Templates . . . . . . . . . . . . . . . . . . . . . . . . 62
5.2 Subset Computation for Delta Deployment . . . . . . . . . . . . . . . . . . . . . . 64

# List of Abbreviations

**API** Application Programming Interface.

**AWS** Amazon Web Services.

**BPEL** Business Process Execution Language.

**BPMN** Business Process Model and Notation.

**CLI** Command-Line Interface.

**CPU** Central Processing Unit.

**CSAR** Cloud Service Archive.

**DA** Deployment Artifact.

**DSL** Domain-Specific Language.

**EC2** Elastic Compute Cloud.

**HTTP** Hypertext Transfer Protocol.

**IA** Implementation Artifact.

**IaC** Infrastructure as Code.

**IP** Internet Protocol.

**JSON** JavaScript Object Notation.

**NIST** National Institute of Standards and Technology.

**OASIS** Organization for the Advancement of Structured Information Standards.

**OSGi** Open Service Gateway Initiative.

**Regex** Regular Expressions.

**REST** Representational State Transfer.

**TOSCA** Topology and Orchestration Specification for Cloud Applications.

**UI** User Interface.

**URL** Uniform Resource Locator.

**VM** Virtual Machine.

**XML** Extensible Markup Language.

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

# 1 Introduction

Due to the recent growth of data collection and processing workload[1], there is an urgent need for computing resources that can provide enough processing power, storage capacity, and the ability to scale to the actual demands of a system. In the past, it was typical to obtain computing resources via the acquisition of physical hardware. However, it is now commonplace to request these computing resources over the Internet within the cloud. *Cloud computing* enables the provisioning and management of seemingly unlimited computing resources in a self-service manner [MG11]. Most of this provisioning can be manually performed by utilizing the respective service portal, but this quickly becomes infeasible and inefficient when dealing with larger systems. Enterprise software is constantly growing in size and becoming more complex in terms of its internal structure, which lends itself to a more modular-based architecture. For instance, the microservice architectural style promotes the idea of building software that consists of multiple distinguished and interconnected components that are distributed throughout different locations within the Internet [DGL+17]. Despite the fact that cloud computing provides valuable properties, such as scalability and high availability, the additional workload of managing these resources should not be neglected. This includes defining the configurations of the individual components, the way they communicate with each other, and most importantly how to successfully provision them in the cloud. In order to reduce time and effectively save money, there is a set of practices that helps specify well-defined configurations of components and allows to automate necessary management operations throughout the life cycle of an application. This particular set of practices is referred to as *DevOps* [EGHS16].

*Infrastructure as Code* (IaC) is one of the many methods that has proven to be an essential component within the domain of DevOps [ABD+17]. IaC describes the paradigm of defining and modeling an application's structure in a portable and machine-readable format. When talking about an application's structure, we refer to its individual components including their relationships among each other and their respective management operations. The strength of IaC lies in the fact that it is possible to project a conceptual configuration onto a concrete definition that could now act as a single source of truth. This is especially useful since it allows the use of other conventional software methodologies like versioning, reusability, or automation. Particularly, in the domain of cloud computing, vendors would often enforce the use of IaC because of its aforementioned properties and its tendency to avoid manual processes, which would, eventually, result in fewer errors during provisioning [Mor16]. For instance, Amazon AWS Cloudformation[2] and Terraform[3], are among many technologies that utilize this paradigm to automate provisioning and management of applications running in the cloud. Since writing these configurations require expertise in numerous areas, it can be difficult to check for completeness and soundness. Fortunately, there are tools that

---

[1] https://www.gartner.com/en/newsroom/press-releases/2019-02-18-gartner-identifies-top-10-data-and-analytics-technolo

[2] https://aws.amazon.com/cloudformation/

[3] https://www.terraform.io/

offer additional support when modeling an application's structure. Instead of manually writing configurations, users have the option to use visual-based modeling applications that enable the design of these templates using an intuitive user interface. For instance, with the help of AWS Cloudformation Designer[4], users can visually model complete infrastructures or applications that are composed of the various resources residing in Amazon Web Services.

Although tools like AWS CloudFormation Designer can be of great help during modeling time, they only function within the respective vendor's ecosystem. In order to counteract these limitations, the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) has been officially introduced as a standard to model and describe cloud applications under a generic specification language [OAS13]. Besides increasing portability and interoperability, is also facilitates the development of application models within a vendor-neutral ecosystem. However, because TOSCA only describes a standardized format for the exchange of application topologies, it relies on external software ecosystems to elevate its applicability in real-world scenarios.

Eclipse Winery is a web-based management and modeling tool for TOSCA-based cloud applications [KBBL13]. Its topology modeler enables users to visually model the topology of an application via its intuitive and accessible user interface. In order to facilitate a straightforward exchange and distribution of models, Winery also offers the option to package an application as a compressed *Cloud Service Archive* (CSAR) file, that contains the application's topology, management plans and needed artifacts during operation. A TOSCA supported runtime environment, such as OpenTOSCA Container [BBH+13], can then be used to deploy and manage these cloud applications.

The goal of this thesis is to design and prototypically implement a concept that enables users to deploy and retrieve relevant information of application instances during modeling time. Since Winery is already part of the OpenTOSCA ecosystem, we plan on introducing a *live-modeling* feature by integrating a direct communication channel between its topology modeler and the OpenTOSCA Container runtime. We expect to shorten development cycles and offer a viable alternative way of working through the integration of an input validation system and a visual feedback loop between the modeling and deployment of an application.

## 1.1 Motivation

Modeling an application can be performed in various ways, e.g. by manually writing configurations or by utilizing a graphical user interface that facilitates the process through different ways of interaction. Independent of the method used, the users will end up with a model that explicitly states the application's structure and components. These models are commonly saved using standardized data serialization languages such as *JSON*, *XML*, or *YAML* [BBF+18]. These text files can then be provided to a provisioning system in order to actual provision and configure the infrastructure and computing resources based on the content of the given file. In addition to provisioning, this system also enables the management and orchestration of these resources during their lifetime. This includes creating instances, changing configurations, executing management operations, and retrieving status information. While most of these operations are, or can be configured to be executed in an automated manner, some of them still require additional manual input from the users.

[4] https://console.aws.amazon.com/cloudformation/designer

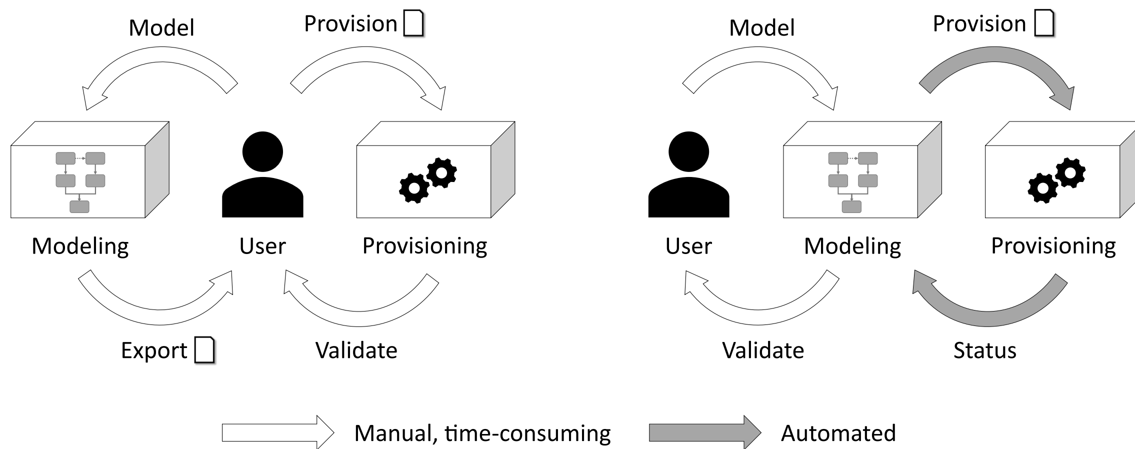



**Figure 1.1:** Illustrations showing the current (left) and an alternative (right) workflow

As depicted on the left side in Figure 1.1, it is important to note that the modeling and provisioning systems operate independently of one another and are considered to be two separate processes. As a result, users often find themselves controlling two feedback loops at once, leading to a more demanding and cumbersome workflow. It becomes especially unwieldy during the early stages of modeling applications where solutions often run through multiple iterations until they are completed and error-free. Therefore, refining these models can become time-consuming due to the nature of manual testing and long feedback loops. This can be especially frustrating for inexperienced users who may encounter more errors during the modeling and deployment of applications due to misconfigurations or bad semantics [XZ15]. An alternative workflow, shown on the right side in Figure 1.1, could address this issue by reducing the users' interaction to only one feedback loop instead of two. In respect to the users' point of view, the users are required to interact with fewer systems which results in less overhead and shorter feedback times. Direct communication between these two systems would also enable new capabilities that could be utilized to further increase support and automation during modeling time.

Until now, we assumed that the responsibility for modeling, as well as provisioning, lies with one single individual. However, practice has shown that this can quickly become infeasible and too slow during development. With the recent emergence of DevOps, development teams tend to distribute their workload among their members in order to maximize efficiency and reduce the required amount of responsibility per individual. In our case and within the scope of this thesis, we presume two roles within a team: *architects*, who are mainly responsible and have deep knowledge in modeling applications, and *users*, who for the most part use and provision these models to meet their own requirements. Combined with the possibility to model entire infrastructures as code, this distribution of work allows for a more versatile way of working due to the clear separation of concerns. For instance, architects can work on testing and refining application structures, and then share these templates as portable configuration files with other parties. Users could then utilize these templates, tweak them to their own needs, and provide them to a provisioning system to automatically provision the defined computing resources. This method of working significantly reduces the workload for users because it eliminates the need to have a profound knowledge of the correct modeling and configuring of applications. Nonetheless, users should always have the chance to turn to the respective architects and seek additional support for cases of improvement or

uncertainty. However, it would be in the interest of all if the level of communication between these two parties can be reduced to a minimum in order to increase efficiency. Thus, it is the architects' responsibility to develop models that are not only error-free but also provide additional guidance to users when utilizing them. Clear documentation or the incorporation of a proper validation system, which in particular supports novice users during the modeling of fault-free models, would be required for an effective workflow. It is noteworthy to mention that this does not necessarily imply that these two roles are always segregated from one another. In contrast, it is just as applicable that single individuals fulfill both roles at once. Independently of the number of people involved, it is beneficial in both scenarios to have a more condensed method that would reduce the amount of overhead from interacting with two systems at once.

The focus on one feedback loop at a time and the introduction of a validation system aim to bridge the gap between modeling an application and its actual deployment. By incorporating these two concepts into the users' workflow, we expect to not only reduce the amount of knowledge overhead but also increase the efficiency of development throughout the modeling and deployment phase.

## 1.2 Structure

The remaining content of this thesis is structured as follows:

**Chapter 2 - Foundations**

Chapter two gives an introduction and overview of various topics that are considered essential for the comprehension of this thesis.

**Chapter 3 - Related Work**

Chapter three presents related work within the field of research and industry and explains their relevance and importance for our work.

**Chapter 4 - Use Case Scenario**

Chapter four provides a concrete use case scenario to showcase the current state of application modeling and approaches on how to improve its usability and efficiency.

**Chapter 5 - Concept**

Chapter five presents a generic overview of the concept our solution is based upon. It also covers how our concept can be used to solve the issue within our use case scenario.

**Chapter 6 - Implementation**

This chapter focuses on the prototypical implementation of the aforementioned concept and gives insight into the technical details of our work. It especially covers the extension and the interconnection between the various components of the OpenTOSCA ecosystem.

**Chapter 7 - Conclusion and Outlook**

The last chapter summarizes the work of this thesis and discusses the current limitations of our prototype while also proposing starting points for future work.

# 2 Foundations

The purpose of this chapter is to introduce various topics and works that provide the fundamental knowledge required for the remainder of this thesis. The first section covers the general concept of cloud computing. The second section incorporates the fundamentals of DevOps, particularly explaining the approach of IaC and the differences between declarative and imperative modeling. Afterward, we introduce the TOSCA standard, a meta-model for defining cloud applications in a portable manner. The last section presents OpenTOSCA, an open-source TOSCA implementation developed at the University of Stuttgart, that provides an end-to-end toolchain to model, provision, and manage TOSCA-based applications.

## 2.1 Cloud Computing

Nowadays, cloud computing is widely adopted and used in the modern industry and considered to be the de facto standard on how computational infrastructure should be based on[1]. In 2011, the *National Institute of Standards and Technology* (NIST) provided an official definition for the term cloud computing which is often referred to as the general baseline on what cloud computing is built upon [MG11]. Based on their tech report, the model of cloud computing represents the idea of on-demand provisioning of computing resources over the Internet, as follows:

Cloud computing is composed of five essential characteristics that describe different abilities and responsibilities between the provider and consumer of these cloud resources. *On-demand self-service* describes the consumer's ability to provision computing resources through the provider's service portal without the need for any manual interaction. *Broad network access* ensures the accessibility to these resources through standardized communication protocols, and thus, allowing access through the use of heterogeneous devices and platforms. *Resource pooling* accounts for the provider's capability to pool together computing resources and distribute them among their consumers in order to match the individual needs and requirements. The term computing resource can thereby include storage, processing, memory, and networking capabilities. *Rapid elasticity* refers to the automated provisioning and release of computing resources based on the actual amount of demanded workload. From the consumer's perspective, the number of computing resources can, therefore, appear to be unlimited. *Measured service* indicates the provider's responsibility to control and measure any type of offered resource. Not only does this help optimize the use of resources, but it also provides both parties with a clear insight into the number of resources that were used throughout their operation.

[1] https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g

As previously stated, there is a wide variety of computing resources that can be provisioned by consumers. Because of this and in combination with the aforementioned characteristics, the NIST tech report also includes three different base service models to further differentiate among various cloud providers: *Infrastructure-as-a-Service*, *Platform-as-a-Service*, and *Software-as-a-Service*. Depending on the type of computing resources each cloud provider offers to its customers, it can be categorized into one or more of these service models. However, since this distinction is negligible for the remainder of this thesis, we will use the term *computing resources* as a general description to cover all of these service models.

## 2.2 DevOps

The introduction of cloud computing caused many businesses in the industry to start adapting their way of working when it comes to the acquisition or offering of computing resources. For instance, it became more common and cost-efficient to forgo the acquisition of physical hardware and instead shift towards the provisioning of virtual computing resources in the cloud, such as virtual machines or containerized applications [Han12]. Although this created many new opportunities throughout development, working teams had to face additional workload that focuses on the operational aspects of provisioning and managing these resources. This led to further complications because companies were traditionally used to segregating their development and operations teams from one another [EGHS16]. Since fast builds and delivery are crucial to successfully compete on the market, there was a need for a set of practices that could help mitigate these issues and introduce new standards within this new field. As a result, *DevOps* has become a common approach within the industry to bring these two branches closer together by aiming for collaboration rather than segregation [EGHS16]. Instead of two independently working entities, DevOps encourages the involvement of the operations team throughout the software development cycle and vice-versa. This encourages the establishment of different concepts within the software development life cycle, such as continuous integration, delivery, and deployment, which inevitably reduces production time and improves reliability. There is no clear definition of what the term DevOps exactly comprises [SMB17][EAD17]. In contrast, teams and organizations often come up with their own definitions. Smeds et al. [SNP15] define DevOps as the collective set of different engineering process capabilities that get supported by various cultural and technological enablers. The term *technological enablers* thereby refers to different automation aspects throughout the software life cycle, such as automated building, testing, or monitoring. Such automation is made possible by various tooling solutions that established themselves throughout the last few years. They enable the automation of various processes that were historically performed manually. This includes software testing, provisioning of computational resources, or scaling of application instances during operation. Not only does this increase the overall efficiency, but more importantly implements repeatability and maintainability while reducing the likelihood of human errors.

### 2.2.1 Infrastructure as Code

IaC, or also referred to as *software-defined infrastructure*, describes the approach of defining infrastructure through written code [Mor16]. In the past, system administrators had to manually provision computing resources through the series execution of static operations. Now, IaC makes it

possible to manage a system's provisioning and operation in an automated manner. This automation eliminates slow and time-consuming tasks like provisioning server instances, installing operating systems, or the establishment of connections between different components. Especially in the domain of agile software development where software is deployed multiple times each day, IaC becomes essential in order to optimize cost and productivity. An early iteration of this concept consisted of installation and configuration scripts that automated the execution of various smaller tasks. However, these scripts tend to become hard to maintain over time and lack the conceptual information of how a system was structured and how to properly manage it after its provisioning. IaC strives for maintainability and provides a single source of truth by encouraging the use of written definition files that fully represent a system's structure and components. These definition files are written in a machine-readable language, allowing humans as well as machines to properly interpret them. Apart from providing a single source of truth, IaC also offers the following benefits:

- **Versioning**
  Since infrastructure is treated as generic source code, it can be checked into any version control system, and therefore, enables tracking of any configuration changes over time. This also allows for collaboration among different developers, in the same manner as modern software development is organized.
- **Rapid Execution**
  IaC allows for a quick and reliable way to provision entire infrastructures by eliminating manual processes and the need for human input.
- **Repeatability and Consistency**
  Manual processes that require human input are notoriously more prone to include errors or fail entirely. IaC standardizes these procedures and mitigates errors by ensuring that configurations remain consistent throughout multiple executions.
- **Documentation**
  Defining and managing infrastructure can be a challenging task and often require expertise in various areas. As a result, teams are constantly confronted with the fear of potentially losing valuable know-how whenever qualified members decide to leave. However, since definition files are by default required to be self-contained, IaC ensures that knowledge remains preserved even in the absence of those experts.

Provisioning systems, such as AWS CloudFormation or Terraform, are capable of automatically provision and configure infrastructure in the cloud with regard to the given description of a system. Listing 2.1 shows the minimum content of a template file for the provisioning of an *Elastic Compute Cloud* (EC2) instance. It is written as a YAML file since it is one of the text formats supported within AWS CloudFormation. The template consists of an EC2 instance and a corresponding security group that ensures SSH access via port 22. Each system resource can be further described through the declaration of properties within its respective definition block. In our case, the EC2 instance is based upon the *ami-6ec3b91f* machine image and is from type *t2.micro*. Furthermore, we specify a reference to our defined security group in order to let the provisioning system know that we require external SSH access. Since security groups are also considered to be resources that are provisioned, we also have to properly define them within our template. We can then utilize the AWS CloudFormation CLI to actually provision our EC2 instance after providing it with our credentials. Our instance should be automatically provisioned within a matter of minutes, and we can retrieve its current status by opening the control panel within our Internet browser.

**Listing 2.1** AWS CloudFormation YAML template for an EC2 instance

```
Description: Sample Template for EC2 instance
Resources:
  SimpleEC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-6ec3b91f
      InstanceType: t2.micro
      SecurityGroups:
      - !Ref SSHSecurityGroup

  SSHSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Enable SSH access via port 22
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
```

As mentioned previously, IaC is not only limited to the provisioning of computing resources but can be also used to orchestrate resources during their operation. To demonstrate this, we shortly touch upon an example that takes advantage of this concept. Kubernetes[2], for instance, is a platform for deploying and orchestrating containerized applications that allows the definition of template files to determine when certain management operations should be executed based on a given ruleset. Listing 2.2 shows a Kubernetes template for a horizontal pod auto scaler that scales the number of replicas of a containerized application based on a self-defined metric. In this example, the system tries to maintain between 1 and 10 replicas in order to consistently match a target average CPU utilization of 50%. If the average CPU utilization falls under this threshold, it will decrease the number of replicas and vice-versa if it exceeds the threshold.

### 2.2.2 Declarative and Imperative Modeling

Listing 2.1 and Listing 2.2 share the same concept of describing a target state that needs to be reached rather than explicitly stating instructions that need to be performed during the process. Therefore, it is the provisioning system's responsibility and freedom to decide what exactly needs to be executed in order to match the users' defined template. This makes it especially easy for inexperienced users to model and provision resources because it lowers the required amount of technical expertise by subtracting complex implementation details that can otherwise quickly become overwhelming. This concept of only defining *what* has to be achieved is referred to as a declarative approach. In contrast, an imperative approach allows users to manually define each step throughout the process, resolving the question of *how* a certain goal is being achieved [BKLW17]. This enables full customization of the process model, including the implementation details and the execution order of each provisioning step. For instance, API invocations or the execution of shell scripts are common approaches to

[2] https://kubernetes.io/

**Listing 2.2** Kubernetes YAML template for horizontal pod auto scaler based on CPU utilization

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
name: container-app
namespace: default
spec:
scaleTargetRef:
 apiVersion: apps/v1
 kind: Deployment
 name: container-app
minReplicas: 1
maxReplicas: 10
metrics:
- type: Resource
 resource:
   name: cpu
   target:
     type: Utilization
     averageUtilization: 50
```

provision resources or install required software. The order in which these steps are executed can then be modeled by using standardized workflow technologies [LR99] that provide languages such as BPEL [OAS07] or BPMN [OMG11]. An imperative approach of modeling can be particularly useful in cases that deal with systems that require custom provisioning logic. An example of such an imperative provisioning system is Chef[3]. Users can hereby define their own commands and execution order within so-called *Recipes*.

There are many provisioning systems available that support the declarative and/or the imperative approach. Because each platform commonly operates under its own domain-specific language (DSL) or data model, users would often find themselves in a vendor lock-in situation that consequently bound their own written templates to the one chosen ecosystem. In other cases, some platforms only support a specific subset of cloud providers, making it very difficult to provision cloud applications that need to be run in a multi-cloud environment.

### 2.2.3 Remarks

Before advancing any further, we first want to clarify the ambiguity that exists when using the terms *provisioning* and *deploying*. Depending on the context and situation, they have different meanings and cover other aspects. The term *provisioning* is usually used when setting up infrastructure or configuring hardware resources in order to make them available to customers or systems. For instance, customers of AWS have the ability to request the *provisioning* of various AWS resources, e.g. dedicated machines, through the offered platform. The term *deployment*, on the other hand, is often used in conjunction with installing or updating a new piece of software within a given environment. In fact, Leymann [Ley09] states in his paper that the term *provisioning* is used to refer to a joint series of actions consisting of installing, *deploying*, and configuring software in an

[3] https://www.chef.io/

allocated hardware environment. As a result, while *provisioning* can include the deployment of certain components, the term *deploying* does not consequently imply *provisioning*. However, since our application models hold hardware and software components alongside and do not specifically differentiate among them, we consider these two terms to be interchangeable and consider them as synonyms for the remainder of this thesis. Despite the fact that our application models can contain provisioning logic within their definition, we will use the term *deploying* exclusively for the remainder of this thesis. This decision is made to be in accordance with the specification our work is based upon: the TOSCA standard.

## 2.3 TOSCA

In January 2014, TOSCA was formally approved by the *Organization for the Advancement of Structured Information Standards* (OASIS) as an official standardized language. It provides a meta-model for describing the deployment and management of cloud applications in a vendor-neutral ecosystem. As of right now, there exist two versions of TOSCA that are semantically equivalent but differ in the selected serialization format used within the respective specification. The initial draft that was initially accepted and referred to as TOSCA Version 1.0, proposes the use of the official XML Schema 1.0 to describe the structure of its applications [OAS13]. In 2017, the TOSCA Simple Profile was introduced as an alternative, which proposes the use of a YAML syntax in order to simplify and promote a more human-readable format [RBL17]. For the remainder of this thesis, we introduce and focus on the XML-based version of TOSCA. Independent of the version, TOSCA enables two important attributes during the development of cloud applications: *portability* and *interoperability*. While portability enables ease of exchange of application models, interoperability pushes the ability to manage and communicate among multiple components even if they are deployed on completely separate cloud providers. Both aspects prevent the risk of running into vendor lock-ins that would otherwise likely occur when choosing a commercial platform as mentioned in Section 2.2.2. Since our concept relies on TOSCA, this section focuses on its specification and, moreover, introduces relevant work for this thesis.

### 2.3.1 Structure

The TOSCA specification introduces the concept of *Service Templates* for modeling the topological structure and management operations of a cloud application. Hereby, the term cloud application refers to any arbitrary computing resource as described in Section 2.2.1. Figure 2.1 depicts all structural elements a Service Template can be composed of:

- **Topology Template**
  Directed, acyclic graph, that defines the overall structure of an application, where vertices correspond to Node Templates and edges correspond to Relationship Templates. Each Node and Relationship Templates' semantics are thereby defined by their respective type component.
- **Node Types**
  Collection of Node Types that are referenced among all included Node Templates. A Node Type dictates the properties and interfaces a Node Template offers when deriving from it.

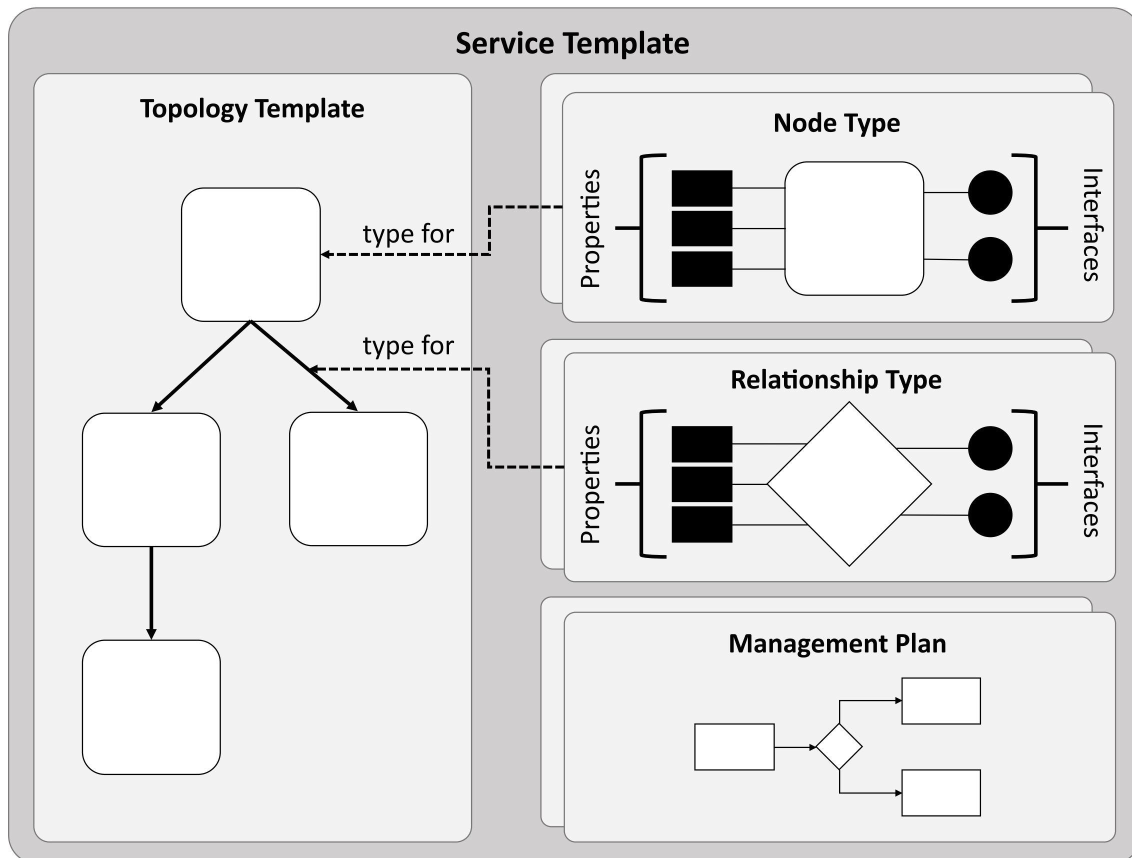



**Figure 2.1:** Structural elements of a Service Template [OAS13]

- **Relationship Types**
  Collection of Relationship Types that are referenced among all included Relationship Templates. A Relationship Type dictates the properties and interfaces a Relationship Template offers when deriving from it.

- **Plans**
  A collection of process models that formally define all tasks that need to be performed during the various life cycle stages of an application, such as during creation or termination. TOSCA does not enforce the use of any specific language but encourages the use of standardized workflow languages, such as BPEL [OAS07] or BPMN [OMG11]. Each task has access to all operations that are defined within Node and Relationship Types and, therefore, can call them on the respective Node or Relationship Template.

An important aspect of TOSCA is the concept of abstraction. Figure 2.2 shows the conceptual layers and the order of the definition between the elements [BBKL13]. The TOSCA meta-model introduces a two-layer type system that can be used to define templates in order to increase reusability and the level of abstraction, e.g. between Node Types and Node Templates. The third layer is implemented by a TOSCA supported runtime that is responsible for instantiating the given templates during operation, i.e. deriving Node Instances from Node Templates. Node Types define properties and operations via interfaces for their respective Node Templates. For instance, when defining a Node

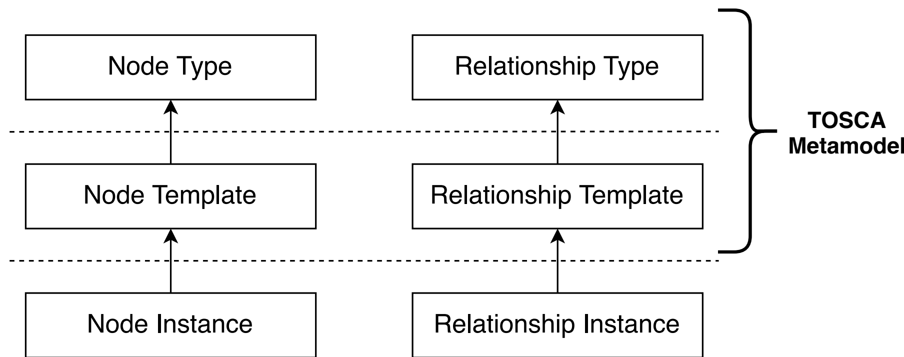



**Figure 2.2:** Conceptual layers of TOSCA elements [BEK+16]

Type *operating system*, we can add the properties *username* and *password* to its definition. When modeling a topology that contains this Node Type, we instantiate a Node Template from this type and provide concrete values for these properties. After the deployment of our service, we can then log in to the respective operating system instance using these defined credentials. Correspondingly, the same concept applies to relationship entities. Similar to Node Types, there are different Relationship Types that differ in their semantics. For instance, the Relationship Type *hostedOn* can be used to indicate that component A is hosted on or needs to be installed in the environment of component B. This type system supports the concept of inheritance and extensibility which make it easy to define custom elements outside the TOSCA specification.

While types enable the formal definition of components and their respective management operations, they do not include the actual implementation behind them. TOSCA relies on *artifacts* to include all necessary files and contents that are needed during the deployment and operation of the respective Node Type. As a result, TOSCA differentiates between two types of artifacts: *Deployment Artifacts* and *Implementation Artifacts*. As the name suggests, Deployment Artifacts incorporate all resources that are required to instantiate an instance of a respective Node Template. Implementation Artifacts, on the other hand, incorporate executables that describe the actual implementation behind an operation of a Node Type. Independent of the artifact type, artifacts themselves can be implemented in an arbitrary format, such as shell scripts, binary executables, images, or web services. Since a runtime is used to deploy and manage instances, it is essential that the runtime is compatible and capable of executing these artifacts.

In order to ensure the portability of cloud applications, TOSCA defines its own archive format called CSAR. When packaging a cloud application, its respective CSAR file is composed of the Service Template as well as the Deployment and Implementation Artifacts that were referred to within all used type definitions. Additionally, each CSAR file must contain a TOSCA meta file that provides additional information about its version, authors, and MIME-types of all contained artifacts. Not only does this format enable the mutual exchange of cloud applications over the Internet, but it also offers a standardized way of packaging all necessary files into one single self-contained file.

### 2.3.2 Runtime

TOSCA enables the modeling of a cloud application's structure and management plans in a standardized exchange format. It does not, however, impose any restrictions or specifications on how to deploy and orchestrate these services when given their respective CSAR file. A TOSCA supporting runtime elevates the capabilities of this concept by automating the deployment and management of applications [BBKL13]. Given an arbitrary CSAR file, the runtime is capable of interpreting its content and deploying the modeled application in its respective target environment. As mentioned in Section 2.2.2, there are various systems that support different kinds of approaches during the deployment of resources. While some systems only support one kind of deployment approach, TOSCA enables runtimes to be capable of deploying instances in a declarative as well as an imperative manner. We already touched upon the differences between these two approaches in Section 2.2.2. Based on this, an imperative approach is best fitted for cloud applications that require custom logic during the deployment of its instance. An imperative runtime supports the execution of management plans that are based on standardized workflow languages. Therefore, the runtime contains a workflow engine that manages the state of deployment processes during operation. When deploying an instance of a given Service Template, the runtime would execute a special management plan, called *build plan*, that is contained in the CSAR of the respective application. This build plan contains information about the order that the Node Templates need to be instantiated, and also provides actual values for all properties that are defined within the various Node and Relationship Templates. An imperative runtime requires the inclusion of such a build plan within the Service Template. The inclusion of a build plan within the Service Template ensures the portability and consistency of deployment processes among various runtimes. In contrast, declarative runtimes are able to dynamically determine all necessary actions to deploy applications, and thus, allow the omission of build plans within Service Templates. For this, TOSCA defines a non-normative *life cycle interface* that Node Templates are required to implement, in order to ensure the availability of the following management operations: *install*, *start*, *configure*, *stop* and *terminate* [LRH+13]. When instantiating a Service Template, the runtime would automatically derive the deployment order of Node Templates based on their topological order and the type of relationships they are connected with. For instance, a Node Template that is connected with another Node Template through a hostedOn relationship needs to be logically instantiated before the other. After a sequence is determined, the runtime would then sequentially execute the install, start and configure operation on each occurring Node Template. Similar to every other operation, life cycle operations are also expected to be defined through a corresponding implementation artifact. Therefore, it is still required to provide the corresponding implementation when utilizing a declarative runtime.

### 2.3.3 Roles

In addition to the official specification of TOSCA, OASIS also published a primer document that serves as a guide on the practical use and concepts of the TOSCA meta-model [LRH+13]. One aspect of this document focuses on distinguishing three different service roles when working with TOSCA: *Cloud Service Consumer*, *Cloud Service Developer*, and *Cloud Service Provider*. Consumers primarily interact with TOSCA-based cloud services that are provided to them through the offerings of cloud service providers; they generally do not deal or work with the specification. Developers, on the other hand, use the TOSCA standard to develop and define their cloud services in a standardized way that can be provided to consumers later on. Cloud service providers act as

middlemen between the other two parties by operating a TOSCA supported environment to offer and manage these developed cloud services for their consumers. Since consumers and providers are not involved in the modeling of these cloud services, we primarily focus on the role of cloud service developers. The primer further differentiates this role into three technical specializations: *Type architect*, *Artifact developer*, and *Application architect*. Although each specialization has its own number of concerted actions, it is possible that a single developer fulfills multiple specializations.

Type architects are experts in defining the necessary Node and Relationship Types for various service components. Since types will be used later to derive template and instance counterparts, type architects are required to have profound knowledge about the technical details of the components they are working with. This includes the definition of properties and management operations that are needed during the whole life cycle of the component. Moreover, type architects are usually aware of the valid values that properties can be assigned. A trivial example of this would be the declaration of a generic port property on a component. Since port numbers are technically limited to the range between 0 to 65535, it is the type architects' responsibility to provide a type definition that not only functions but is also well-documented.

As mentioned in Section 2.3.1, types only define operations within a component and do not contain their actual implementation. Consequently, artifact developers deal with the development and management of all Deployment and Implementation Artifacts that are needed during operation. If a Node Type implements the life cycle interface, for instance, it is expected that artifact developers provide a semantically correct implementation for each of the five operations.

Application architects focus on the modeling of Service Templates. They combine the works of the two previous specializations by composing Node and Relationship Templates together that will eventually form one cloud application. Apart from defining an application's structure, they also have the responsibility of providing suitable values for each defined property and define management plans that utilize the underlying operations defined by the type architects. Since application architects rely on a correct definition and implementation of Node and Relationship Types, it is common for them to seek additional help from their colleagues whenever they encounter errors or misconfigurations during operation. However, this can lead to communication overhead and result in inefficiency on both sides. Therefore, it is ideal to prevent these issues early on.

### 2.3.4 Vino4TOSCA

Since TOSCA only provides a textual syntax for defining the topology of services, it can quickly occur that users struggle with identifying a clear structure when looking at a template definition. This is especially true when dealing with bigger topologies that consist of numerous Node and Relationship Templates. As a result, Breitenbücher et al. [BBK+12] proposed a visual notation for TOSCA-based application topologies. Their model provides visual representations for each essential element that is used throughout the modeling of a Topology Template. Not only does this allow for a better and faster way to grasp an overview of the overall topology, but it also increases the applicability of UI-based tools that can help users during the modeling of cloud applications.

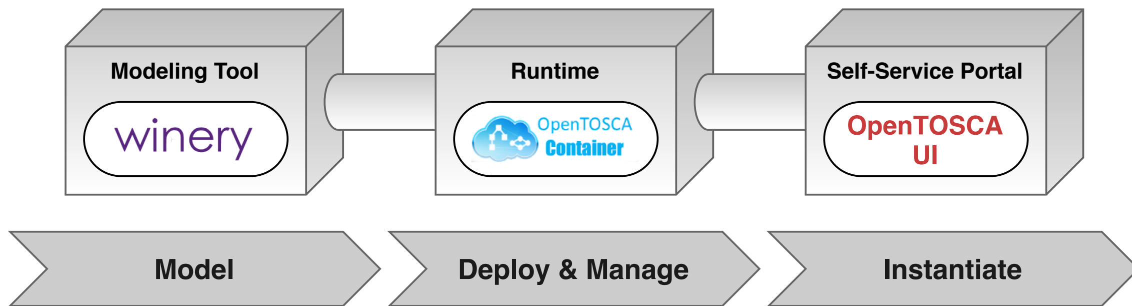



**Figure 2.3:** OpenTOSCA ecosystem [BEK+16]

## 2.4 OpenTOSCA

TOSCA enables the portability of cloud applications through a standardized meta-model and package format but does not offer a concrete implementation. Therefore, TOSCA relies on external tooling in order to fully take advantage of its applicability. The OpenTOSCA ecosystem is an open-source implementation of the TOSCA standard and is currently in development at the University of Stuttgart. It describes an end-to-end toolchain that enables modeling and automating the deployment and management of TOSCA-based cloud applications. Figure 2.3 shows the various components that reside within the OpenTOSCA ecosystem [BEK+16].

Winery is a graphical web-based editor that is mainly used to manage TOSCA entities and visually model the Topology Template of cloud applications [KBBL13]. In order to allow the exchange of cloud applications, Winery further supports the packaging of Service Templates as CSARs. The OpenTOSCA Container is a TOSCA supported runtime that facilitates a straightforward approach to deploy and manage applications [BBH+13]. OpenTOSCA UI, formerly known as Vinothek [BBKL14], allows end-users to easily deploy and manage applications through the use of its graphical user interface. The following subsections will introduce each component in detail and explain their relevance to the remainder of this thesis.

### 2.4.1 Winery

Winery is a graphical modeling and management tool used to model TOSCA-based cloud applications [KBBL13]. Since it is primarily web-based, there is no additional software needed because users are able to access all components through the use of generic web browsers such as Google Chrome[4] or Mozilla Firefox[5]. As previously mentioned, Winery covers two important tasks throughout the process of modeling. On one hand, it offers a user interface for defining reusable TOSCA entities such as Node and Relationship Types. On the other hand, it includes a graphical editor to model Topology Templates by using these predefined entities. Therefore, Winery differentiates between two sets of TOSCA entities: entities that are needed for semantics and configuration (e.g., Node Types, Relationship Types, or Deployment and Implementation Artifacts) and entities that are primarily used throughout modeling (e.g., Node or Relationship Templates). This separation

[4] https://www.google.com/chrome/
[5] https://www.mozilla.org/en-US/firefox/

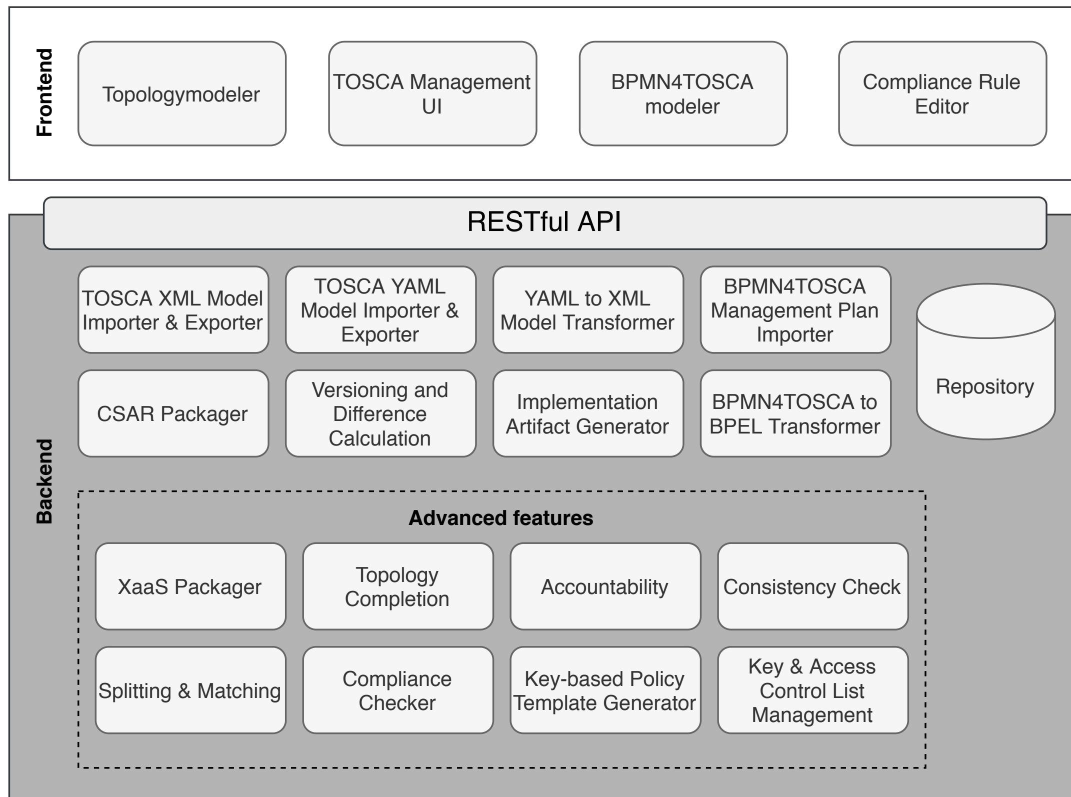



**Figure 2.4:** Architectural overview of Winery [WINE20]

of entities fosters an environment of sharing and collaboration among users. For instance, as mentioned in Section 2.3.3, type architects would define and configure these entity types in order to make them available to others. Application architects could then utilize these definitions to visually model topologies without worrying about the technical details behind them. Figure 2.4 shows the internal structure and the core components of Winery: TOSCA Management UI, Topology Modeler, BPMN4TOSCA Modeler, and its internal repository.

The TOSCA Management UI enables the creation, configuration, and deletion of different TOSCA entities, such as types, plans, and artifacts. Figure 2.5 shows the user interface of the TOSCA Management UI during the definition of properties of a Node Type. Users can upload concrete artifact implementations and associate them with the respective artifact type. Since Java web services are directly supported within the OpenTOSCA container, the management UI offers the option to generate a Java-based web service template to provide the users with a starting point when implementing a management operation. In addition, it features the possibility to generate the aforementioned life cycle interface (see Section 2.3.2) and attach it to a given Node Type. Winery further allows the export of Service Templates as CSAR files using either the XML or YAML notation. Furthermore, users can enrich each type with optional data, such as icons, descriptions, or author information, that are accessed within the topology modeler or the OpenTOSCA container.

The topology modeler is a graphical editor that enables users to visually model Topology Templates of cloud applications. Figure 2.6 shows the user interface of the topology modeler during editing. In order to generate a graphical representation of Topology Templates, the topology modeler

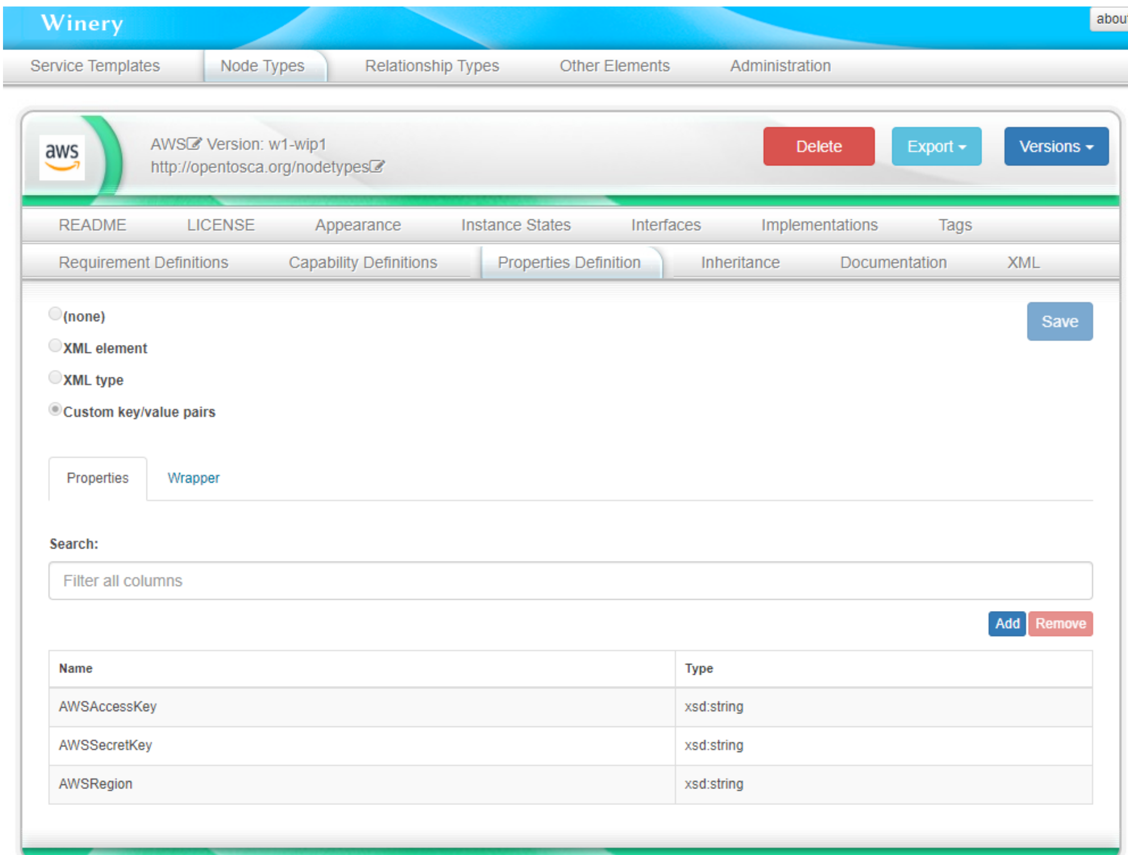



**Figure 2.5:** Screenshot of the TOSCA Management UI displaying the properties of a Node Type

utilizes the Vino4TOSCA notation of Breitenbücher et al. [BBK+12]. As a result, Node Templates are represented by rounded rectangles, while Relationship Templates are visualized via arrow connections. Figure 2.6 also depicts a palette on the left side of the editor consisting of all defined Node Types that are stored within Winery's repository. Clicking on one Node Type and dragging it onto the editor plane will instantiate the respective Node Template. The icon and border color of the rectangle is derived from the information that was previously defined within the TOSCA Management UI. In order to define a Relationship Template between two Node Templates, the users have to click on the source Node Template, select the respective Relationship Type, and drag the mouse onto the target Note Template. Node Templates can be arbitrarily placed and aligned within the editor plane. Other characteristics, such as properties, requirements, capabilities, or Deployment Artifacts, can be edited when toggling the respective button on the navigation bar. The navigation bar also includes additional features that can help during the modeling of topologies, such as auto-completion, problem detection, or split-and-match.

The BPMN4TOSCA modeler enables the modeling of imperative management plans based on a BPMN-based notation extension, called BPMN4TOSCA [KBBL12]. It incorporates TOSCA domain-specific elements, such as handling of type properties or execution of management operations. Similar to the topology modeler, it also includes a graphical-based editor to simplify the modeling

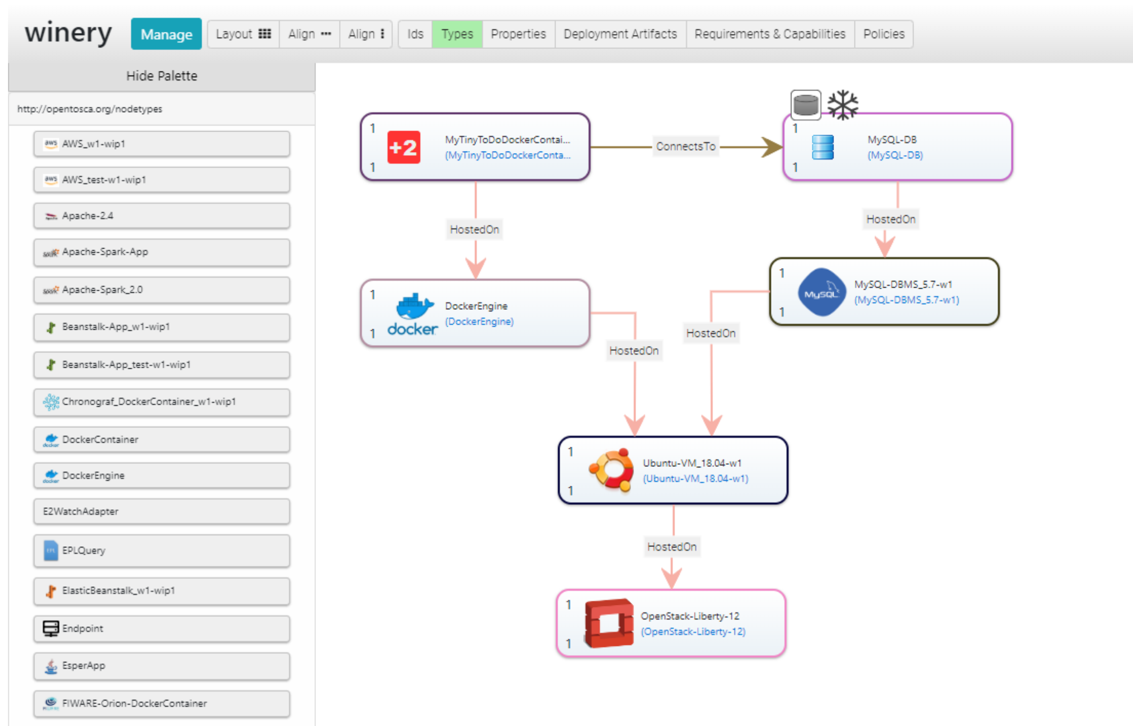



**Figure 2.6:** Screenshot of the topology modeler displaying an example application

process. Since BPEL is the chosen language for the automation of workflows within the OpenTOSCA Container, Winery also includes a BPMN4TOSCA to BPEL Transformer for converting these workflows into a more widely used format.

The Winery repository stores all relevant data and information, such as templates, types, or artifacts. It is entirely file system based, and thus, allows for manual edits and the ability to use a version control system for versioning. All entities are accessible through a RESTful interface [RR07].

As described in Section 1.1, users are currently obligated to interact with two systems at once when modeling and deploying applications. Winery and its accompanying topology modeler correspond to the modeling system within our motivational scenario. Because of this, the work within this thesis focuses on extending Winery by implementing a live-modeling feature that enables users to directly deploy instances from within its topology modeler.

### 2.4.2 OpenTOSCA Container

The OpenTOSCA container is a TOSCA supported runtime, built on top of the OSGi framework[6], that enables the automated deployment and management of TOSCA-based applications [BBH+13]. Similar to the Winery repository, all the container's functionalities are accessible through a RESTful interface [RR07]. The instantiation of applications is done through the execution of an imperative build plan that is conventionally included within the contents of a CSAR. If a build plan is missing,

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

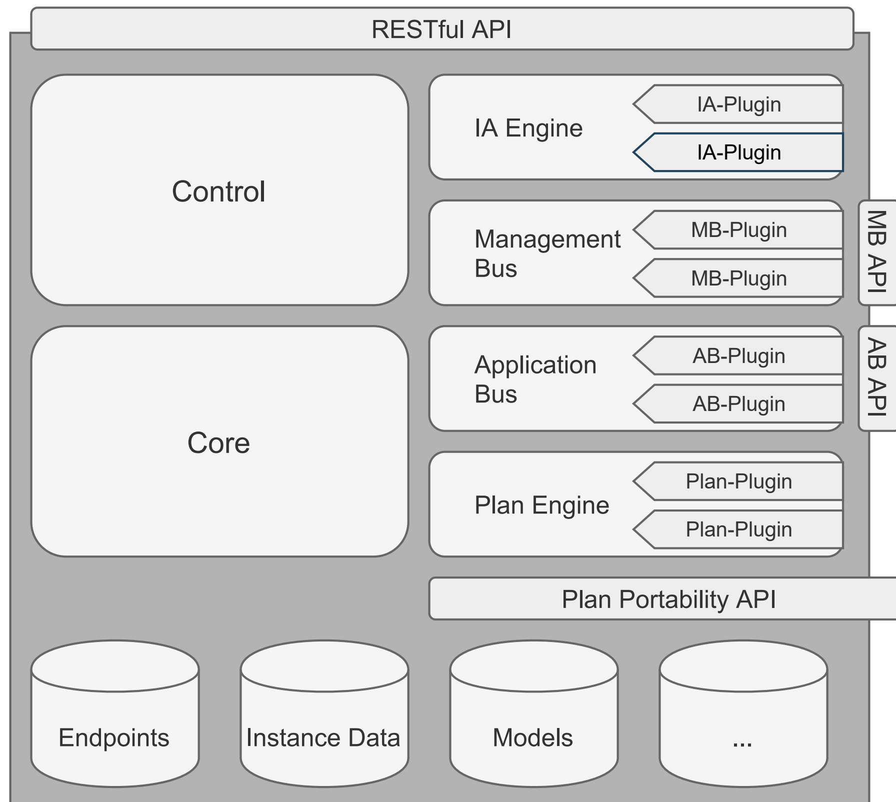



**Figure 2.7:** Architectural overview of the OpenTOSCA container [BBH+13]

the container can automatically generate one based on the application's topology model and semantics of the Relationship Templates among the Node Templates. This principle represents a hybrid approach between the declarative and imperative modeling approach, and thus, combines the strengths of both methods [BBK+14]. Apart from enabling deployments, the OpenTOSCA container is also able to invoke declared management plans during the life cycle of application instances. Figure 2.7 shows an architectural overview of the OpenTOSCA container. It comprises the following main components: Implementation Artifact Engine (IA Engine), Plan Engine, Management Bus, Application Bus, and Plan Builder.

As mentioned in Section 2.3, Implementation Artifacts represent the actual business logic of management operations. Since the container allows the execution of management plans, which can consequently call management operations, the IA Engine provides a way to manage and process Implementation Artifacts. The container differentiates between two types of Implementation Artifacts: local and remote. *Local Implementation Artifacts* are those that are deployed on a local Apache Tomcat Servlet Container, and whose endpoints are stored within the Endpoints database so they can be invoked by plans later on. On the other hand, *Remote Implementation Artifacts* encompass all artifacts that need to be invoked in the application's target environment, which implies that they cannot be locally deployed within the container's environment. For instance, this can include installation shell scripts that need to be executed within a virtual machine's environment. They are stored within the Models database and can also be accessed by plans when needed. In

order to support different types of Implementation Artifacts, the OpenTOSCA container has a plug-in system that provides a suitable execution plugin for each artifact type. This plug-in system is easily extensible to facilitate the execution of arbitrary artifact types.

The Plan Engine is responsible for the processing of management plans. Therefore, the container employs a local workflow engine that supports the deployment and execution of BPEL-based plans. Similar to the IA Engine, the Plan Engine also incorporates a plug-in system that can be further extended to support different kinds of workflow languages. Throughout execution, management plans have the capability to access the application's topology and instance information through the Plan Portability API [BEK+16].

Since Implementation Artifacts can be of arbitrary format, the coordination of operation invocations can become complex and cumbersome to manage. The Management Bus is a communication middleware that enables the invocations of management operations through a unified interface [WBB+14b] [WBB+14a]. By utilizing a unified interface, workflow engines are able to invoke these operations without worrying about the technical details on how to properly execute them.

It is the Plan Builder's responsibility to generate imperative build plans whenever the container receives CSARs that do not contain a respective build plan [BBK+14]. Therefore, it analyzes the application's Topology Template to dynamically generate a BPEL-based workflow model. In detail, it first generates an abstract control flow, then transforms it into a BPEL-based skeleton, and eventually completes it by filling in the skeleton's placeholder tasks with actual low-level code that will implement the actual logic to install and configure the respective Node Templates. For instance, the last step can be derived by implementing the life cycle interface within Node Types. Apart from generating build plans, the Plan Builder is also capable of generating other types of plans, such as termination plans or scale-out plans [OPTO20].

Referring back to Section 1.1, the OpenTOSCA Container corresponds to the provisioning system within our scenario. It enables the automated deployment and management of applications when provided with a TOSCA-conform CSAR file. We plan on utilizing its RESTful interface in order to capitalize on the container's features within Winery's topology modeler. This reduces the number of experienced feedback loops and enables users to simultaneously model and deploy applications during modeling time.

### 2.4.3 OpenTOSCA UI

The OpenTOSCA UI, formerly known as Vinothek [BBKL14], enables end-users to easily deploy and manage applications through the use of a graphical user interface. Figure 2.8 depicts a screenshot of the graphical user interface of the OpenTOSCA UI. It displays the information of a cloud application that has been uploaded to the OpenTOSCA container. By utilizing its user interface, end-users are able to deploy, terminate, or execute management plans of application instances. The OpenTOSCA UI will automatically ask the users to fill in any properties that are required during the deployment of an application. Moreover, the user interface lists any application instance that is currently being run and managed by the OpenTOSCA Container. In order to add new CSARs, users can either manually upload CSAR files or configure an endpoint that points to an external repository, e.g. the Winery repository.

Figure 2.8: Screenshot of the OpenTOSCA UI

The OpenTOSCA UI currently remains the conventional way to manually trigger the deployment of applications and invoke the executions of management plans. Therefore, users have to continuously switch between two systems whenever they model an application within Winery and decide to deploy it. Especially during the early test stages where application modeling often consists of multiple iteration steps, this can become cumbersome and time-consuming. Thus, our implementation represents an alternative to the current status quo and enables users to work more efficiently.

# 3 Related Work

This chapter presents the status of previous research and works that is related to our work, such as modeling cloud deployments, verification of TOSCA models, and existing modeling tools when working with TOSCA-based cloud applications. Each section shortly summarizes the works of different authors and then offers an analysis of their relevance to our work.

## 3.1 Models@Runtime

In 2014, Ferry et al. [FSR+14] developed a cloud provider-agnostic solution, called *Cloud Modelling Framework*, that deals with the management of multi-cloud applications both at design and runtime. Their model-driven framework includes a tool-supported DSL, called *Cloud Modelling Language* (CloudML), and a model@runtime environment that supports the provisioning and adaptation of cloud applications. CloudML enables developers to describe multi-cloud applications at two different levels of abstraction. *Cloud Provider-Independent Models* (CPIM) are cloud provider-agnostic models that contain information about the cloud application's structure and provisioning. In contrast, *Cloud Provider-Specific Models* (CPSM) refine these CPIM by specifying the use of particular cloud providers within their models. This includes the disclosure of any required information, such as virtual machine type, RAM size, or storage, and any cloud-provider specific data that is essential during the provisioning and management process. Such refinements usually require a certain amount of expertise concerning the technical details of every used cloud provider. Due to this, the models@runtime environment is able to support the developer in such scenarios by requesting the needed meta-data for each respective context. For this purpose, the models@runtime environment contains up-to-date operations for each supported cloud provider to automatically retrieve and refine the CPIM with the respective meta-data. CloudML's level of abstraction promotes independence by putting a stronger emphasis on generic operation logic instead of focusing on the implementation details of specific cloud providers. Thus, the possibility of running into vendor lock-ins is avoided. To further increase portability, the models are defined as plain java models that can be easily serialized into more generic textual formats such as JSON or XML. Apart from the textual syntax, their models can also be graphically edited through a graphical modeling editor. During provisioning, their framework offers a visual representation of the modeled application and includes up-to-date instance information. Figure 3.1 shows the user interface during the provisioning of a sample application. While circles are used to visualize single components within the application, arrows are used to represent relationships and dependencies among them. The provisioning status of each component is dictated by its color. Green indicates a provisioned and running instance, yellow indicates that the provisioning is still ongoing, while gray indicates a pending process that waits for other components to finish first.

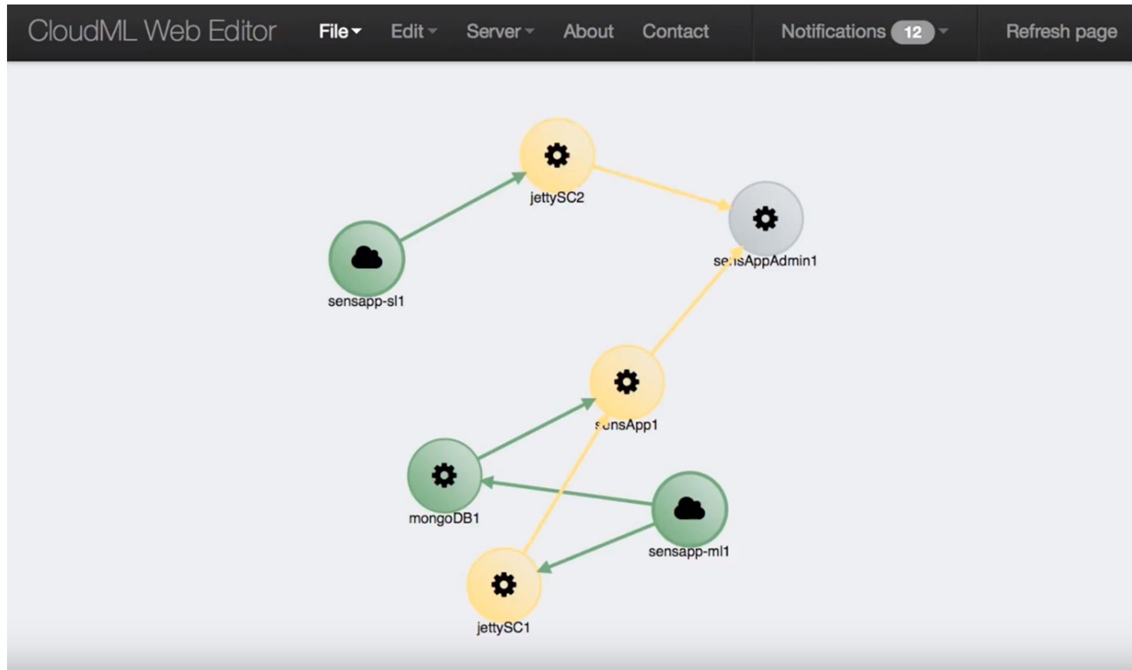



**Figure 3.1:** User interface of CloudMF Web editor during provisioning [FSR+14]

Models@runtime describes an architectural pattern that extends the applicability of models by treating them as abstract representations of a running system [BBF09] [MBJ+09]. In particular, this means that any changes made to the running system will be reflected within the model, and vice-versa, any changes to the model will enact modifications to the running system. This causal connection between the model and the running system establishes a link between the design and runtime of applications, allowing the synchronization and immediate reflection of any changes. Figure 3.2 shows the architecture of the CloudMF models@runtime environment. Hereby, the current CPSM describes the running system. After providing it to an external reasoning engine, the models@runtime environment receives a new target CPSM and computes the differences between these two models. Based on this *Diff*, the models@runtime environment derives all necessary adaptation steps in order to match the current CPSM with the target CPSM. An adaptation engine then executes these steps, such as provisioning a new virtual machine, changing a property of a component, or removing an internal component instance from a host. After executing all adaptation steps, the target CPSM becomes the current CPSM. This synchronization loop ensures at all times that the model remains consistent with the current status of the running system. Lushpenko et al. [LFS+15] further expanded the work of Ferry et al. by enabling the manual manipulation and analysis of adaptation plans before their execution. Their work includes the definition of a DSL for adaptation plans and extended upon CloudMF's models@runtime environment by enriching it with an interactable adaptation system.

The work of Ferry et al. describes a similar approach when compared to ours. For instance, we also plan to introduce a feedback-based loop within our modeling system that enables users to gain insight into the current state of their deployed application. The immediate reflection of changes and the

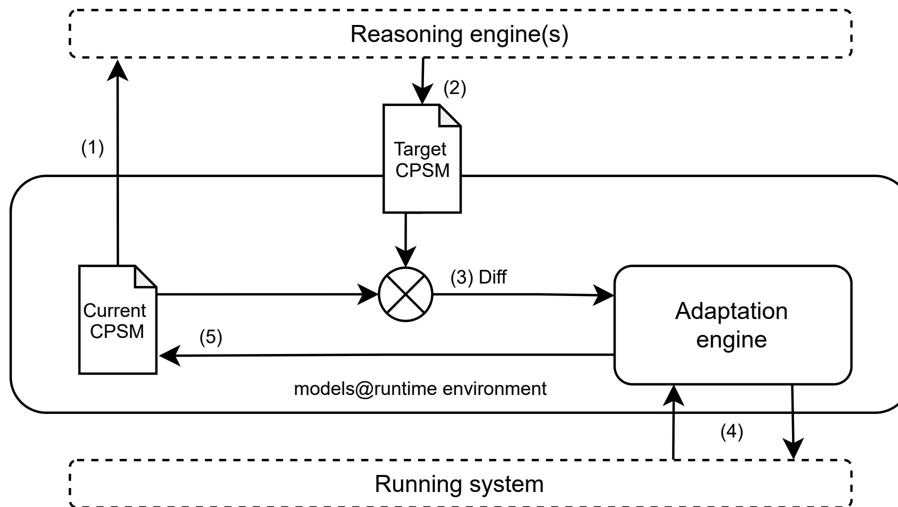



**Figure 3.2:** Architectural overview of CloudMF models@runtime environment [FSR+14]

bi-directional synchronization process portray a crucial aspect when coupling design and runtime together. However, our solution is designed to be based on TOSCA, a standardized specification language, and additionally puts greater emphasis on the visual representation of instance data.

## 3.2 Graphical Modeling Editors

Graphical modeling editors alleviate the development of application models by providing users with a visual representation of the application's structure and ease of use when performing modifications to the model. Through the simple click of a button or by performing conventional mouse gestures, such as drag-and-drop, users can perform various actions that would normally require tedious manual editing of configuration or template files. These editors are often built on top of a DSL, and thus, specifically made for the respective use case or ecosystem. Bergmayr et al. [BBF+18] performed an extensive review of existing cloud modeling languages by comparing them on various aspects and providing a concise overview of each language's main characteristics. Two of their characteristics focus on the modeling tooling support and the support for dynamic service provisioning, i.e. provision computing resources in an automated manner through the use of a provisioning system. Using their findings as a starting point, we compiled a list of various modeling tools that can be used to visually create application models for provisioning purposes. Because their review was focused on languages that promote interoperability between different cloud providers, we also incorporate modeling tools that only operate within a specific ecosystem. Table 3.1 shows this list. To further differentiate the various modeling tools, we analyze each tool on two main aspects. The column *Automatic Deploy* denotes the ability to directly trigger a deployment within the modeling tool without having the need to manually export and upload the model to a provisioning system. On the other hand, the column *Deployment Status* refers to the ability to monitor the status of the deployment within the modeling tool.

| Modeling Tool | Modeling Language | Vendor-Neutral | Modeling Approach | Automatic Deploy | Deployment Status |
|---|---|---|---|---|---|
| *unnamed* | Caglar et al. | Yes | Imperative | No | No |
| Arbitrary UML editor | CAML | Yes | Declarative | No | No |
| Modelio | CloudML | Yes | Declarative | No | No* |
| Winery | TOSCA | Yes | Both | No | No |
| Cloudify Composer | DSL (TOSCA) | Yes | Both | Yes* | No |
| AWS Cloud-Formation Designer | DSL (AWS) | No | Declarative | Yes | No |
| CloudCraft | DSL (HCL) | No | Declarative | No | Yes |

**Table 3.1:** Comparison we made of existing modeling tools

Interestingly, Caglar et al. [CASG13] is the only system that only supports an imperative modeling approach. While this ensures complete customization over the provisioning process, it might hinder novice users who lack technical expertise in certain areas. CAML [BTN+14] provides a mapping to TOSCA, making it possible to utilize TOSCA supported runtimes, such as OpenTOSCA Container, during provisioning. However, since it does not come with a dedicated model editor, it lacks customization and any sort of advanced functionalities. As illustrated in Figure 3.1, CloudMF remains one of the systems that provide a dynamic graphical user interface for monitoring the status of the deployment. Unfortunately, this view is separated from the graphical model editor, and thus, still requires the users to manually provide the application model to the provisioning system. Interestingly, only AWS CloudFormation Designer and Cloudify Composer incorporate a mechanism to automatically provide the application model to a provisioning system. However, both systems lack the ability to visually incorporate deployment information within their graphical modeling editor. Although CloudCraft[1] offers a *live-view* of all components within a running environment, it only supports components that reside within the AWS ecosystem. In fact, to the best of our knowledge, none of the presented tools offer both of the aforementioned aspects. As a result, we plan on extending OpenTOSCA's modeling system, *Winery*, to incorporate these capabilities. With TOSCA's extensive feature set and acknowledged standardization, we are confident that implementing these changes will offer new opportunities and help users throughout the modeling process. In order to demonstrate the use and capabilities of other graphical modeling editors, we take a closer look at AWS CloudFormation Designer and Cloudify Composer in the following.

AWS CloudFormation Designer[2], or in short *Designer*, is a graphical editor that enables its users to visually create and edit AWS CloudFormation templates. It eases the development of computational infrastructure by incorporating graphical representations for each respective template resource and visualizing the interrelationship between them. Since Designer is part of AWS's product catalog, it supports the majority of resource types and services that are also used when defining AWS CloudFormation templates in a textual manner, e.g. Amazon EC2 instances, AWS Elastic Beanstalk,

[1] https://cloudcraft.co/

[2] https://console.aws.amazon.com/cloudformation/designer

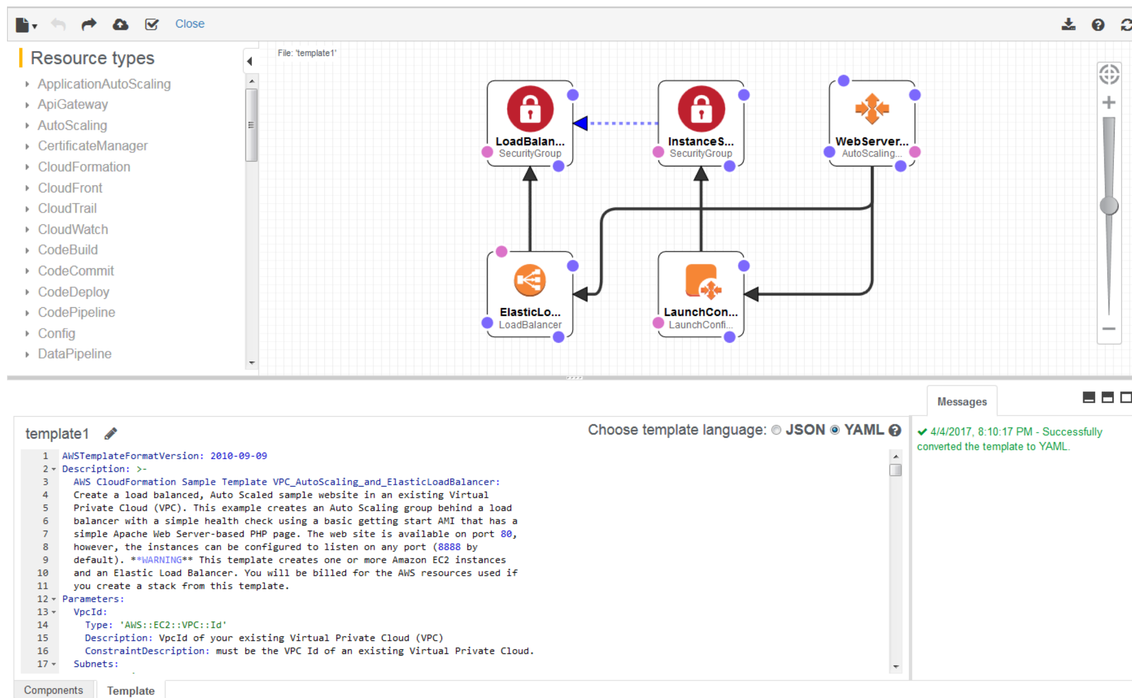



**Figure 3.3:** User interface of AWS CloudFormation Designer [AWSCF20]

and Amazon S3. While CloudFormation templates are formatted as JSON or YAML files, Designer further enriches them by incorporating size and position data that is used to properly visualize the template's content on a two-dimensional editor pane. Figure 3.3 shows Designer's user interface after loading an AWS CloudFormation template. Users are able to add new resources to the current template or create relationships among them via a drag and drop interface. As an example, dragging a resource type from the selection pane onto the editor pane will add the respective resource to the template. In addition, clicking on a resource will display various user actions that are commonly performed on the respective resource type, such as duplication or accessing its documentation. Designer incorporates various mechanisms to further assist users during the modeling process. For instance, it enforces some basic modeling rules that prevent the creation of invalid templates, e.g. it is not possible to directly add an EC2 instance inside a virtual private cloud without creating a subnet beforehand. Furthermore, AWS offers a ValidateTemplate API[3] that enables the validation of existing templates based on various characteristics. This includes the syntactical correctness, the proper declaration of parameters, and the presence of any circular dependencies among resources. By clicking the respective button within the toolbar, users receive immediate feedback about the current state of their template. The graphical user interface also integrates a text editor for directly editing the underlying template during the modeling process. It includes an auto-completion feature that supports users during editing by suggesting the respective properties of a resource type when defining it. After manually editing the template, the model pane indicates its out of sync state, and therefore, needs to be manually refreshed before further modeling actions can be taken. In order to limit the number of valid values when filling in deployment parameters, templates can include regular expressions that will be used later on to validate the users' input before each deployment.

[3] https://docs.aws.amazon.com/AWSCloudFormation/latest/APIReference/API_ValidateTemplate.html

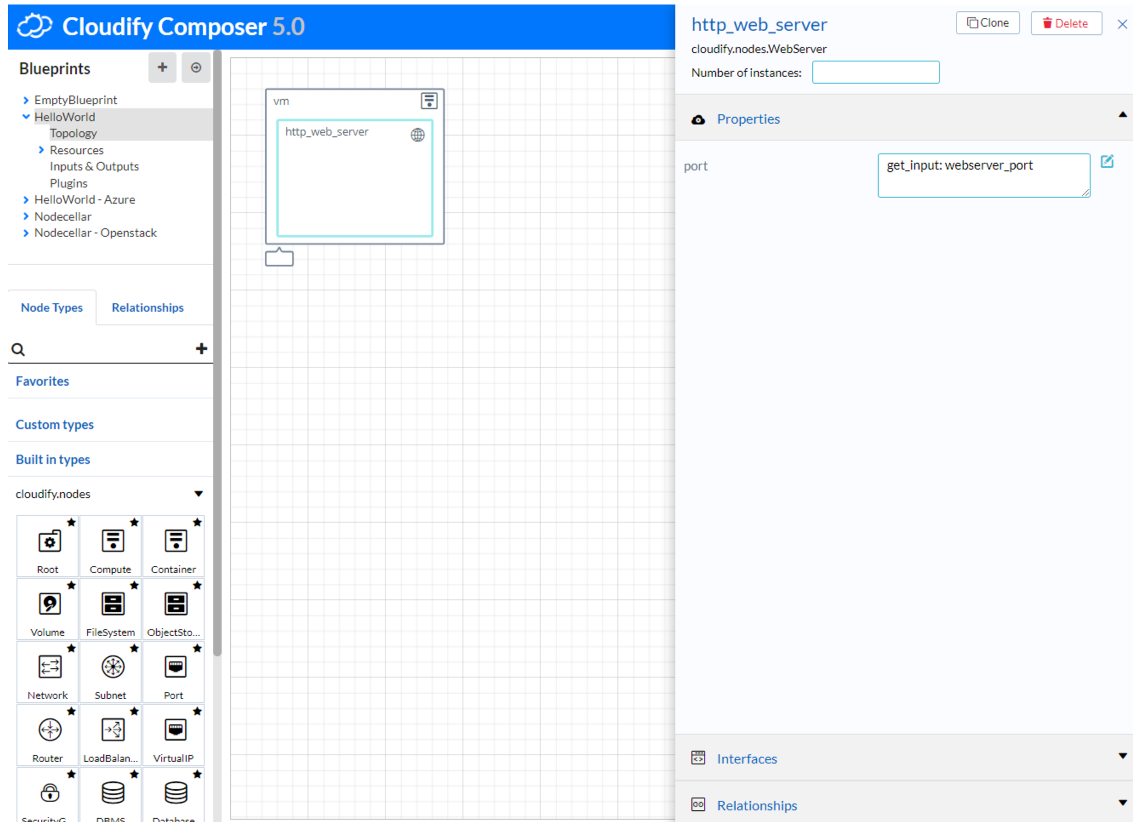



**Figure 3.4:** User interface of Cloudify Composer [CCDR20]

Users are also able to directly trigger the deployment of the current template by clicking the respective deploy button within the toolbar. By clicking it, the template is saved in an S3 bucket and opens the AWS CloudFormation Create Stack Wizard in another tab. The Wizard receives the template's respective URL and, if needed, asks the users to enter any other required information in order to successfully deploy the template. After triggering the deployment, users can monitor the process through the stack event viewer within the AWS CloudFormation Console. While AWS CloudFormation Designer enables users to create and modify templates in a visual-based manner, it is strictly limited to the resource types that reside within the AWS ecosystem. As a result, it is not possible to create templates that contain resources from other external cloud providers. Moreover, although users can trigger the deployment of templates from within the user interface of Designer, it is merely a redirect to an external system that is fed with the respective information rather than visually incorporating deployment information within the user interface.

Cloudify[4] is an open-source cloud orchestration platform that enables users to model cloud applications, including their life cycle operations, in a generic and descriptive manner. Apart from modeling, Cloudify also facilitates deployments and the orchestration of application instances. For modeling applications, Cloudify utilizes a DSL that is based upon TOSCA's YAML Simple Profile [RBL17]. As a result, application models are not bound to a specific ecosystem, and thus, support portability and interoperability across different cloud providers. Their YAML-based models are

[4] https://cloudify.co/

referred to as *blueprints* and are a topological representation of an application. In order to ensure portability, blueprints can be packaged into *blueprint archives* that contain all resources that are needed throughout the life cycle of the application, e.g. installation files, shell scripts, and binaries. Cloudify Manager is part of Cloudify's ecosystem and describes a dedicated environment that allows the management and deployment of blueprints. Uploading blueprints and triggering deployments can either be done through Cloudify Manager's user interface or by utilizing Cloudify's CLI. Instead of manually editing blueprints, Cloudify incorporates a visual-based editor, called Cloudify Composer, that allows users to visually model applications. Figure 3.4 shows its user interface during the modeling of a blueprint. The selection pane on the left contains all defined resource types that can be used to model the application's topology. Dragging a resource type onto the editor pane creates a respective node within the topology. Since Cloudify's DSL is TOSCA-based, all nodes and relationships contain characteristics that further describe their implementation and behavior details. As a result, when clicking on a node, a detailed view is displayed that shows the node's properties, interfaces, and relationships. Cloudify distinguishes between different Relationship Types. For instance, a *contained-in* relationship is created by visually nesting one node within another. A database server hosted on a virtual machine is a common example of such a relationship. In contrast, a *connected-to* relationship between two different nodes is created by clicking on the right side of the source node and dragging the mouse to the left side of the target node. Similar to AWS CloudFormation Designer, Cloudify Composer also directly incorporates an integrated text editor that enables manual editing of the underlying YAML file. Cloudify Composer offers the ability to validate the contents of a blueprint "to ensure that logical concepts are valid" [CCDR20]. While users can directly upload blueprints to the Cloudify Manager, it is not possible to trigger deployments from within Cloudify Composer.

## 3.3 Formal Verification of TOSCA Models

During modeling, novice application architects may encounter more errors due to subtle misconfigurations within the model that otherwise require deep knowledge to overcome. This can include using the correct Relationship Types between two Node Templates or ensuring that the deployment can be completed within a finite amount of steps. As a result, there are several works that concentrate on the formal verification of TOSCA application models before runtime:

Brogi et al. [BDS18] developed an open-source prototype, called *Sommelier*, that automatically validates TOSCA application topologies based on different validity conditions. Their main focus lies in verifying the correctness of all inter-component relationships, e.g. whether the sources of relationships properly align with their respective targets. For instance, connecting a NoSQL database component with a SQL DBMS component is an example of such a violation. Their prototype implementation takes a CSAR file as input and either confirms the correctness of the application or outputs a list of all present violations. Within their work, they also mentioned OpenStack's *TOSCA parser*[5] as an additional tooling solution to further verify an application on its syntactic correctness based on TOSCA's official grammar specification.

[5]https://github.com/openstack/tosca-parser

Apart from verifying the structural correctness of application topologies, formal verification can also be performed on characteristics that go beyond the design aspect. Yoshida et al. [YOF15] formalized the behavior of application topologies in order to formally prove the liveness property of application deployments. Therefore, they introduced the concept of *total reachability*, i.e. whether an application instance eventually converts to a final state where all of its components are started and fully functioning, independently of the transition sequence that occurred during the deployment process. In order to properly model the deployment of an application, Yoshida et al. utilized the algebraic specification language *CafeOBJ*[6] and formalized necessary transition rules that reflect the semantics between common Node operations and Relationship Types within TOSCA. In comparison to other works, their solution is based on theorem proving and furthermore provides an abstract approach without depending on the implementation details of the underlying management system. Chareonsuk and Vatanawood [CV16] proposed an alternative verification process of TOSCA applications based on the verification language *Promela*[7]. Based on their works, users are able to define their own safety properties in order to express states that shall never occur during the runtime of the application. Their implementation automatically transforms TOSCA templates and their appended BPEL processes into Promela code. A SPIN model checker can then be used to verify whether any safety property is being violated during any time of operation. The work of Saatkamp et al. [SBKL18] focused on integrating formalized patterns in order to automatically detect problems within TOSCA-based deployment models. Their solution builds upon patterns since they provide a structured approach to how and when to resolve problems in different scenarios. For instance, the secure channel pattern addresses the problem and context of securely exchanging information across a public network. In order to automate the detection of configurations that violate such patterns, they used the logic programming language *Prolog* to formalize patterns in a machine-readable format. Correspondingly, a fact is then created for each existing component within the application's topology. By solely providing an application's deployment model, their implementation is capable of applying these formalized patterns onto the generated topology facts and automatically deduce any contained problems, including the affected components and the associated pattern.

All presented works enable application architects to formally prove their model on certain criteria before its actual deployment. This is most beneficial to novice users in order to catch model errors early on throughout the process. Moreover, this avoids unnecessary deployments of faulty models that would have failed anyway. As a result, the inclusion of such a verification system can be of great advantage in our case. However, implementing our own validator is a complex challenge by itself, and thus, falls out of the scope of our work. Nonetheless, external solutions like Sommelier can be easily included within our pipeline in order to extend the capabilities of our implementation.

## 3.4 TOSCA-specific Modeling Tools

Since our implementation will be built upon the TOSCA standard, we take a closer look at the various tooling support that is geared to help successfully develop TOSCA-based application models. Some of these tooling solutions can either be directly integrated within the modeling system or

[6] https://cafeobj.org/

[7] http://spinroot.com/spin/Man/promela.html

offered as standalone applications. In particular, novice application architects, who lack important knowledge in certain areas, benefit from such solutions due to the reduction of unnecessary overhead and automation of various tasks.

As mentioned in Section 2.3, the provisioning of cloud resources is represented by the definition of a respective Node Type. However, the creation of these types requires deep knowledge about the respective cloud provider, and thus, can easily become overwhelming for users. DrACO [BCS16] is a web-based application that enables users to automatically download TOSCA-conform type definitions of popular cloud offerings through its graphical user interface. This can include the provisioning of virtual machines on Amazon AWS or computational nodes on different PaaS offerings. These definitions can act as a starting point for further customization and are especially useful for users who lack experience as type architects. Since DrACO is accessible as a standalone web application, it is compatible with every TOSCA-conform modeling system.

Throughout the years, there have been major contributions to the OpenTOSCA ecosystem that provide users with additional support during the modeling process. For instance, the work of Hirmer et al. [HBBL+14] enables the creation of *incomplete* topology models in order to further simplify the modeling process. Before deployment, these incomplete models are automatically enriched with the necessary data to ensure their syntactic and semantic correctness. This allows users to concentrate on the implementation of business-relevant components and leave the definition of other components to experts. Harzenetter et al. [HBF+18] discuss a similar approach by enabling users to utilize cloud patterns as a vendor and technology-agnostic components directly in Topology Templates. Furthermore, there are certain cases where users need to make adjustments to an existing application topology, e.g. due to privacy concerns or financial issues. This can include the substitution or splitting of the infrastructure an application is hosted on. However, due to the immense amount of required expertise, changing the underlying infrastructure of an application can be a complex and error-prone task. As a result, Saatkamp et al. [SBKL17] proposed a solution for such scenarios by implementing an automated approach that is able to split topology models based on the users' selection. Their solution represents a viable alternative to the otherwise manual and potentially more error-prone approach when making modifications to an existing model. So far, we considered a deployment of an application to be successful whenever no errors occurred during its deployment process. However, a technical correct deployment execution does not necessarily ensure the proper functioning of a system. Therefore, Wurster et al. [WBKL18] introduce a modeling concept that enables users to include custom written test code within the deployment model. These tests are then executed by the respective runtime, e.g. OpenTOSCA container, to verify that the application meets all requirements and is functioning properly.

Similar to the presented works, we plan to further expand the range of functionalities within Winery's topology modeler. Our contribution consists of an integrated validation system and a live-modeling feature that enables the deployment of TOSCA-based applications at modeling time.

# 4 Use Case Scenario

This chapter presents the current state of the art for modeling and deploying a TOSCA-based cloud application. It gives an overview of all involved system components, their inter-communication, and in particular, how users interact with them. We demonstrate the various steps involved by means of an example application topology. The chapter concludes by discussing the current limitations and possible improvements to this approach.

## 4.1 Workflow

Figure 4.1 depicts the current workflow and respective actions when developing an application model. We assume that the users' goal is to successfully model and test a TOSCA-based application. Moreover, we expect that all necessary types and artifacts are defined in advance and are accessible to the users. Throughout the process, the users interact with two separate systems: the modeling system and the runtime system. The modeling system is used to model and package the application in a self-contained format. The runtime system, on the other hand, is responsible for instantiating and managing applications in their respective target environment. Both systems are linked through the textual description of an application, e.g. a CSAR file, provided by the users. Given these two systems, the users' workflow consists of four consecutive steps. First is the modeling step (see 1 in Figure 4.1), which consists of defining the application's topology and management plans that are needed throughout its life cycle. Since we assume that all necessary TOSCA entities are already provided to the users, this step mostly involves syntactically linking the respective Node and Relationship Templates while also specifying concrete values for each type property. This step can be done either in a textual manner or through the use of a visual-based editor, such as Winery's topology modeler. When the users finish modeling, the next step involves the packaging of the respective application model as a CSAR file (see 2 in Figure 4.1). Since these files are self-contained, and therefore comprise all required definitions and implementations, they act as an exchange format between the modeling and runtime systems. After providing the runtime system with the respective CSAR file, the users can manually trigger the deployment (see 3 in Figure 4.1). Therefore, the runtime system executes the build plan of the application, and if necessary, requests additional build plan parameters from the users. If a build plan can not be found within the CSAR file, the runtime system is able to dynamically generate a respective build plan based on the application's topology. The last step consists of evaluating the status of the deployment. The deployment status is usually provided by the runtime system, e.g. through logs or state information, but can also be manually checked by directly interacting with the application (see 4 in Figure 4.1). If everything functions as expected, the users can stop here and, for instance, distribute the CSAR file to other parties. In the case of errors, the users enter a feedback loop by reiterating the previous steps in order to refine or correct mistakes within the application's model. Thus, the modeling of an application can take multiple loop iterations until the model is complete and free of any errors.

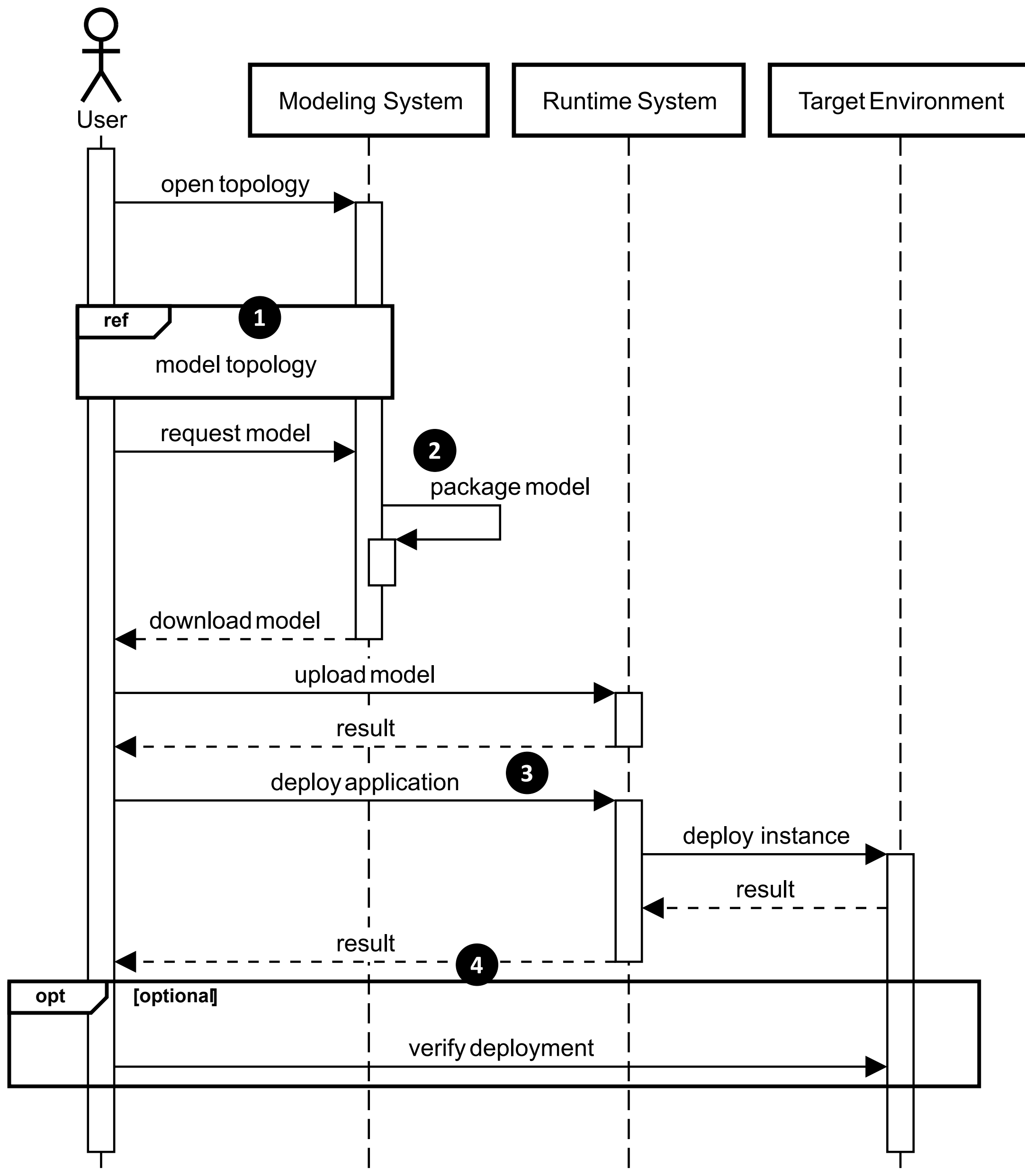



**Figure 4.1:** Sequence diagram illustrating the actions and messages when developing an application

## 4.2 Example Topology

In order to demonstrate the structure of TOSCA-based applications and elaborate on the limitations of the current approach, we refer to the *MyTinyToDo-MySql_OpenStack* Service Template which is part of the OpenTOSCA's public TOSCA definitions repository[1]. When instantiated, it installs the PHP application MyTinyToDo as a Docker Container on an OpenStack Ubuntu virtual machine alongside a MySQL database. The MyTinyToDo[2] application enables users to simply manage todo lists that are persistently stored within a database instance. Figure 4.2 depicts the topology of this application using the visual notation by Breitenbücher et al. [BBK+12]. We will use this topology as an example application for the remainder of this thesis.

The Topology Template contains six Node Templates and six Relationship Templates. Our application's foundation consists of an OpenStack Node Template that allows us to connect to an existing OpenStack platform and provision various computing resources. Therefore, the Node Type contains properties that specify the platform's endpoint and credentials, which must be provided either during modeling or deployment time. Within the OpenTOSCA ecosystem, properties that contain the keyword *get_input:* are automatically included within the build plan, and thus, filled in during the application's deployment. Hosted on this OpenStack, is an Ubuntu 18.04 virtual machine, indicated by the *Ubuntu-VM_18.04* Node Template within the topology. Its respective Node Type holds properties, such as virtual machine type or predefined user credentials, that are used to configure the virtual machine. The remainder of the application is split between two component stacks that are both hosted on the Ubuntu virtual machine. While the left stack focuses on the installation of the MyTinyToDo Docker container on a Docker Engine, the right stack sets up a MySQL database on a MySQL DBMS instance. The connectsTo Relationship Template ensures the connection between the actual MyTinyToDo application and the MySQL database by executing a shell script that stores the respective database credentials within a PHP configuration file. If successfully deployed, the users are able to access and interact with the user interface of the MyTinyToDo application through the public IP address of the underlying Ubuntu virtual machine.

## 4.3 Example Use Case

The modeling and configuration of TOSCA-based application models can require knowledge in various areas. Even smaller topologies, as depicted in Figure 4.2, necessitates experience and insight about the technical details of the used components. Experienced application architects possess the skill to model entire topologies as a whole and avoid common mistakes that would otherwise slow down progress or lead to unnecessary errors during the process. In contrast, novice users often choose a bottom-up approach that comprises several steps. Figure 4.3 depicts this iterative approach. Rather than completing the model as a whole, beginners would model parts of the topology one after the other, referred to as submodels. This method is advantageous because users are able to verify and optimize each submodel prior to moving forward, which reduces the likelihood of prolonging errors. In our use case, the users would start with a topology that solely consists of the OpenStack Node Template, fill in all necessary properties, and eventually export the

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

[2] https://www.mytinytodo.net/

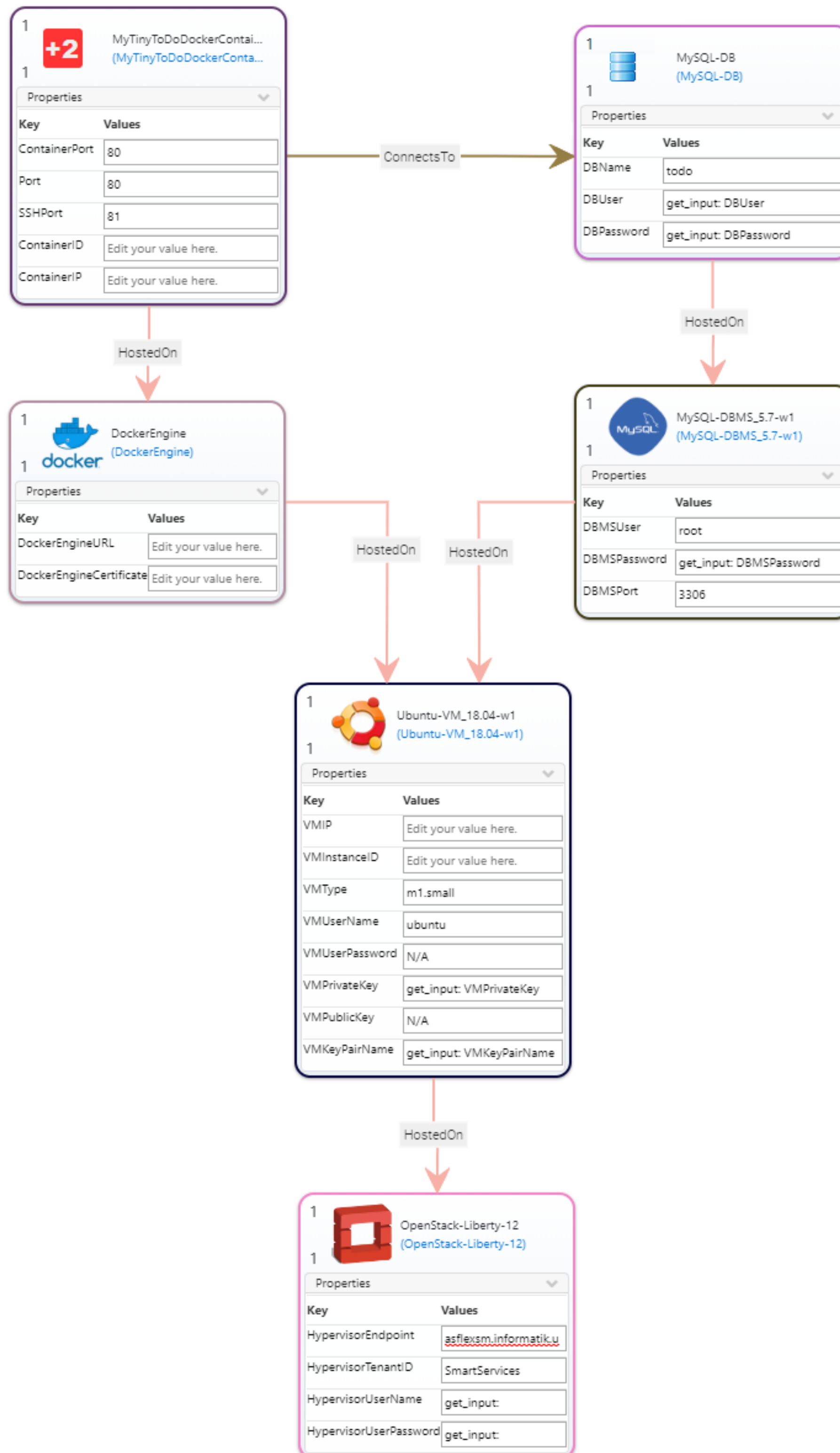



**Figure 4.2:** Topology of our example application

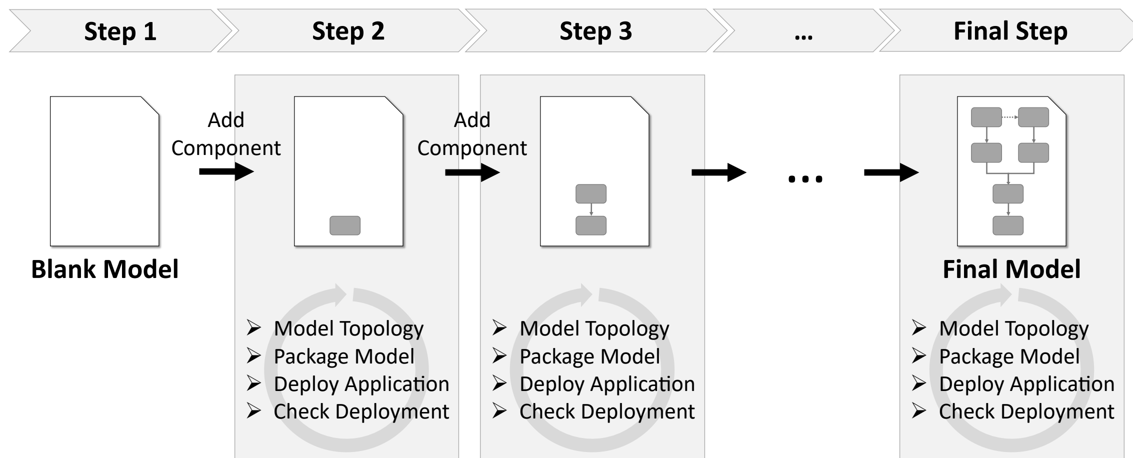



**Figure 4.3:** Typical iteration steps when developing a new application model

model as a self-contained file. This file can then be provided to a runtime system in order to trigger the deployment of the submodel. If there are any errors during deployment or the application does not work as expected, the users remain within the feedback loop as explained in Section 4.1. If no errors occur and the application functions properly, the users would then refine the submodel by adding the Ubuntu Node Template to the current topology, thus, advance to the next iteration. This iterative process of refining and testing the topology is repeated until the entire application model is fully completed and meets the users' requirements.

While this iterative approach enables beginners to develop application models in a trial and error like approach, there are certain limitations that can be improved upon. Some of these limitations stem from the problem that manual action is needed in order to advance through the workflow process. For instance, users are currently obligated to manually export and provide the application model to the runtime system before each deployment. Furthermore, since every deployment is independent of the ones before, termination of the previous instance is required in order to avoid unnecessary standby-time of computational resources, e.g. Ubuntu VM instances continuing to run in the background even though they are no longer in use. This proper termination might not be obvious to novice users, and thus, potentially lead to high operational costs or even unsuccessful subsequent deployments. As a result, one could propose the reuse of components throughout subsequent deployments. As an example, we can avoid the reprovisioning of the Ubuntu VM instance if the respective Node Template remains unchanged throughout two subsequent deployment steps. This would reduce the deployment time and the number of allocated computing resources throughout the modeling process. Apart from defining the topological dependency among Node and Relationship Templates, users also have to properly define values for each respective type property. Especially for inexperienced users who lack knowledge about the technical details of each Node Type, this can become a difficult task to overcome as it is not apparent at first glance what to fill in for each property. As an example, users need to explicitly know what VM types exist on OpenStack when configuring the Ubuntu VM Node Type. Therefore, it would be beneficial if a type definition includes information about valid values or ranges for each of its properties. Type architects could define them in advance, and thus, support application architects during the modeling process.

## 4.4 Requirements

Although our example application topology only consists of a few Node and Relationship Types, it demonstrates various important aspects within the TOSCA modeling space and serves as an appropriate reference model for the remainder of this thesis. While the current workflow, as presented in Section 4.1, represents a complete and valid method to model and deploy applications, it also imposes various limitations in certain scenarios. For instance, while the modeling of smaller topologies can still be done within a few iterative steps, it becomes obvious that it can become cumbersome and notably time-consuming when dealing with applications that consist of numerous components. As illustrated in Section 4.3, the constant switch between modeling and runtime systems and the need for manual execution steps can lead to frustration and an inefficient way of working. Furthermore, by extending the modeling system with a proper validation system, that enables the recognition of misconfigurations early on, we can eventually help decrease the number of errors during deployment time. This also lowers the amount of expertise required, and thus, makes it particularly more accessible to novice application architects. As a result, we define a list of requirements that - when implemented - can help mitigate the mentioned drawbacks and lead users to work in a more efficient manner. We will also use this list to establish the foundation of our concept and validate our prototype implementation later on.

- **Requirement 1 (R1)**
  The concept should enable users to trigger the deployment of the current modeled application without being required to switch between the modeling and runtime system. Users can decide which runtime system to use, and moreover, pass all required information that is needed throughout the deployment, e.g. initial deployment parameters.
- **Requirement 2 (R2)**
  Apart from deploying applications, users should also be able to manage those instances throughout their life cycle, e.g. starting or terminating instances, from within the modeling system.
- **Requirement 3 (R3)**
  The communication and data exchange between modeling and runtime systems should be based on a standardized protocol and continue to preserve the autonomy among each other.
- **Requirement 4 (R4)**
  The modeling system needs to enable users to retrieve relevant information about the current deployment and present them in a visualized manner. Such information comprises log data, state information, or output parameters.
- **Requirement 5 (R5)**
  Users are capable of reconfigure existing deployments to match the current state of the model. This includes the reuse of components across consecutive deployments in order to save time and computing resources.
- **Requirement 6 (R6)**
  Users are capable of deciding whether to deploy the full application or only parts of it. This includes the starting and termination of specific Node Instances during and after deployment.

- **Requirement 7 (R7)**
  The modeling system should incorporate a means to facilitate fault-free modeling of *deployable* application models. Hereby, the term *deployable* refers to application models that can be successfully deployed without the occurrence of any errors.

The option to directly trigger the deployment within the modeling system represents an alternative approach to increase the users' efficiency during the modeling of TOSCA-based applications. We will use the term *live-modeling* to describe this new functionality. Requirements **R1**, **R2**, and **R3** incorporate the minimum criteria for such a system while also ensuring proper usability. Requirement **R4** is built upon existing modeling solutions that already comprise visual-based modeling within their system and extends them by integrating appropriate information about the current deployment status. Requirement **R5** addresses the issue of reusing computing resources as mentioned in Section 4.3. Requirement **R6** deals with scenarios where users already modeled bigger topologies, but only desire deploying or testing part of it. Instead of deploying the whole application at once, this requirement ensures more fine-grained control over which parts of the application are being deployed. Requirement **R7** helps prevent modeling errors early on and especially supports beginners who lack knowledge about the technical details of the respective components.

# 5 Concept

This chapter presents our concept for deploying TOSCA-based applications during modeling time and integrating a type-based validation system within TOSCA's ecosystem. Section 5.1 covers the required prerequisites for the remainder of this chapter. Section 5.2 provides an overview of the concept's fundamentals by demonstrating an alternative workflow for creating application models in an iterative manner by integrating validation and deployment capabilities within the modeling system. Subsequently, the following two chapters cover the two main topics of our concept by explaining their modalities and describing their usefulness within the modeling process in detail. The presented information acts as a foundation for the subsequent chapter which focuses on the prototypical implementation of this concept. The chapter concludes by revisiting the use case scenario from the previous chapter by incorporating the concept's components.

## 5.1 Prerequisites

Before elaborating on the details of our concept, we set a number of assumptions concerning the setup and capabilities within the modeling and deployment space. As mentioned in Section 4.1, we assume the presence of two TOSCA supported systems: the **modeling system** and the **runtime system**. Users utilize the modeling system to define resource components and make use of them to model applications in a portable manner. We further assume that the modeling system contains a graphical editor that enables users to alternatively model an application's topology through its user interface. These models can be packaged and provided to the runtime system to deploy and manage application instances within its defined target environment. Moreover, the runtime system is capable of interpreting declarative application models, i.e. dynamically deriving deployment tasks if not provided by the users. While both systems operate independently of each other, they also offer API endpoints that allow the execution of actions and retrieval of model relevant information. This includes the packaging and uploading of application models, triggering deployments, managing instances, and acquiring data about the current status of deployments. We utilize these capabilities to implement the following two components within the modeling system: a **live-modeling** feature and an integrated **input validation system**. These two concepts intend to assist users during the modeling stage of developing TOSCA-based application models. We are confident that they provide a necessary means to enable users to work in a more efficient manner.

## 5.2 Overview

In order to tackle the aforementioned issues and meet the requirements described in Section 4.4, we propose an alternative modeling workflow that enables the simultaneous deployment of applications during modeling time. Furthermore, the concept incorporates a means that facilitates the fault-free

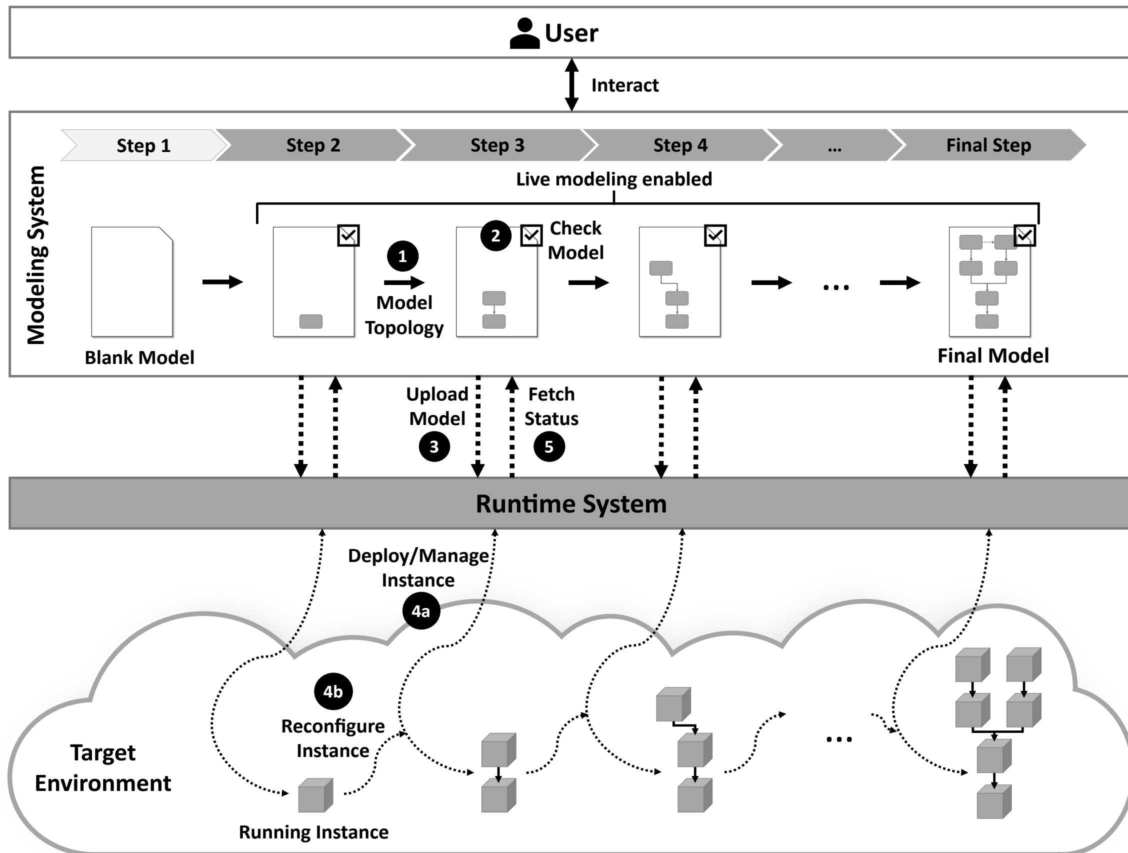



**Figure 5.1:** Iteration steps when developing an application model incorporating both concepts

modeling of deployable application models. Since our application does not require any custom deployment logic, we use a declarative model to describe the structure of our application and let the runtime system handle how to explicitly deploy it. Before modeling, we assume that all required type definitions, i.e. Node Types and Relationship Types, are already declared and stored within the modeling's repository. In contrast to our previous scenario, these type definitions now also include suitable validation information for some of their type properties that help us assign correct values to them. Figure 5.1 depicts the workflow of our alternative use case scenario. Similar to before, we choose an iterative approach during modeling, i.e. deploying and testing the existing topology before adding another component to it. Each iterative step comprises up to five different subtasks. It is possible and also encouraged in certain cases to perform some of these subtasks more than once within each step. First is the modeling task, which remains unchanged when compared to the previous approach (see 1 in Figure 5.1). In contrast to before, after adding new templates and assigning values to their properties, we can now validate our topology by utilizing the new validation information that was incorporated within the type definitions (see 2 in Figure 5.1). This ensures that our property values comply with the type architects' intended value assignments, and thus, prevent potential deployment errors early on. The validation subtask can be further extended by including additional tooling solutions that help spot misconfigurations or syntactical errors, such as Sommelier or DrACO. The third subtask consists of packaging and uploading our application to the runtime system (see 3 in Figure 5.1). Contrary to before, we no longer perform this task manually

anymore. Instead, we are now able to use the live-modeling feature of the modeling system to trigger this action. When the model is completely uploaded, we can trigger its deployment and, if required, provide values to each respective build plan input parameter while remaining within the modeling system. If there is currently no running instance, the runtime system will deploy a new instance (see 4a in Figure 5.1). Otherwise, the runtime system will reconfigure the current instance so it matches the updated model (see 4b in Figure 5.1. Throughout the whole deployment process, the modeling system fetches data from the runtime system and provides us with visual feedback and textual information about the status of our deployed/reconfigured instance (see 5 in Figure 5.1). Based on this, we can make corrections to our current topology if there were any errors during its deployment/reconfiguration. After adjusting the topology accordingly, we repeat all respective subtasks to see whether our changes fixes the former issue. If there are no errors, we can continue adding new components to our application topology until we reach our final model.

## 5.3 Live-Modeling

During development, users can prematurely spot and fix errors within their models by testing their applications early on. However, as shown in Section 4.1, users are currently obligated to constantly switch between the modeling and runtime systems in order to model and deploy an application. While this approach is valid and straightforward, all of these tasks require manual action from the users, and therefore, become cumbersome and time-consuming when performed multiple times. This becomes increasingly relevant as the applications become more complex because the number of iterations required for completion creates an overwhelming amount of manual input that is impractical for users to perform. Hence, we propose a live-modeling feature that allows users to directly trigger the automated deployment of application models within the graphical editor of the modeling system. Thus far, users were obligated to perform various actions on both systems whenever they intend to deploy an instance of their modeled application. Instead of requiring manual action from the users, most of these actions are now taken care of by the modeling system. This shift in responsibility allows users to focus more on the modeling aspect within the aforementioned feedback loop. In addition, by limiting the users' interaction to only one system, we reduce the overall overhead and amount of oversight required when operating both systems at once. It is important to note that live-modeling is only designed for the declarative modeling approach. As a result, when using the term *model* or *modeling*, we hereby only refer to defining the application's components; We are not specifying how they are deployed. Thus, the users' modeling actions only consist of defining the application's Topology Template, i.e. visually creating Node Templates and connecting them through the use of Relationship Templates. When the packaged application is provided to the runtime system, it becomes then the runtime system's responsibility to dynamically derive all necessary deployment tasks, generate a suitable build plan, and execute it accordingly. This modeling approach ensures that users can concentrate on creating the application's topology rather than dealing with the technicalities on how to deploy each component in detail. Particularly novice users, who lack such expertise, can vastly benefit from this approach.

Figure 5.2 illustrates an alternative workflow that incorporates this live-modeling concept. In accordance with requirement **R1**, it now becomes the modeling system's responsibility to directly communicate with the runtime system in order to deploy and manage application instances. This is achieved by directly invoking the runtime system's respective API endpoints from within the modeling system instead of relying on manual user inputs. Furthermore, replacing manual and

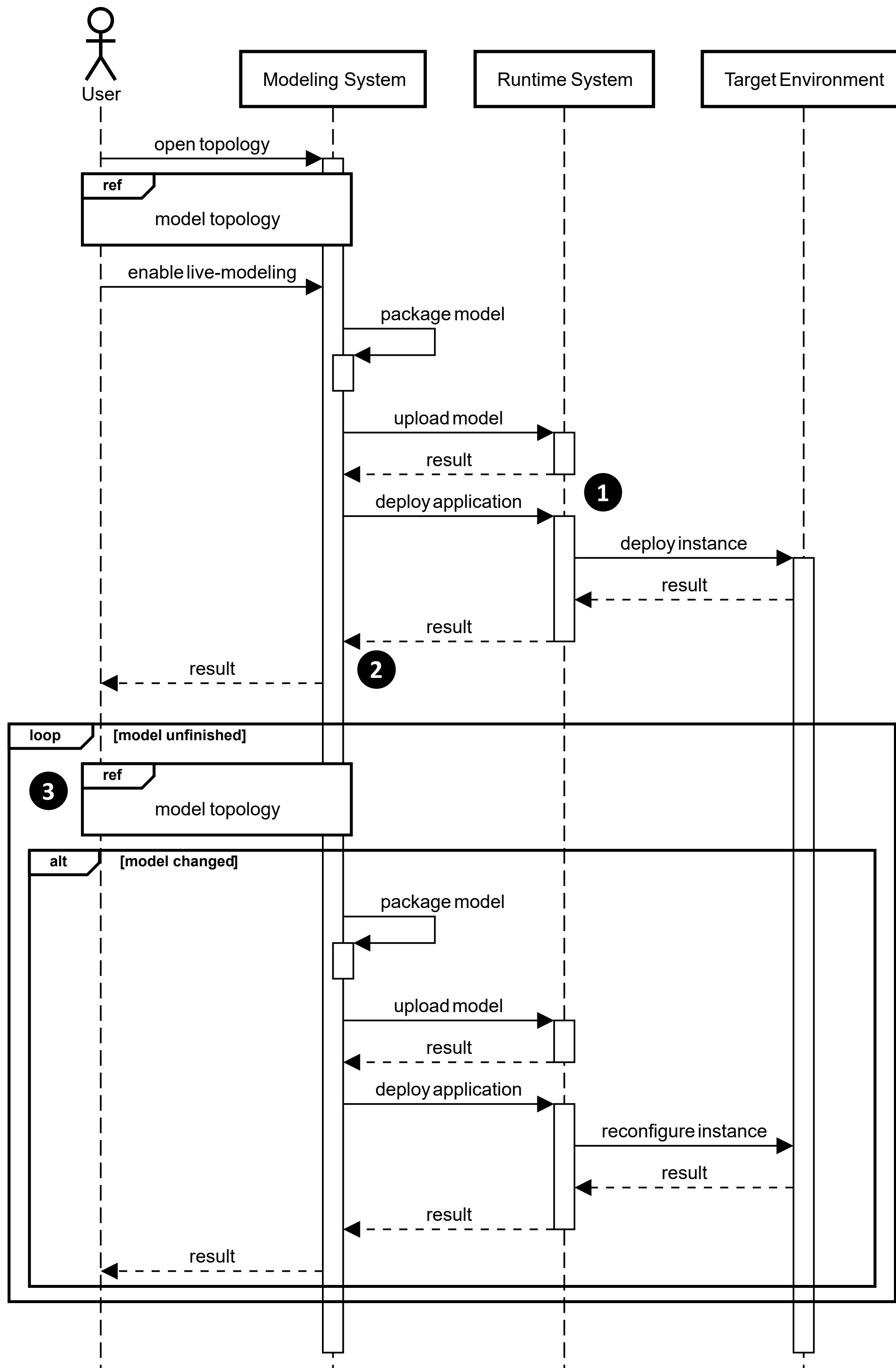



**Figure 5.2:** Sequence diagram illustrating the actions and messages when developing an application using our live-modeling feature

repetitive tasks with automated equivalent processes decreases the likelihood of errors during the operation [BBKL13]. However, before users reenter the familiar feedback loop, an initial setup is now required to properly configure the connection between the modeling and runtime systems. When enabling live-modeling, users must specify the location of the runtime system they want to use first, e.g. through a URL. After checking the availability of the runtime system, the modeling system automatically packages the currently opened application model, uploads the created CSAR file to the runtime system, and triggers its deployment (see 1 in Figure 5.2). If the build plan requires any input parameters, the modeling system will ask the users to enter suitable values for each of them. This deployment is being used to test and gather valuable information about the current application model when deployed in its respective target environment. When the deployment finishes, the modeling system retrieves its results and any log messages that were generated throughout the process (see 2 in Figure 5.2).

Following this initial setup task, the users find themselves within the usual feedback loop again. The modeling workflow remains consistent with the previous approach. Users retrieve type definitions from the modeling system's repository to instantiate Node and Relationship Templates in order to model the Topology Template of their application. However, now (re-)deploying the model diverges significantly from the previous approach. Instead of deploying one instance after the other, the new workflow demands that there is only one running instance at any given time. This is done by reconfiguring the current instance such that it matches the updated model. By doing so, we implicitly mitigate an issue within the previous workflow where it was possible to end up with multiple running instances if users forgot to properly terminate instances before triggering new deployments. In order to reflect these changes within the runtime system, it is now the modeling system's responsibility to detect changes simultaneously while the users are editing the application model (see 3 in Figure 5.2). A change within the model can stem from various sources, such as adding a new Node Template, creating a Relationship Template between two Node Templates, or modifying the properties of a Template. Thus, it is essential to detect these changes on a recurring basis to ensure proper synchronization between the modeling and runtime systems. Algorithm 5.1 shows how to decide whether two Topology Templates are considered unequal. By temporarily storing the last deployed Topology Template, we can utilize this algorithm to detect whether any changes have been made since the last deployment. Lines 1-6 are used to prematurely exit the algorithm in case the number of Node or Relationship Templates is unequal. The remaining lines investigate for each Node and Relationship Template whether the same entity can be found within the other Topology Template. As denoted in line 8 and 22, we compare each Node and Relationship Template tuple based on their semantic equality. `NotEqual` represents an implementation-dependent function that compares two data objects to determine if they are syntactic equivalent.

When it comes to the reconfiguration of instances, there are three different approaches. One can either redeploy the application model, transform between two different application models, or adapt the current instance. Redeploying the model means that the previous instance is terminated and the runtime system deploys a new instance from scratch using either the updated model. A redeployment ensures that the application instance is running through its whole life cycle and is independent of the deployment from the previous iteration loop. Apart from model changes, redeployments are also necessary whenever the users intend to tweak any of the input parameters that are used during the build plan execution of the deployment. Transforming the current instance, on the other hand, is particularly useful when Node Instances can be reused to match the deployment result of the newly provided application model. Figure 5.3 shows a basic example of this concept. Here, we can reuse the already deployed virtual machine during the reconfiguration step since its configuration

**Algorithm 5.1** Determine whether two Topology Templates are unequal

**Input:** TopologyTemplates $T_1, T_2$
**Output:** $T_1 \neq T_2$

```
1: if T1.NodeTemplates.length ≠ T2.NodeTemplates.length then
2:     return True
3: end if
4: if T1.RelationshipTemplates.length ≠ T2.RelationshipTemplates.length then
5:     return True
6: end if
7: for each NodeTemplate n1 in T1.NodeTemplates do
8:     n2 ← {n ∈ T2.NodeTemplates | n ≡ n1}
9:     if n2 is undefined then
10:        return True
11:    end if
12:    if NotEqual(n1.Properties, n2.Properties) or
13:       NotEqual(n1.PropertyConstraints, n2.PropertyConstraints) or
14:       NotEqual(n1.Requirements, n2.Requirements) or
15:       NotEqual(n1.Capabilities, n2.Capabilities) or
16:       NotEqual(n1.Policies, n2.Policies) or
17:       NotEqual(n1.DeploymentArtifacts, n2.DeploymentArtifacts) then
18:        return True
19:    end if
20: end for
21: for each RelationshipTemplate r1 in T1.RelationshipTemplates do
22:     r2 ← {r ∈ T2.RelationshipTemplates | r ≡ r1}
23:     if r2 is undefined then
24:        return True
25:     end if
26:     if NotEqual(r1.Properties, r2.Properties) or
27:        NotEqual(r1.PropertyConstraints, r2.PropertyConstraints) or
28:        NotEqual(r1.RelationshipConstraints, r2.RelationshipConstraints) then
29:        return True
30:     end if
31: end for
32: return False
```

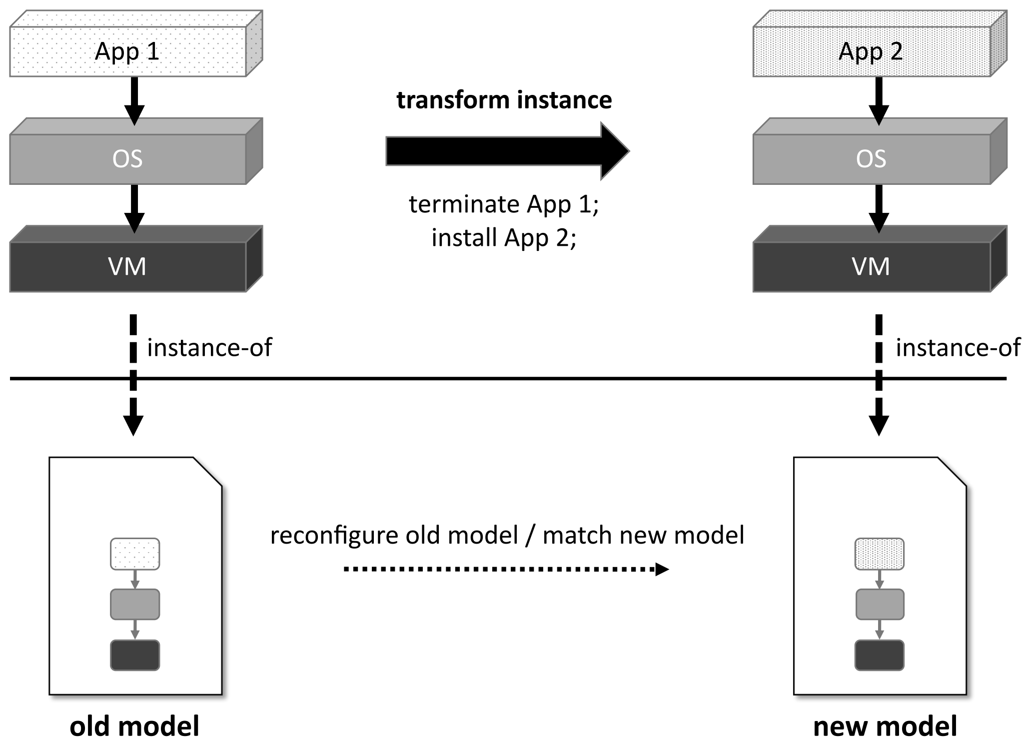



**Figure 5.3:** Basic example of transforming a running instance to match new model

did not change throughout the loop iteration. In order to match the updated application model, we only have to terminate *App 1* and install *App 2* on top of the operating system instance. These steps can be dynamically derived and aggregated within a workflow model so the runtime system can automatically execute them during operation. It is important to note that transformations can also occur in the opposite direction, i.e. terminating running Node Instances to match the updated model. When compared to redeployments, transforming an instance can take less time because every Node Instance does not have to be installed or terminated. Transforming instances represent a suitable solution for fulfilling requirement **R5**. However, since transforming an instance between two application models is out of the scope of this thesis, we omit any further details and presume that our runtime system has such capabilities.

Up until now, we only dealt with reconfiguration scenarios that deal with the transition between two different application models. However, as mentioned in requirement **R6**, there are certain situations where reconfiguring the current instance only encompasses one application model. Instead of deploying the whole application at once, users have control over deploying only a subset of the actual topology. Hence, we refer to this kind of deployment as *delta deployment*. The ability to deploy only a subset of a topology is useful in various scenarios. For instance, if users are only interested in testing a certain part of the application, it can significantly save time during deployment due to the reduced number of Node and Relationship Instances that need to be instantiated. In addition, delta deployment also enables the manipulation of Node Instances even after the initial deployment. As a result, users can gradually start one Node Instance after the other, and therefore, gain insight as to how the application behaves after every step. This is useful in cases, when users are troubleshooting to find out why, and importantly, when a complete deployment fails. In order

**Algorithm 5.2** Compute source and target subsets for delta deployment

**Input:** TopologyTemplate $T$, NodeTemplate $N$, AdaptAction $A$
**Output:** SourceNodeTemplates $S_{NT}$, SourceRelationshipTemplates $S_{RT}$, TargetNodeTemplates $T_{NT}$, TargetRelationshipTemplates $T_{RT}$

```
1: S_NT, S_RT, T_NT, T_RT ← {}
2: for each NodeTemplate n in T.NodeTemplates do
3:     if n.instance.state is NodeInstanceStates.STARTED then
4:         S_NT ← S_NT ∪ n.id
5:     end if
6: end for
7: for each RelationshipTemplate r in T.RelationshipTemplates do
8:     if n_1, n_2 ∈ S_NT where n_1.id = r.source.ref ∧ n_2.id = r.target.ref then
9:         S_RT ← S_RT ∪ rel.id
10:    end if
11: end for
12: Temp_NT ← {N.id}, Min_NT ← {}
13: while Temp_NT.length > 0 do
14:     temp_id ← Remove first element in Temp_NT
15:     Min_NT ← Min_NT ∪ temp_id
16:     if A is AdaptAction.START_NODE then
17:         Temp_RT ← {r ∈ T.RelationshipTemplates | r.source.ref = temp_id}
18:     else if A is AdaptAction.TERMINATE_NODE then
19:         Temp_RT ← {r ∈ T.RelationshipTemplates | r.target.ref = temp_id}
20:     end if
21:     for each RelationshipTemplate r in Temp_RT do
22:         if A is AdaptAction.START_NODE then
23:             Temp_NT ← Temp_NT ∪ r.target.ref
24:         else if A is AdaptAction.TERMINATE_NODE then
25:             Temp_NT ← Temp_NT ∪ r.source.ref
26:         end if
27:     end for
28: end while
29: if A is AdaptAction.START_NODE then
30:     T_NT ← S_NT ∪ Min_NT
31: else if A is AdaptAction.TERMINATE_NODE then
32:     T_NT ← S_NT \ Min_NT
33: end if
34: for each RelationshipTemplate r in T.RelationshipTemplates do
35:     if n_1, n_2 ∈ T_NT where n_1.id = r.source.ref ∧ n_2.id = r.target.ref then
36:         T_RT ← T_RT ∪ rel.id
37:     end if
38: end for
39: return S_NT, S_RT, T_NT, T_RT
```

to incorporate delta deployment into our existing workflow, it is required that the runtime system allows the management of application instances on a Node Instance-based level. Therefore, similar to the creation of build and transformation plans, the runtime system is capable of dynamically generating workflow models, referred to as *adaptation plans*, that comprise the needed deployment steps in order to transition between two subsets of an application model. Each subset denotes the Node and Relationship Templates that should be considered during operation. Consequently, we need to calculate and provide a source and a target subset to the runtime system based on the users' actions. The source subset reflects the status of the currently deployed application instance and the target subset indicate the status we want to eventually achieve. Algorithm 5.2 computes both of these subsets when given the identifier of a Node Template and its designated target state, i.e. *started* or *deleted*. At the beginning of the algorithm, we first compute the source subset by filtering for all Node Templates that are already started, and consequently, add all Relationship Templates that are connected to them (see lines 2-11). Afterward, we recursively traverse through the topology to determine the minimum set of Node Templates that need to be started/terminated in order to reach the target state of the given Node Template (see lines 12-28). In our case, we assume that there are no cycles within the Topology Template of the model. The remainder of the algorithm deals with the composition of the target subset based on the result of the while loop and the desired target state. After the creation of these adaptation plans, we can trigger its execution and provide users with information about its progress and result within the graphical editor. Similar to transforming instances, we omit any further details about the creation of these adaptation plans.

The continuous synchronization between models and deployed instances gives users immediate feedback during modeling time. Therefore, users can quickly react to deployment or configuration errors without having to exit the modeling system. Verification checks that go beyond the deployment of instances, such as manual testing, can still be performed by interacting with the instance within its respective target environment. While enabling deployments during modeling time remains an essential aspect throughout this concept, it is equally important to visually incorporate these new capabilities in a clear and cohesive way. Therefore, an effective user interface has to be developed that not only provides users with valuable information but also feels intuitive and unobtrusive upon interaction. As mentioned in requirement **R4**, this includes state information about the current application instance, such as data about its various Node Instances, and any logs that are generated throughout the deployment process. Based on the provided data and visualization, users should be able to draw conclusions as to why and which parts of the application failed during its deployment. Moreover, in order to meet requirement **R2**, the user interface incorporates a means that allows users to perform basic management operations, i.e. starting and terminating the current instance.

Despite the proposed changes to the workflow, both the modeling system and runtime system remain autonomous, and therefore, can be used independently of one another. However, the capability of deploying application models during modeling time does not necessarily render the runtime system useless. On the contrary, the runtime system remains essential in other scenarios. For instance, services that compose multiple application models still depend on the conventional approach since live-modeling only allows the deployment of one instance at a time. Moreover, end-users, who mostly provision cloud resources rather than modeling, also do not benefit from this concept. They still rely on the runtime system in order to deploy their cloud applications in an automated manner.

| | | |
|---|---|---|
| **manual** | non-blocking UI during reconfiguration & manual change propagation | blocking UI during reconfiguration & manual change propagation |
| **automatic** | non-blocking UI during reconfiguration & automatic change propagation | blocking UI during reconfiguration & automatic change propagation |
| | **non-blocking** | **blocking** |

**Table 5.1:** Matrix illustrating the different combinations between the two live-modeling aspects

### 5.3.1 Change Propagation and Synchronization

Another aspect of reconfiguring instances focuses on determining the exact interval these changes should be propagated at. For the implementation of our concept, we consider two different approaches: *manual* and *automatic*. A manual change propagation means that it is the users' responsibility to decide when to exactly propagate any changes to the runtime system. This can be done, for instance, through the simple click of a button within the modeling system. In contrast, an automatic change propagation implies that the modeling system has the capability to automatically detect any changes within the topology of an application and immediately propagate any occurring changes to the runtime system. While this approach does not rely on any manual user input, it poses the risk of flooding the runtime system with a large number of reconfiguration requests that could lead to either long wait times or synchronization issues between the modeling and runtime system. Independently of the selected approach, the runtime system would then perform these reconfiguration requests in a transactional manner, similar to database transactions that fulfill the ACID properties [GR92]. Apart from deciding when to propagate these changes, we also have to take into consideration on how to present the information to the users. Analogous to before, there also exist two different approaches: *blocking* and *non-blocking*. Hereby the term blocking refers to the possible actions users can take during the periods whenever the modeling system notifies the runtime system of any changes. For instance, the blocking approach would prohibit any modeling activities from the users until the reconfiguration of the previous loop iteration is fully completed. By preventing any changes to the model, we can ensure that after each reconfiguration the running instance always reflects an up-to-date representation of the current model state. However, long reconfiguration times can quickly lead to an unsatisfied user experience due to the wait times users have to endure whenever they update and propagate the changes of the model. Conversely, a non-blocking approach resolves this issue by neglecting any sort of restrictions towards the users' modeling actions. Thus, users can continue editing the application model even during the reconfiguration step, allowing for a more responsive user interaction. This can be especially useful for cases where deployments take additional time because of the slowness of some contained components, such as the provisioning of virtual machines. In such cases, users can then continue modeling while the reconfiguration is performed in the background. Both presented aspects play an important role when it comes to the design of the user experience during the modeling process. Table 5.1 depicts the various implications when combining both aspects together. For our concept,

we decide to opt for a manual and non-blocking approach. This approach enables us to maintain a responsive user interaction while also giving the users full control over when to propagate any occurring changes. In order to further support the users, the modeling system keeps checking the current model for any changes since the last deployment or reconfiguration, and consequently, notifies the users whenever the model and running instance are out of sync. This reduces the potential flooding of reconfiguration requests, and more importantly, makes it transparent to the users when reconfigurations can and should be performed.

## 5.4 Validation System

Live-modeling enables users to directly deploy instances of their modeled application from within the graphical editor of the modeling system. This helps users to quickly detect errors that occur during the deployment of an instance. A deployment error can stem from various issues, such as a missing component within the model, using the wrong Relationship Type, or entering invalid input parameters. An additional reason an error may occur is due to the misconfiguration of a component within the application's model. Here, the term misconfiguration refers to an incorrect value assignment of a component's property during modeling time. Novice users, who lack deep knowledge about the internal specification of components, are especially prone to running into these issues. They are often not aware of these misconfigurations until the deployment eventually fails and is reported to the users. These errors can result in a reduction of productivity when the users are unable to correct problems until deployment is done. The time for deployments to finish often depends on the application's complexity. For instance, it can take up to several minutes for our example application to be fully deployed and functioning. As a result, users have to wait for the whole deployment just to find out if their models contain any misconfigurations or not, leading to slow development progress throughout the modeling process. A solution for this problem is to assist users at creating deployable application models, i.e. models that do not contain any misconfigurations. This can be executed by detecting misconfigurations and notifying users early on during modeling which prevents the initiation of deployment that would have failed regardless. Thus, this process helps speed up development progress by avoiding unnecessary waiting times.

As mentioned in Section 2.3.3, type architects are responsible for predefining Node and Relationship Types in order to provide them to application architects later on. The application architects then use these definitions to model the topologies of their application. While it is possible that both of these roles are performed by the same person, it is also conventional that they are performed exclusively by two different parties. Type architects have profound knowledge and the technical expertise about the correct configurations of these components. For instance, the *MyTinyToDoDockerContainer* Node Template in our example application contains a port property in its declaration that is used to determine which port the application should be accessible at. While this may seem trivial to some users, novice application architects might struggle to find a correct value for the port number without the assistance of a more experienced user or any guidance from the modeling system. Frequently asked questions in this context could revolve around the valid range of ports, whether two applications can share the same port number, or the difference between the properties "Port" and "ContainerPort". It is important to note that property assignments are not only constrained by technical formalities, e.g. ports can only be assigned a value between 0 and 65535. In contrast, there are occasionally cases where properties are constrained by attributes that are highly specific to certain business use cases, e.g. only allow a certain range of port numbers to be exposed, and

**Listing 5.1** XML syntax for TOSCA Node Type [OAS13]

```
<NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
          abstract="yes|no"? final="yes|no"?>
  <Tags>
    <Tag name="xs:string" value="xs:string"/> +
  </Tags> ?
  <DerivedFrom typeRef="xs:QName"/> ?
  <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
  <RequirementDefinitions>
    <...>
  </RequirementDefinitions> ?
  <CapabilityDefinitions>
    <...>
  </CapabilityDefinitions>
  <InstanceStates>
    <InstanceState state="xs:anyURI"> +
  </InstanceStates> ?
  <Interfaces>
    <...>
  </Interfaces> ?
</NodeType>
```

therefore, require special care when assigned during modeling. Consequently, application architects, who lack this certain knowledge, would seek additional guidance from type architects whenever they encounter issues concerning the respective component. Since this hinders development progress, we intend to lower the unnecessary communication overhead between type and application architects. Hence, we propose an integrated validation system that enables type architects to formally enrich TOSCA-based type definitions with property constraints and help application architects to detect misconfigurations during modeling time. These constraints denote proper value assignment for each respective type property and are fully integrated within the modeling system's graphical editor, i.e. incorporate a means to check and actively assist users to create deployable application models as mentioned in requirement **R7**. This reduces the number of failed deployments, and therefore, accelerates the development progress when modeling TOSCA-based applications.

There are various approaches when it comes to defining property constraints. For instance, one could directly include such information into the description XML tag that is contained in the respective type definition. Application architects could then use this description as a reference manual when assigning values to properties. While this approach allows for high flexibility when defining property constraints, it may prove to be difficult to integrate this information into the graphical editor in order to properly validate the users' assignments given the lack of standardization and high variety. In addition, application architects are expected to be actively aware of this information to be able to take advantage of this approach. As a result, we strive for a more sophisticated and TOSCA supported approach that also provides a forthright way of working. Listing 5.1 shows a shortened version of the TOSCA XML syntax for defining a Node Type. According to its syntax, a Node Type's properties can be defined by an external XML element or complex type whose reference is stored as an attribute within the Node Type's *PropertiesDefinition* element. Listing 5.2 shows a Service Template that contains a sample schema for defining such a property. In this example, our *ApplicationProperties* property definition comprises three elements: Owner, InstanceName, and AccountID. Apart from their name, the definition also stores an appropriate data type for each of its

**Listing 5.2** TOSCA XML Service Template sample [OAS13]

```
1  <Definitions name="MyServiceTemplateDefinition"
2              targetNamespace="http://www.example.com/sample">
3    <Types>
4      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
5                 elementFormDefault="qualified"
6                 attributeFormDefault="unqualified">
7        <xs:element name="ApplicationProperties">
8          <xs:complexType>
9            <xs:sequence>
10             <xs:element name="Owner" type="xs:string"/>
11             <xs:element name="InstanceName" type="xs:string"/>
12             <xs:element name="AccountID" type="xs:string"/>
13           </xs:sequence>
14         </xs:complexType>
15       </xs:element>
16       <...>
17     </xs:schema>
18   </Types>
19
20   <ServiceTemplate id="MyServiceTemplate">
21     <Tags>
22       <Tag name="author" value="someone@example.com"/>
23     </Tags>
24     <TopologyTemplate id="SampleApplication">
25       <NodeTemplate id="MyApplication"
26                     name="My Application"
27                     nodeType="abc:Application">
28         <Properties>
29           <ApplicationProperties>
30             <Owner>Frank</Owner>
31             <InstanceName>Thomas' favorite application</InstanceName>
32           </ApplicationProperties>
33         </Properties>
34       </NodeTemplate>
35       <...>
36     </TopologyTemplate>
37     <...>
38   </ServiceTemplate>
39
40   <NodeType name="Application">
41     <documentation xml:lang="EN">
42       A reusable definition of a Node Type representing an
43       application that can be deployed on application servers.
44     </documentation>
45     <NodeTypeProperties element="ApplicationProperties"/>
46     <...>
47   </NodeType>
48 </Definitions>
```

**Listing 5.3** Proposed XML sample extension for TOSCA Node Type

```
1  <Types>
2    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3        elementFormDefault="qualified"
4        attributeFormDefault="unqualified">
5      <xs:element name="PropertiesName">
6        <xs:complexType>
7          <xs:sequence>
8            <xs:element name="PropertyName1" type="Type1" pattern="Pattern1"
                 description="Description1"/>
9            <xs:element name="PropertyName2" type="Type2" pattern="Pattern2"
                 description="Description2"/>
10           <xs:element name="PropertyName3" type="Type3" pattern="Pattern3"
                 description="Description3"/>
11         </xs:sequence>
12       </xs:complexType>
13     </xs:element>
14   </xs:schema>
15 </Types>
```

elements as an additional XML attribute. Therefore, a modeling system could validate against this defined data type and if necessary, warn the users if the value assignment does not comply with the provided type declaration. However, a simple type declaration is often not sufficient enough to cover more complicated scenarios. For such cases, the TOSCA specification proposes the use of *PropertyConstraints* to optionally define any type of constraint per property. The specification intently does not impose any restrictions on how to define these constraints and leaves it to the actual implementation on how to implement them. While this approach seems viable for our use case, it contains one drawback. The TOSCA specification specifies that *PropertyConstraints* elements can only be added to certain components, such as Service or Node Templates, but not to formal definitions like Node or Relationship Types. Due to this, *PropertyConstraints* are only applicable within the respective templates and must be defined by application architects and not type architects. However, this limitation is removed within TOSCA's newer specification draft, the TOSCA Simple Profile YAML v1.2 [RBL17]. Contrary to its predecessor, the TOSCA Simple Profile allows the definition of constraints on any property that is defined in a TOSCA Service Template or any of its contained entities, including Node and Relationship Types. For defining constraints on properties or parameters, the TOSCA Simple Profile provides a list of possible clauses that can be declared as operators within the respective entity definition. Table 5.2 lists all of these constraint clauses. However, because the TOSCA Simple Profile is not officially recognized as a standard yet, and the fact that our base implementation still operates on the XML standard of TOSCA, we cannot make use of this novel concept.

As a result, in order to enable defining property constraints within type definitions that are based on the XML TOSCA standard, we propose the inclusion of constraint information as additional XML attributes within the actual definition of the respective property. Listing 5.3 demonstrates the extension of properties that incorporate this concept. Besides specifying the name and data type, type architects can now also include a pattern and a description when defining the structure of a type property. While the pattern attribute is used to validate values on their correctness, the description attribute is intended to provide application architects with additional information when assigning values to the respective property. Within our concept, we decide to use regular expressions (regex)

| Operator | Type | Description |
|---|---|---|
| equal | scalar | Constrains a property or parameter to a value equal to ('=') the value declared. |
| greater_than | scalar | Constrains a property or parameter to a value greater than ('>') the value declared. |
| greater_or_equal | scalar | Constrains a property or parameter to a value greater than or equal to ('>=') the value declared. |
| less_than | scalar | Constrains a property or parameter to a value less than ('<') the value declared. |
| less_or_equal | scalar | Constrains a property or parameter to a value less than or equal to ('<=') the value declared. |
| in_range | dual scalar | Constrains a property or parameter to a value in range of (inclusive) the two values declared. |
| valid_values | list | Constrains a property or parameter to a value that is in the list of declared values. |
| length | scalar | Constrains the property or parameter to a value of a given length. |
| min_length | scalar | Constrains the property or parameter to a value to a minimum length. |
| max_length | scalar | Constrains the property or parameter to a value to a maximum length. |
| pattern | regex | Constrains the property or parameter to a value that is allowed by the provided regular expression. |
| schema | string | Constrains the property or parameter to a value that is allowed by the referenced schema. |

**Table 5.2:** TOSCA Simle Profile YAML v1.2 Constraint Operators [RBL17]

as pattern syntax in order to check a given value assignment on its validity. We choose regex because of its high flexibility and wide adoption recognition within the industry for defining search patterns [BC07]. In fact, it is theoretically possible to replicate all other operators, listed in Table 5.2, using only regex patterns. For instance, the regex pattern *^([1-9]|[1-8][0-9]|9[0-9])$* matches all non-zero leading numbers between 1 and 99. In addition, most programming languages or frameworks offer built-in functions that efficiently recognize strings matching a given regex. Thus, regular expressions represent a suitable solution for allowing type architects to clearly define property constraints within type definitions and for utilizing them to actively support application architects during modeling time. Figure 5.4 depicts this workflow scenario. Type architects create type definitions as usual, but now also have the option to include additional constraint information to each of the properties. After storing these definitions in the modeling system's repository, application architects can retrieve and utilize them to model the topology of their respective applications. While doing so, the graphical editor reads the attached constraint information and validates the value assignment for each property. The user interface of the graphical editor will then point out every property value assignment that violates its underlying constraint definition. This concept provides optimal support for modeling deployable cloud applications while minimizing the communication overhead between type and application architects. To signify users of any constraint violation, we

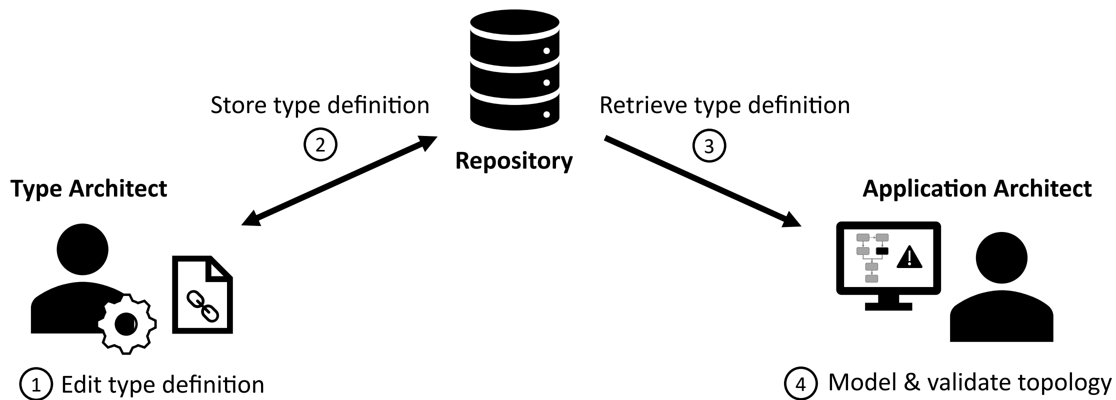



**Figure 5.4:** Workflow scenario for our input validation system

can resort to familiar user interface patterns that have been proven to be highly effective within the user experience domain [STOB12][SHB+14]. Figure 5.5 shows common approaches on how to inform users of incorrectly formatted input strings. Displaying constraint violations via user interface concepts that are common among other systems, such as web forms, ensures that users intuitively understand such misconfigurations.

While the use of the validation system remains completely optional, it can also be used in combination with any of the other features that reside within the modeling system. This also includes the solutions that we covered in Section 3.3. In contrast to the presented works that concentrate on the formal verification based on the structural correctness of the application's topology, our solution focuses on the semantic correctness of type properties. Therefore, both approaches can be combined and work in conjunction to further improve the tendency of creating deployable application models. Moreover, users can incorporate the validation system when working with the aforementioned live-modeling feature. When combined, users are now actively supported in creating deployable application models, and thus, reduce the risk of running into deployment errors that stem from misconfigurations. As a result, this leads to fewer wait times, and consequently, higher throughput when developing TOSCA-based applications.

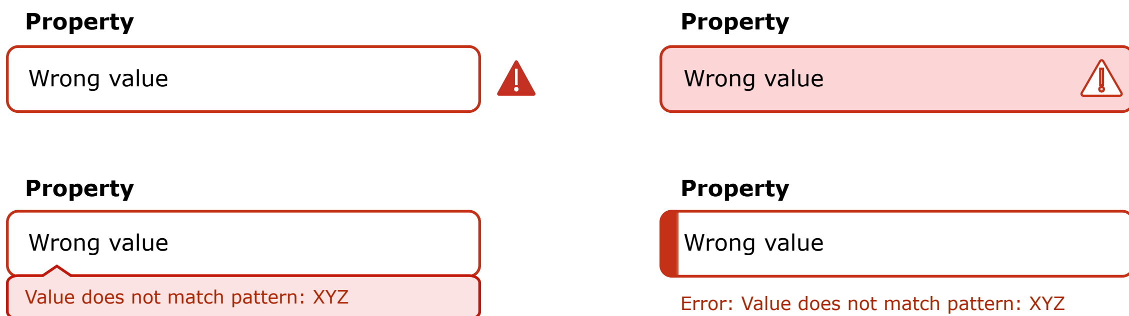



**Figure 5.5:** Common form validation approaches

## 5.5 Use Case Revisited

In order to demonstrate the use and the benefits of our presented concept, we revisit the use case example scenario from Section 4.3 and incorporate our alternative modeling workflow. As described earlier, our goal is to model an application whose topology is shown in Figure 4.2. Starting with a blank Service Template, we open the graphical editor of the modeling system in order to edit the topology of our application. Within the editor, we have access to all predefined type definitions and can instantiate Node and Relationship Templates via the drag-and-drop interface. We start by instantiating an OpenStack-Liberty-12 Node Template and assigning suitable values for each of its properties. Since this Node Type is used to access a running OpenStack instance in the cloud, most of its properties are related to how to access and authorize against it. These values can vary widely between different use cases, and thus, do not include any validation information. In order to test our current topology, we enable the live-modeling feature and provide a URL that points to an existing runtime system. The modeling system automatically packages and uploads our model to the runtime system and triggers its deployment. Because there is no running instance so far, the runtime system deploys a new instance from scratch. We observe the deployment progress from within the graphical editor and receive the result that our deployment was successful. This comes as no surprise since a single OpenStack Node Type does not entail any deployment logic, but rather is used to communicate with an OpenStack instance when provisioning computing resources, such as virtual machines. Therefore, we advance to the next iteration step by adding a Ubuntu-VM_18.04-w1 Node Template to the topology and connect it with the OpenStack Node Template through a hostedOn Relationship Template. As for validation data, this Node Type contains a regex pattern for the type property VMType that ensures that we only enter valid virtual machine types, also referred to as flavors, that are supported on OpenStack, such as *m1.small* or *m1.large*. By adding new components to the topology, the graphical editor recognizes these changes and notifies us that a reconfiguration is needed to incorporate them within the deployed instance. Since we did not edit the underlying OpenStack Node Template, the runtime system will use the previous deployed instance and transform it to match the updated model. However, if we make adjustments to previously deployed Node Templates, the runtime system will recognize this and redeploy the respective Node Instances. Because our runtime system is capable of interpreting declarative application models, it automatically derives all necessary deployment tasks in order to provision a Ubuntu virtual machine via the OpenStack instance. In comparison to the iteration step before, this deployment can take additional time due to the long boot times of virtual machines. However, due to the non-blocking UI of the graphical editor, we have the option to continue modeling our application instead of waiting for the deployment to finish. This enables us to save time and work more efficiently. When the deployment finishes, we again receive information about its result and perform actions accordingly. This process of making adjustments to the current topology and adding new components in an iterative way is repeated until the final model is reached. If we disable live-modeling at any point while an instance is currently running, the modeling system will automatically terminate it for us.

# 6 Implementation

This chapter covers the prototypical implementation of our presented concept within the OpenTOSCA ecosystem. It deals with the extension of OpenTOSCA's modeling system, Winery, in order to incorporate the live-modeling feature as well as the input validation system into the users' workflow. We highlight the changes towards the user interface of Winery's topology modeler and demonstrate the proper use of these new concepts. Moreover, we describe the means we have taken to facilitate the communication and data exchange between Winery's topology modeler and the OpenTOSCA Container. This includes, among others, the retrieval of instance information and how to properly visualize them to enable an intuitive way of working. We conclude this chapter by elaborating on the challenges and decisions we made throughout the development process.

## 6.1 OpenTOSCA Ecosystem

As mentioned in Section 2.4, the OpenTOSCA ecosystem consists of three main components: Winery, OpenTOSCA Container, and OpenTOSCA UI. Since Winery's primary purpose is the modeling and managing of TOSCA-based application models, we extend its capabilities to incorporate the features of our concept. The OpenTOSCA Container corresponds to the runtime system in our concept and will be used to deploy and manage application instances during modeling time. Because the OpenTOSCA UI is primarily geared towards end-users, and therefore, not used during the modeling process, we can neglect it for the remainder of this chapter. Although we cover both features individually, we want to emphasize that users can freely combine them with each other or also incorporate them within their existing workflow. Each section gives insight into the implementation work and encountered issues of the respective feature.

The development was performed on a Windows 10 64-bit computer that incorporates an i7-8585U Intel Core CPU and 16GB of RAM. At the time of this thesis, the development environment comprises the following software components: NodeJS 10.15.2, NPM 6.4.1, Java SE Development Kit 8u251, Angular CLI 8.3.15, IntelliJ IDEA Ultimate 2019.3, and Apache Maven 3.6.2. We use Google Chrome 80.0 and Firefox 76 to access and test our implemented features.

## 6.2 Live-Modeling

The following content outlines the changes to Winery that were made in order to incorporate the aforementioned live-modeling feature. These changes allow users to directly trigger and receive visual feedback about the deployment of their modeled application within Winery's topology modeler. It enables the simultaneous modeling and deploying of TOSCA-based applications while ensuring a seamless and insightful modeling experience.

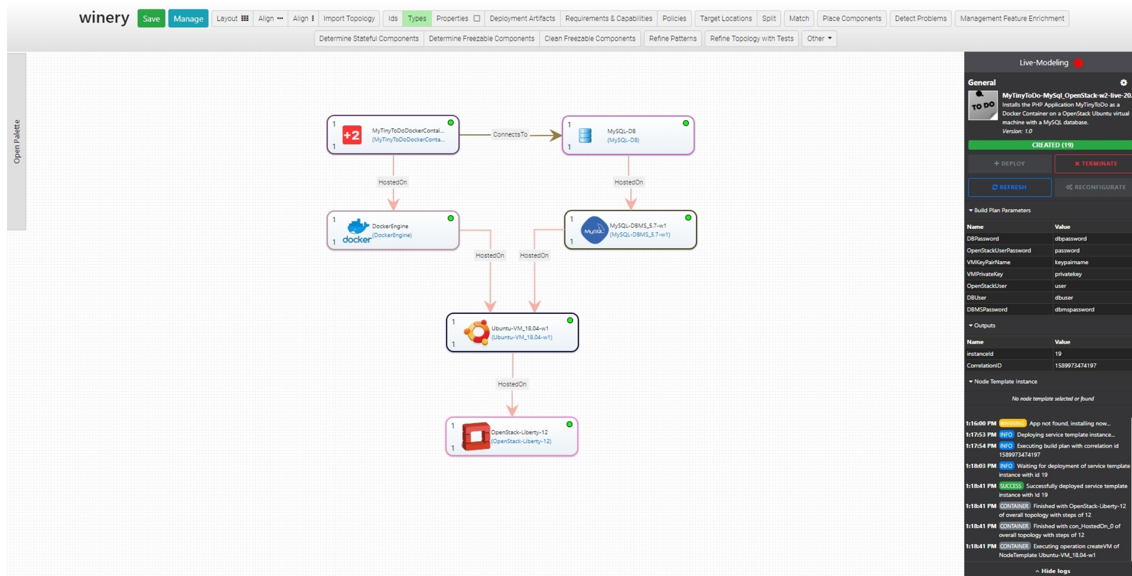



**Figure 6.1:** Screenshot of the enhanced topology modeler displaying our example application topology while utilizing the live-modeling feature

Figure 6.1 depicts the new user interface of the topology modeler when editing the Topology Template of our example application. When compared to the previous state, there are not any drastic changes made towards the user interface. In fact, the only change comprises an additional sidebar panel on the right side of the canvas panel. This ensures that existing users, who have been already familiar with the user interface before, are not disrupted by these new changes and can ignore them if not needed. Because adding another sidebar panel reduces the amount of space to work with, we added the option to pan within the canvas panel by holding the middle mouse button and dragging it towards different directions. Similar to the Node Template palette, clicking on the live-modeling sidebar will toggle an extended view that inhibits all live-modeling related actions and information in one dedicated place. Since live-modeling automatically deploys the current state of the application model, it is required to save any unsaved changes before utilizing this feature. When enabling live-modeling for the first time, the users are asked to provide a valid URL that points to a running OpenTOSCA Container instance. After checking its availability, the topology modeler starts the process for automatically uploading and deploying the application model. If the users unchecked the option to deploy an instance after model upload, we skip the actual application deployment, and instead, initialize a Service Template Instance with no running Node Instances.

In order to keep track of the deployment process and the state of the deployed instance, we incorporated an internal state machine within the topology modeler that dictates which actions a user can take at any given point throughout the modeling process. Figure 6.2 depicts the different states of our state machine. Double-circled states represent idle states, i.e. situations in which users interact with the user interface to perform various actions. For instance, the *Disabled* state is taken whenever the live-modeling feature is currently not in use. During this state, users can enable the live-modeling feature which subsequently causes the state machine to advance to the *Init* state. In contrast, single-circled states indicate transitioning states, i.e. they automatically transition to the next state after finishing their respective task (see dashed arrow connections in Figure 6.2). As an example, the *Deploy* transitional state is taken during the duration when deploying

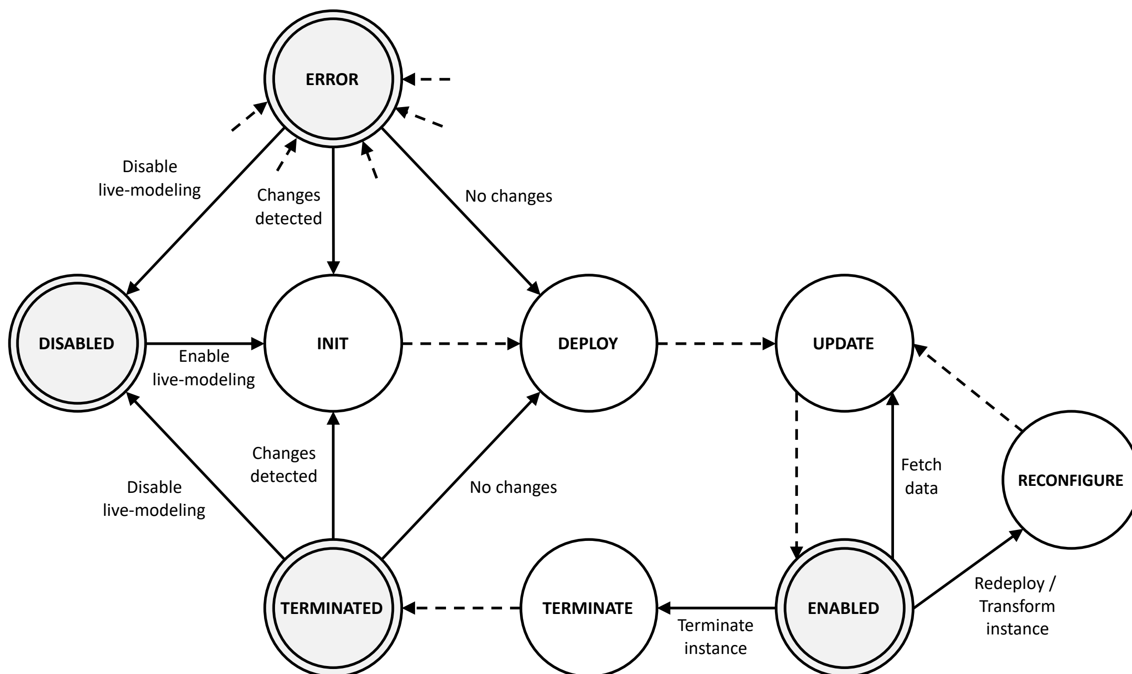



**Figure 6.2:** Internal state machine of topology modeler when utilizing live-modeling

an instance. When the deployment is finished, the state machine automatically transitions to the *Update* state to retrieve various instance information. With the exception of the *Init* state, each task is performed asynchronously in the background to ensure that users can continue to interact with the topology modeler while the respective process is still on-going. An animated indicator within the live-modeling sidebar informs the users about the current state and progress of the on-going process. Table 6.1 lists all available states with a short description of their purposes and the actions a user can take respectively. We included a universal *Error* state that is taken whenever an error occurs at any point in time. Based on the state that was taken when the error occurred we can narrow down the cause and present the users with suitable information as to why the action failed.

In order to fulfill requirement **R4**, we make use of OpenTOSCA Container's RESTful interface to establish the communication and data exchange between our modeling and runtime systems. This interface allows us to send standardized HTTP requests to receive access to various modeling- and instance-based resources and actions, such as uploading CSAR files, triggering deployments, or retrieving instance data. The OpenTOSCA UI, for example, is using this interface to enable end-users to easily deploy and manage application instances through the use of its graphical user interface. As a result, the OpenTOSCA Container is completely unaware of our live-modeling feature and, consequently, did not require any modifications during our development process. While most of the HTTP requests entail a synchronous response from the OpenTOSCA container, there are certain actions that can be requested through HTTP requests but do not deliver an immediate response about its result, e.g. triggering deployments. For such cases, we incorporated a polling mechanism that regularly checks the status of the respective action in a given interval and asynchronously notifies the modeling system about any occurring changes. If there is no change within a specific time span, the polling mechanism times out and throws an error message. Users can tweak the interval and time span through accessing the settings modal of the live-modeling sidebar.

| State | Description | User Actions |
|---|---|---|
| Disabled | Default state | Enable live-modeling. |
| Init | Create and upload live-modeling template | - |
| Deploy | Trigger and monitor deployment of instance | - |
| Update | Fetch current CSAR and instance data | - |
| Enabled | Idle state | Manually trigger data update. Reconfigure or terminate instance. |
| Reconfigure | Trigger and monitor redeployment/transformation/adaptation of instance | - |
| Terminate | Terminate instance | - |
| Terminated | Idle state | Deploy new instance. Transition to Init state if topology changed. |
| Error | Universal idle state after error occurred | Deploy new instance. Transition to Init state if topology changed. |

**Table 6.1:** Description of all available states within the live-modeling state machine

Since Winery and the OpenTOSCA Container are considered to be two separate systems with their own individual repositories, it is mandatory to first upload an application model before being able to deploy an instance. After the users provide a valid URL to a running OpenTOSCA container instance, we handle the initial model upload during the *Init* state. Figure 6.3 depicts this workflow. In this context, it is important to note that OpenTOSCA Container does not allow the simultaneous presence of two CSAR files that share the same identifier. In order to avoid potential conflict issues, we implemented a cloning feature within Winery's backend that duplicates an existing Service Template with the only exception of providing it with a new and unique CSAR identifier, hence preventing any conflicts when uploading to the OpenTOSCA Container. Due to Winery's capability of exporting Service Templates as CSAR files, we are able to package the current modeled application from within the topology modeler and directly upload it to the OpenTOSCA Container whenever needed. However, instead of directly uploading a CSAR file from the file system, we send an HTTP POST request with a payload that contains a URL pointing to the respective resource within Winery's backend. Because OpenTOSCA Container supports the declarative modeling approach, it automatically generates and stores an imperative build plan within its process engine so it can be used when triggering deployments later on. As a result, users do not have to deal with creating complex workflow models but rather concentrate on editing the application's Topology Template instead. After finishing generating the build plan, the HTTP POST request is completed, and we are now able to deploy instances based on the provided application model.

When triggering deployments, the modeling system retrieves the build plan of the respective CSAR from the OpenTOSCA container and checks the presence of any input parameters within the build plan. Input parameters are either automatically included when dynamically generating the application's build plan during upload or can be manually added when assigning the keyword *get_input:* to any of the type properties. If there are any input parameters found, the topology modeler will ask the users to fill in values for each of them. Figure 6.4 illustrates the message exchanges between the users and both systems in this scenario. It especially denotes our polling mechanism for retrieving the current status of the deployment. Since the OpenTOSCA Container does not return an immediate response to our initial deployment request, we regularly fetch the state

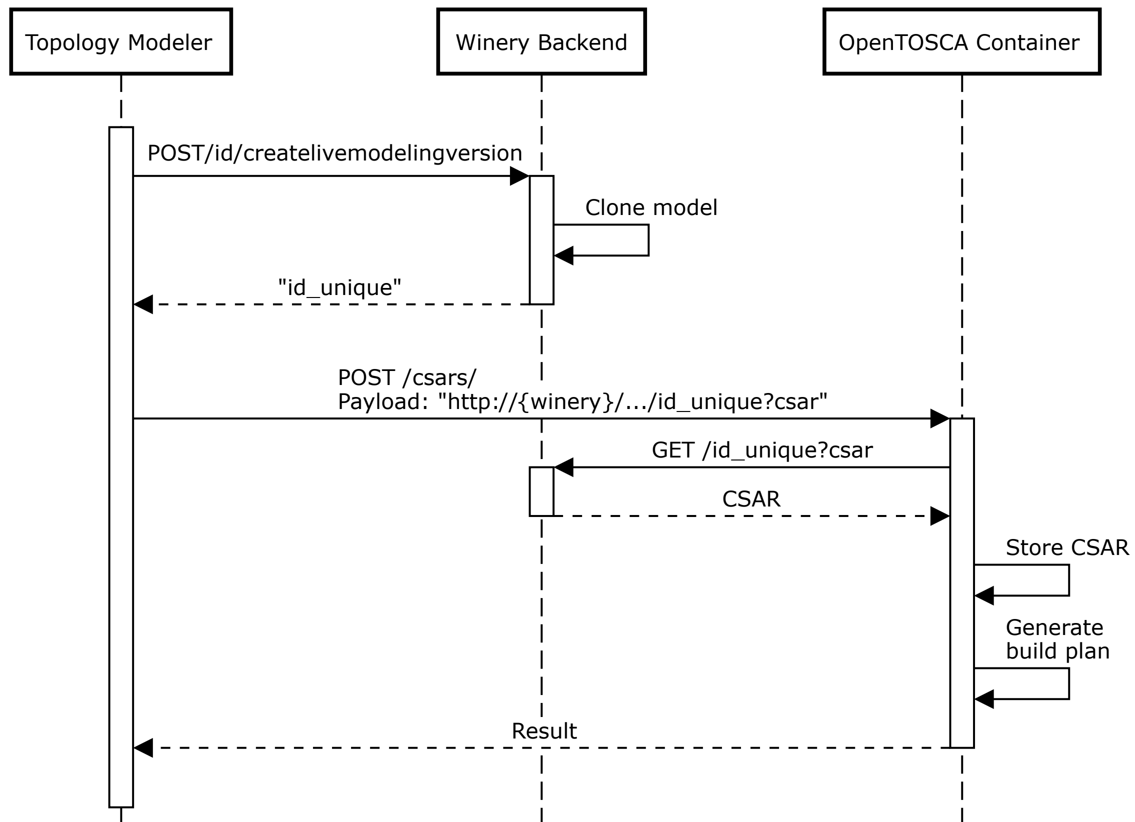



**Figure 6.3:** Sequence diagram illustrating the steps during the Init state

of the instance through its RESTful interface. If the deployment does not complete within a given time span, we notify the users and migrate to the *Error* state. Deployments are either automatically triggered after enabling the live-modeling feature for the first time or can be manually triggered later on using the corresponding button in the live-modeling panel. However, deploying the same model is not possible when the modeling system is already managing an instance. In this case, the users have to manually terminate it and trigger a new deployment afterward. This is also needed, whenever the users intend to trigger a deployment that should utilize different input parameters.

We implemented Algorithm 5.1 to keep track of the users' actions during the modeling process. However, to limit the scope of this thesis, we only checked for syntactical changes within the data object of the Topology Template instead of comparing each Node and Relationship Templates based on their semantic equality. By saving the previously deployed Topology Template in memory, we can compare it to the current one and notify users about its out-of-sync state and that a reconfiguration is needed in case the new changes should be considered within the deployed instance. During reconfiguration, users can decide to either redeploy or transform the deployed instance to match the current Topology Template. When choosing to redeploy, the topology modeler automatically triggers the termination plan of the currently deployed instance, and subsequently, performs the same steps as presented in Figure 6.3 and Figure 6.4 while taking the users' changes into account. In contrast, when choosing to transform, we utilize the capability of the OpenTOSCA Container to generate a transformation plan that comprises all necessary configuration tasks that need to be executed in order to transform a deployed instance to new instances that match the model of the

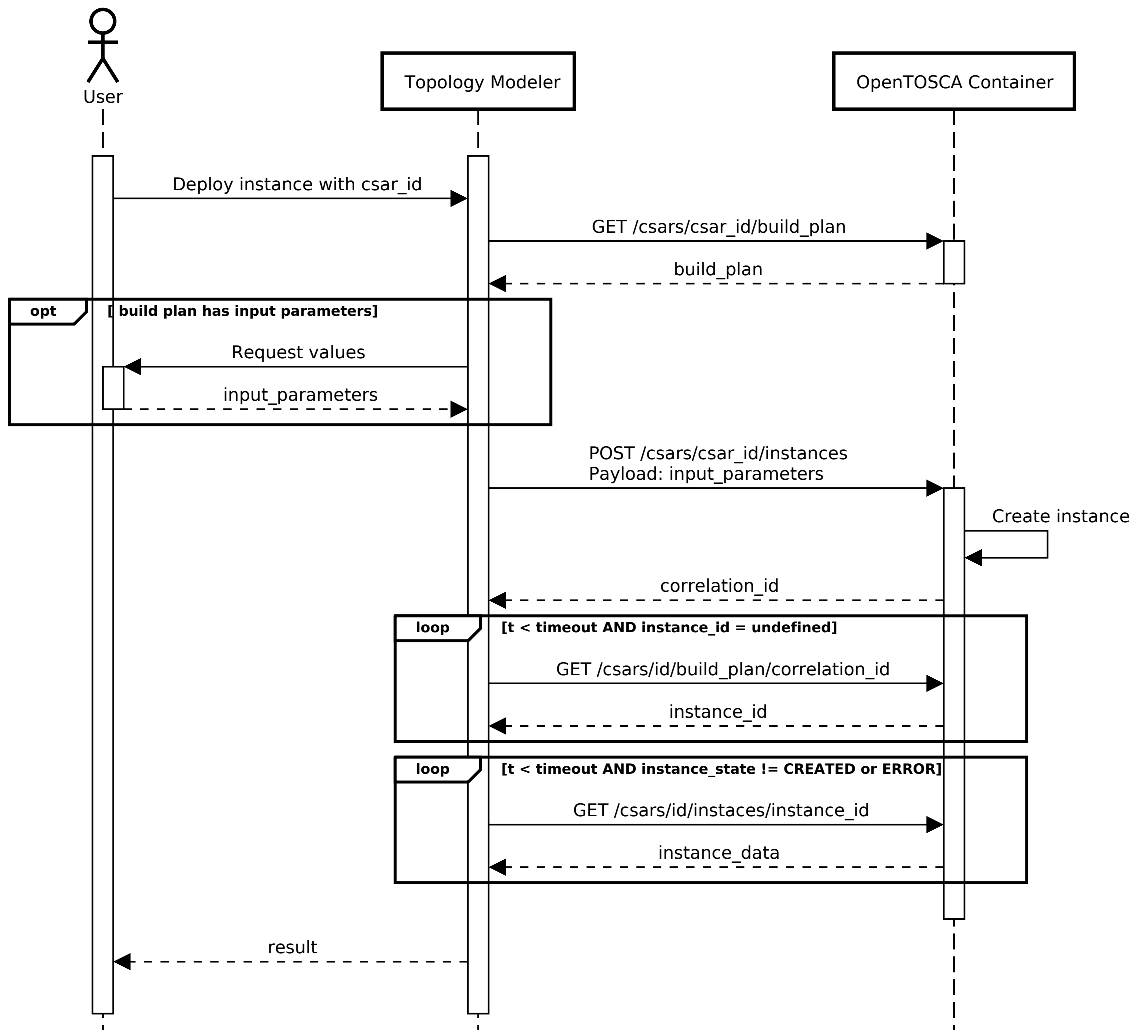



**Figure 6.4:** Sequence diagram illustrating the steps during the Deploy state

target CSAR. Figure 6.5 illustrates the workflow scenario when transforming an instance during the modeling process. Therefore, the topology modeler sends an HTTP request containing the source and target identifiers of the two CSAR files that should be considered during this operation. The OpenTOSCA Container appends this transformation plan as a generic management plan to every deployed instance that is running under the given source CSAR file. We trigger this management plan through a subsequent HTTP request and, if present in the transformation plan, ask the users to provide suitable values for each input parameter. Similar to regular deployments, the workflow incorporates a polling mechanism for monitoring the transformation process and notifying the users of any occurring changes.

In order to start/stop specific Node Instances, and consequently, perform *delta deployments*, users can select the respective Node Template that they intend to deploy/undeploy and click the start/stop button within the live-modeling panel. Figure 6.6 shows the steps when triggering a delta deployment that incorporates the start of a specific Node Instance. We first compute the source and target subset by utilizing Algorithm 5.2 and request the OpenTOSCA Container to create a suitable

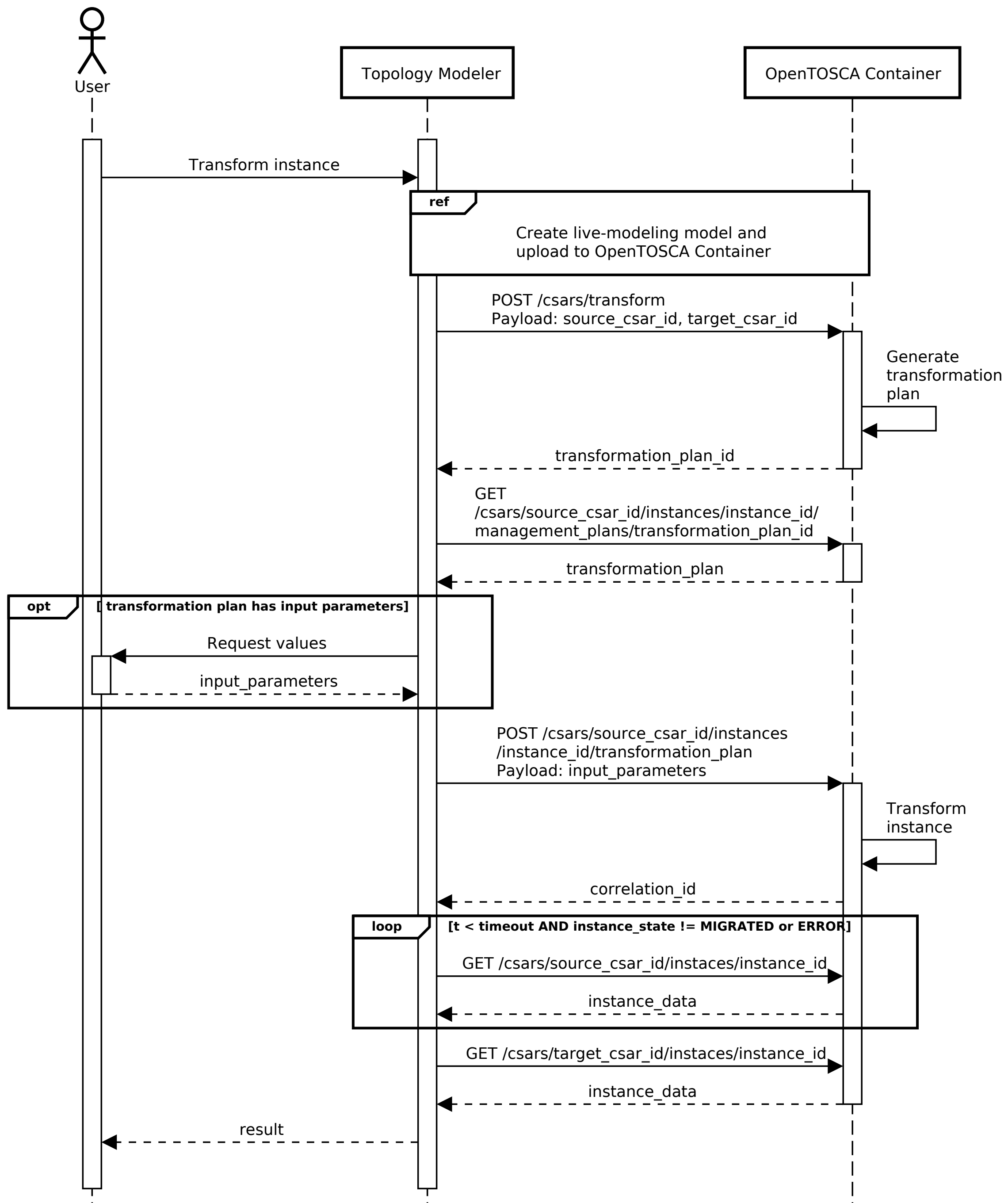



**Figure 6.5:** Sequence diagram illustrating the steps when transforming an instance

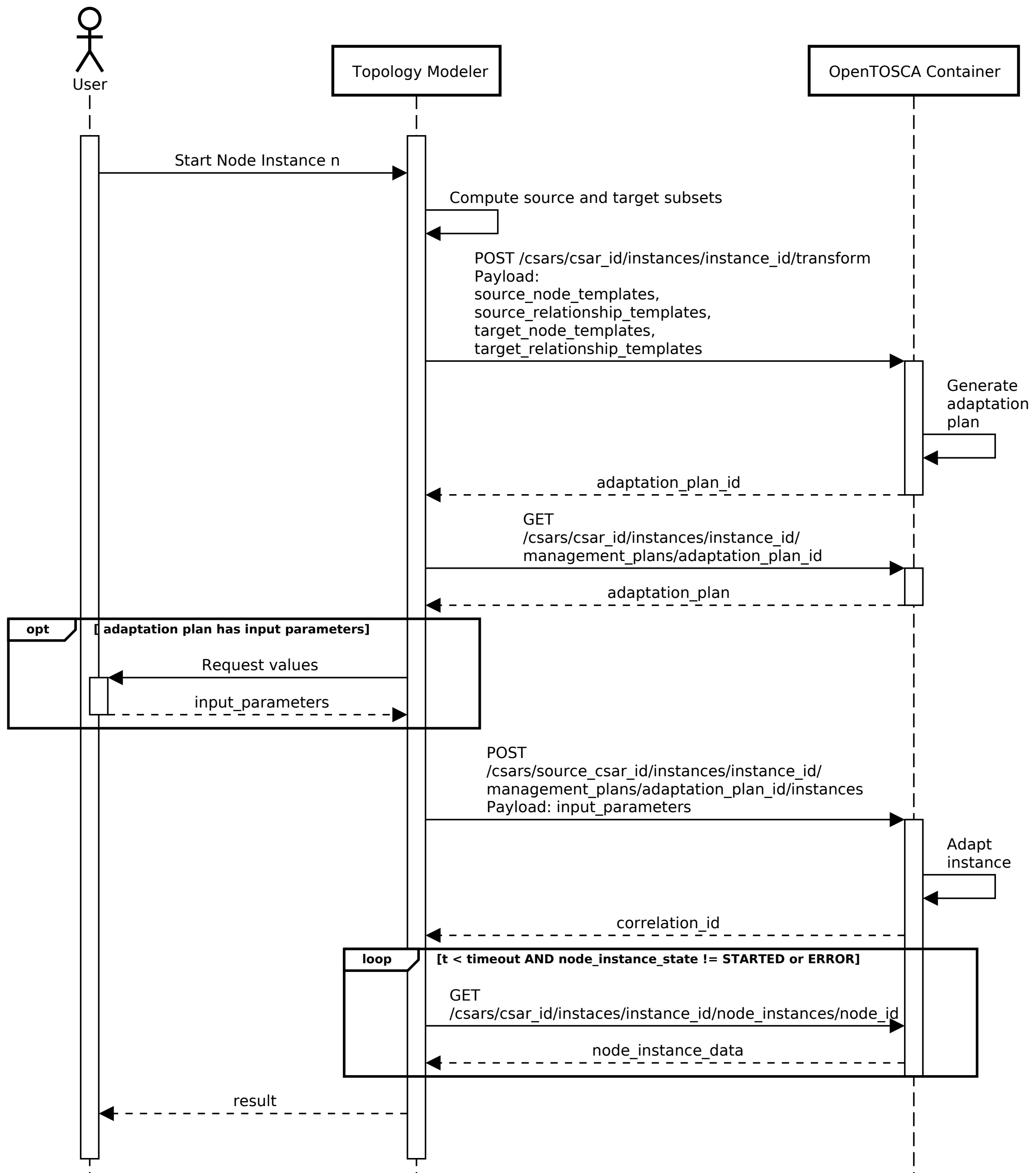



**Figure 6.6:** Sequence diagram illustrating the steps when adapting an instance

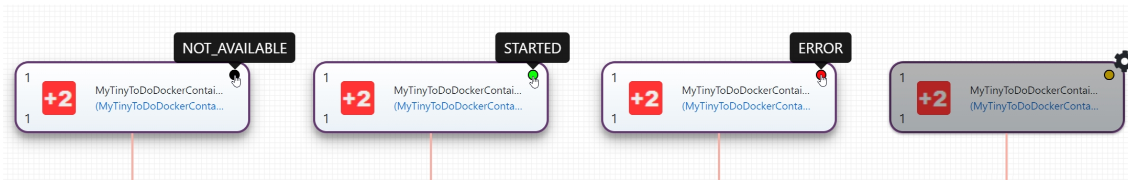



**Figure 6.7:** Node Template visualizations indicating the state of the corresponding Node Instance

adaptation plan based on our given subsets. Afterward, we trigger the adaptation plan and monitor its progress and result during operation. Since delta deployments do not require the upload of another application model, the reconfiguration takes significantly less time to complete.

As illustrated in Figure 6.2, users can trigger a new deployment during either the *Error* or *Terminated* state. This is usual, for example, after users have fixed misconfigurations that caused the initial error or intend to provide different input parameters. Based on the current Topology Template, there are two approaches to handling this new deployment. Similar to the *reconfigure* workflow, if there are any changes detected, the state machine transitions to the *init* state and initiates the creation and upload of a new live-modeling model before deployment. However, if there are no changes, we can directly trigger the deployment using the previously uploaded CSAR.

During live-modeling, the user interface provides users with regular visual feedback concerning the current state and data of the deployment process. In detail, each Node Template now visually incorporates feedback about the state of its Node Instance counterpart within the current deployment. Figure 6.7 illustrates the different states of this visualization. As depicted, a darkened Node Template with an animated gear icon signifies that the deployment of its corresponding Node Instance is still in process. Contrary, a colored circle within the Node Template symbolizes the resulting state of the Node Instance's deployment. Thus, users are able to immediately recognize which part of the application's topology failed during its deployment. Apart from the topology graph, the live-modeling panel offers access to various commands and information that are relevant during operation (see the right side in Figure 6.1). At the top, it shows various information about the current uploaded CSAR, such as its identifier, author information, description, and version number. In order to manage the deployed instance throughout its life cycle, the user interface shows the current state of the Service Template instance and also contains various buttons that trigger different actions by sending the respective HTTP request to the OpenTOSCA Container. For instance, users can trigger or terminate a deployment by clicking the respective buttons. The *refresh* button can be used to manually fetch current instance information and update all related user interface components. If the current Topology Template differs from the previously deployed Topology Template, users can use the *reconfigure* button to either redeploy or transform the deployed instance. The remainder of the panel contains various information that is specifically tied to the respective deployment. For instance, two tables are included to show the input as well as output parameters of the build plan. Moreover, when clicking on a Node Template, the live-modeling panel shows various information about the respective Node Instance that corresponds to the selected Node Template. At last, users have access to the logs that are generated throughout the deployment process. This helps at finding potential misconfigurations or giving insight about the individual deployment tasks that are executed by the OpenTOSCA Container. Users can further customize the panel or tweak live-modeling related settings, such as timeout and interval values, by clicking the gear icon on the top right to access the settings window.

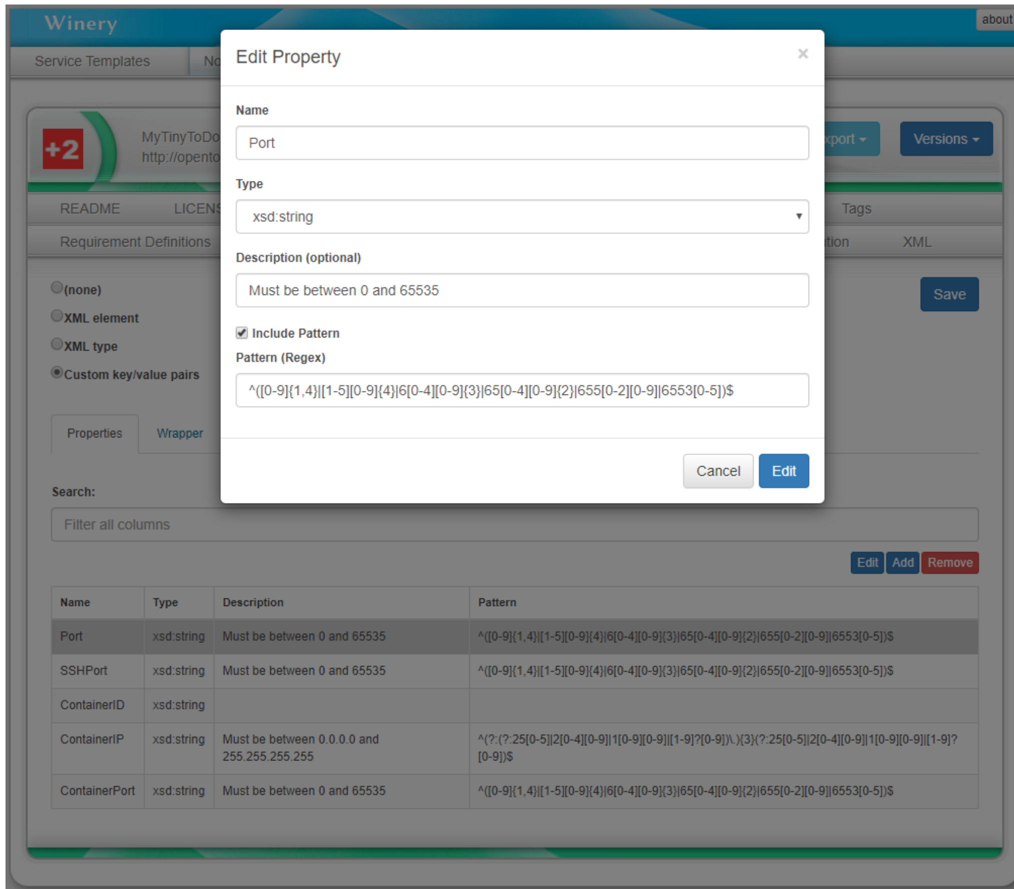



**Figure 6.8:** Screenshot of the TOSCA Management UI during the editing of a Node Type property

## 6.3 Input Validation System

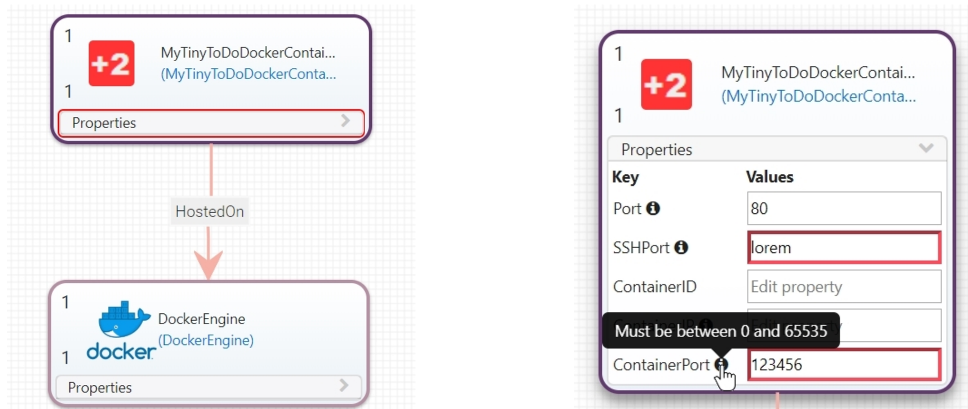
Winery's TOSCA Management UI can be used by type architects to create and edit TOSCA-conform type definitions through the use of its graphical user interface. Using a graphical user interface instead of manually writing these definitions saves time and prevents errors since these systems actively support users through different mechanics, e.g. reducing boilerplate code or automating repetitive tasks. In order to further help users during modeling, we also incorporate an input validation system into Winery. Figure 6.8 illustrates the tweaked user interface when editing the properties of an existing type definition. Type architects can now include a regex expression and a textual description to each property that is later used to assist application architects when using Winery's topology modeler. Listing 5.3 shows an excerpt of MyTinyToDoDockerContainer's type definition that incorporates this concept. In contrast to TOSCA's XML specification, Winery permits the use of a custom format to enable the declaration of type properties using a key-value notation. We extended this custom format by including two additional XML elements: *description* and *pattern*. While the *description* element can be included by type architects to provide users with more context to each property, the *pattern* element contains the actual regex pattern that is used during validation. It is important to note that our implementation does not check the regex pattern for its syntactic or semantic correctness. Therefore, it is the type architects' responsibility to ensure

**Listing 6.1** Extended XML definition for the Node Type MyTinyToDoDockerContainer

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <Definitions ...>
3      <NodeType name="MyTinyToDoDockerContainer" abstract="no" final="no"
           targetNamespace="http://opentosca.org/nodetypes" winery:bordercolor="#633a6f">
4          <winery:PropertiesDefinition elementname="Properties"
               namespace="http://opentosca.org/nodetypes/propertiesdefinition/winery">
5              <winery:properties>
6                  <winery:description>Must be between 0 and 65535</winery:description>
7                  <winery:key>Port</winery:key>
8                  <winery:pattern>
9                    ^([0-9]{1,4}|[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5])$
10                 </winery:pattern>
11                 <winery:type>xsd:string</winery:type>
12             </winery:properties>
13             <winery:properties>
14                 <winery:key>ContainerID</winery:key>
15                 <winery:type>xsd:string</winery:type>
16             </winery:properties>
17             <winery:properties>
18                 <winery:description>Must be between 0.0.0.0 and 255.255.255.255</winery:description>
19                 <winery:key>ContainerIP</winery:key>
20                 <winery:pattern>
21                   ^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}(?:25[0-5]|2[0-4][0-9]|1[0
22                   -9][0-9]|[1-9]?[0-9])$
23                 </winery:pattern>
24                 <winery:type>xsd:string</winery:type>
25             </winery:properties>
26             ...
27         </winery:PropertiesDefinition>
28         <DerivedFrom typeRef="nodetypes:DockerContainer"
               xmlns:nodetypes="http://opentosca.org/nodetypes"/>
29         <Interfaces>
30             ...
31         </Interfaces>
32     </NodeType>
33 </Definitions>
```

that the included pattern correctly reflects the intended meaning. In our example, the type definition holds two different patterns to ensure proper value assignments during modeling. The first pattern makes sure that the given port number lies within the valid range (see line 9 in Listing 6.1), i.e. lies between 0 and 65535. The second pattern guarantees that the given ContainerIP follows the notation of a valid IPv4 address (see line 21f. in Listing 6.1), i.e. lies between 0.0.0.0 and 255.255.255.255. Moreover, both properties also contain a human-readable description that offers more insight and reflects the intent of the respective regular expression.

When using Winery's topology modeler, users can enable this validation feature by toggling the checkbox next to the *Properties* button in the navigation bar (see the top in Figure 6.1). However, this feature can only be activated and used after toggling the *Properties* option. If enabled, the topology modeler continuously validates all assigned property values based on the regex pattern that is stored within each corresponding type definition. Any Node Template whose properties violate their respective regex pattern is marked by a red-colored border around its properties component




**(a)** Screenshot illustrating a misconfigured Node Template

**(b)** Properties component depicting misconfigurations and descriptions

**Figure 6.9:** User interface of the input validation system

(see Figure 6.9a). When unfolding the properties component, the users gain further insight into which properties caused this violation (see Figure 6.9b). The red border is automatically removed as soon as the provided value meets the regex pattern requirements. In addition, users can hover over the small information icon to access the description of the corresponding type definition. If a property's value contains the keyword *get_input* in the beginning, the topology modeler will omit the validation step since these properties are assigned during deployment time.

As mentioned in Section 5.4, our input validation system can be used in conjunction with the live-modeling feature. Since it is likely that users deploy more regularly during live-modeling, it becomes even more important to check for any misconfigurations within the Topology Template so failing deployments can be avoided early on. As a result, when both features are enabled, the topology modeler shows a warning message every time the users try to upload a model that contains any misconfigurations (see Figure 6.10). However, this message only functions as a general remark and users still have the option to advance at their own risk.

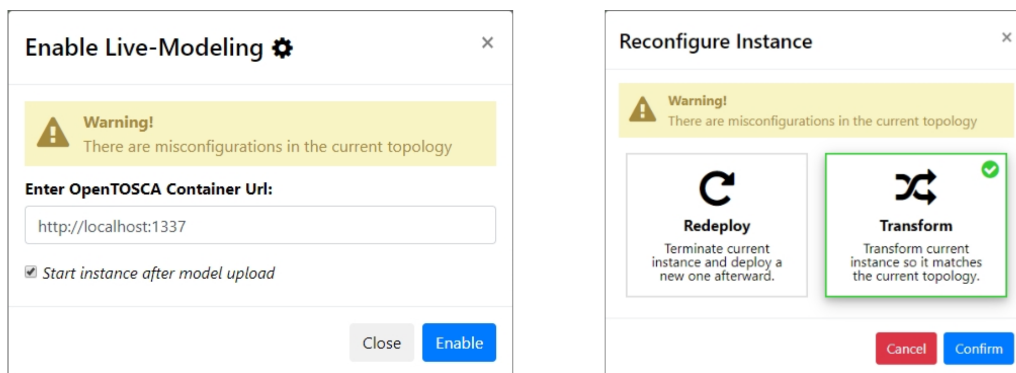



**Figure 6.10:** Warning messages notifying the users of misconfigurations in the topology

## 6.4 Remarks

Before concluding our work, we want to touch upon different aspects and limitations that we encountered during the development of our prototypical implementation. Due to time constraints and to limit the focus of this thesis, we were either unable to implement or neglected certain ideas that could, however, serve as potential starting points for further improvements. One aspect revolves around the actual deployment of computing resources. Since the focus of this thesis is on enabling deployments during modeling time rather than the actual resource provisioning, we decided to utilize OpenTOSCA Container's mocking feature throughout the entire development process. This feature ensures that all service invocations within the management bus are mimicked rather than invoking the actual implementation artifact behind every operation. Moreover, in order to minimize the number of modifications needed towards the OpenTOSCA Container and to continue to ensure the autonomy of the modeling and runtime systems, we had to make certain compromises in different areas. For instance, due to the architectural implementation of the OpenTOSCA Container API, we are currently restricted to the use of a polling mechanism in order to be notified of any changes concerning the status of the current deployment, i.e. progress information, plan logs, or instance data. Another compromise had to be made concerning the deployment of application models. Since the modeling and runtime systems do not share a common model repository, it is currently mandatory to always package and upload the application model to the runtime system whenever a deployment or reconfiguration is triggered. This can quickly over-flood the storage capacity and become especially time-consuming when dealing with larger applications that contain several components and go through multiple iteration cycles. Therefore, an alternative implementation needs to be developed that enables a more efficient and quicker way to provide application models to the runtime system. Resolving these issues will certainly help boost live-modeling's performance and also provide an improved user experience during its operation.

# 7 Conclusion and Outlook

Within the past decade, replacing physical hardware with virtual resources from the cloud became one of the most used methods in the industry of obtaining computing resources. Although cloud computing has vastly expanded the capacity of computing resources available, it has also led to new challenges in provisioning these resources. The heterogeneity of cloud providers has made it difficult to plan and model cloud applications without the risk of encountering vendor lock-ins. To resolve this issue, TOSCA was introduced as a standard to describe the topological structure and management operations of cloud applications in a vendor-neutral manner. Despite these advances, modeling and deploying applications remain a long and complex process due to the necessary knowledge and disjointed workflow.

In an effort to ease the modeling process, this thesis is centered around providing users an enhanced modeling system that also promotes proper configurations of applications. Our alternative approach, referred to as live-modeling, unifies the capabilities of both the modeling and runtime systems to create a more cohesive workflow during application development. Rather than relying on manual user input, our concept pushes the responsibility of managing deployments to the modeling system by incorporating a direct line of communication with the runtime system. Live-modeling enables users to effortlessly synchronize any changes within the application model with the corresponding running instance. Therefore, users are able to deploy their application models during modeling time and gain immediate feedback concerning the current state of the deployment. Additionally, in order to overcome knowledge deficits and support inexperienced users during modeling, we extended TOSCA's type definition syntax to include data constraints that encourage the correct configuration of components. The incorporation of our concept into existing modeling applications would help accelerate modeling time by creating a more streamlined process.

In order to validate the applicability of our concept, we implemented a working prototype within the OpenTOSCA ecosystem. In doing so, we extended the capabilities of Winery by facilitating a loose connection between its topology modeler and the OpenTOSCA Container runtime. Our implementation demonstrated that users are not only able to automatically deploy their application model from within the topology modeler, but also receive visual feedback about its status and progress. To further alleviate the modeling process, Winery has been extended to also incorporate an end-to-end solution for defining constraints and validating template properties on a type-based level using regular expressions. Our solution provides an alternative workflow for developing TOSCA-based applications when working with a graphical user interface. Driven by a visual feedback loop, live-modeling offers an efficient approach to simultaneously model and deploy applications. The option to visually receive deployment information accelerates the detection of misconfigurations, which effectively leads to shortened development times. Our contribution further enriches the toolset within the OpenTOSCA ecosystem by providing an effective method for developing vendor-agnostic cloud applications.

## Outlook

The contributions of this thesis offer numerous starting points for future extensions. As mentioned in Section 6.4, our implementation utilizes the mocking feature of the runtime system to simulate the provisioning of computing resources during operation. Consequently, the next step consists in finding out how our prototype performs in a real-life use case scenario. In particular, it would be interesting to see how novice, as well as experienced users, receive our adaptation when compared to the conventional modeling approach. Performing a suitable user study in a controlled environment will help in gathering valuable feedback and assess the applicability and usability of our implementation. Moreover, since we solely focused on one particular application model within this work, testing different Topology Templates will evaluate the robustness of our implementation.

In addition, instead of enabling the deployment of services from within the modeling system, an alternate solution could take an opposite approach that focuses on obtaining a visual representation of an already running instance. Not only does this help users to grasp a better overview of the overall state and structure of the instance but it also provides access to all capabilities that were introduced in this thesis. Users could, therefore, edit the topology of an instance via the topology modeler's user interface and by triggering a reconfiguration immediately reflect their changes during runtime. While our implementation enables the deployment, termination, and reconfiguration of instances, future work could additionally consider the execution of management operations during modeling time. Examples of such management operations can include the scaling of instances, the configuration of components, or the execution of script files. By incorporating this mechanism into the live-modeling feature, we can further enhance its range of application, and thus, provide more operational control to the users.

During development, we put a strong emphasis on maintaining the autonomy of the modeling and runtime systems. As shown in Section 6.4, while this ensures a clear separation of concern and less modification effort, it also limited us in various aspects. Consequently, integrating these two systems closer together offers new opportunities to improve the overall performance of our solution. For instance, instead of polling data in a given interval, an alternative could resort to push-based approaches such as WebSockets or long-polling to notify the modeling system of any changes within the deployment. This would reduce the number of exchanged HTTP messages and prevent the modeling system from timing out whenever deployments take longer than expected. Furthermore, introducing a shared repository for storing application models would solve the issue of permanently uploading an updated model whenever changes need to be propagated to the runtime system.

At last, a different extension to our concept could focus on the scope of our input validation system. Property constraints are currently limited to only consist of information that is well-known ahead of deployment time. Therefore, it is not possible to dynamically include any additional information that is, for instance, derived from an already running Node Instance. This includes information such as IP addresses, blocked port numbers, or any other custom data that is available during runtime. Extending the current input validation system to include such information within property constraints can increase its range of application and flexibility.

# Bibliography


[ABD+17] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, D. A. Tamburri. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 497–498. DOI: 10.1109/icse-c.2017.162 (cit. on p. 17).

[AWSCF20] *AWS CloudFomartion User Guide*. 2020. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/working-with-templates-cfn-designer-overview.html (cit. on p. 43).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. "A Systematic Review of Cloud Modeling Languages". In: *ACM Computing Surveys* 51.1, 2018, pp. 1–38. DOI: 10.1145/3150227 (cit. on pp. 18, 41).

[BBF09] G. Blair, N. Bencomo, R. France. "Models@ run.time". In: *Computer* 42, 2009, pp. 22–27. DOI: 10.1109/MC.2009.326 (cit. on p. 40).

[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*. 2013, pp. 692–695. DOI: 10.1007/978-3-642-45005-1_62 (cit. on pp. 18, 31, 34, 35).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. "Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA". In: *OTM 2012, Part I*. Vol. 7565. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2012, pp. 416–424. DOI: 10.1007/978-3-642-33606-5_25 (cit. on pp. 30, 33, 51).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA". In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, 2014, pp. 87–96. DOI: DOI10.1109/IC2E.2014.56 (cit. on pp. 35, 36).

[BBKL13] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. 2013, pp. 527–549. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on pp. 27, 29, 61).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Vinothek - A Self-Service Portal for TOSCA". In: *Proceedings of the 6th Central-European Workshop on Services and their Composition, ZEUS 2014, Potsdam, Germany, February 20-21, 2014*. Ed. by N. Herzberg, M. Kunze. Vol. 1140. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 69–72 (cit. on pp. 31, 36).

[BC07] M. Becchi, P. Crowley. “An improved algorithm to accelerate regular expression evaluation”. In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems - ANCS ’07*. ACM Press, 2007. DOI: 10.1145/1323548.1323573 (cit. on p. 71).

[BCS16] A. Brogi, P. Cifariello, J. Soldani. “DrACO: Discovering Available Cloud Offerings”. In: *Computer Science: Research and Development*, 2016. *[In Press]*. DOI: 10.1007/s00450-016-0332-5 (cit. on p. 47).

[BDS18] A. Brogi, A. Di Tommaso, J. Soldani. “Sommelier: A Tool for Validating TOSCA Application Topologies”. In: *Model-Driven Engineering and Software Development*. Ed. by L. F. Pires, S. Hammoudi, B. Selic. 2018, pp. 1–22. ISBN: 978-3-319-94764-8 (cit. on p. 45).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. “The OpenTOSCA Ecosystem – Concepts & Tools”. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016*, 2016, pp. 112–130. DOI: 10.5220/0007903201120130 (cit. on pp. 28, 31, 36).

[BKLW17] U. Breitenbücher, K. Képes, F. Leymann, M. Wurster. “Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?” In: *Proceedings of the 11$^{th}$ Advanced Summer, School on Service Oriented Computing*. IBM Research Division, 2017, pp. 18–27 (cit. on p. 24).

[BTN+14] A. Bergmayr, J. Troya Castilla, P. Neubauer, M. Wimmer, G. Kappel. “UML-based Cloud Application Modeling with Libraries, Profiles, and Templates”. In: *CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems*. CEUR-WS. 2014, pp. 56–65 (cit. on p. 42).

[CASG13] F. Caglar, K. An, S. Shekhar, A. Gokhale. “Model-driven performance estimation, deployment, and resource management for cloud-hosted services”. In: *Proceedings of the 2013 ACM workshop on Domain-specific modeling*. 2013, pp. 21–26. DOI: 10.1145/2541928.2541933 (cit. on p. 42).

[CCDR20] *Cloudify Composer Developer Reference*. 2020. URL: https://docs.cloudify.co/5.0.5/developer/composer/ (cit. on pp. 44, 45).

[CV16] W. Chareonsuk, W. Vatanawood. “Formal Verification of Cloud Orchestration Design with TOSCA and BPEL”. In: *2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. 2016, pp. 1–5. DOI: 10.1109/ECTICon.2016.7561358 (cit. on p. 46).

[DGL+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12 (cit. on p. 17).

[EAD17] F. M. A. Erich, C. Amrit, M. Daneva. “A qualitative study of DevOps usage in practice”. In: *Journal of Software: Evolution and Process* 29.6, 2017, e1885. DOI: 10.1002/smr.1885 (cit. on p. 22).

[EGHS16] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano. "DevOps". In: *IEEE Software* 33.03, 2016, pp. 94–100. ISSN: 1937-4194. DOI: 10.1109/MS.2016.68 (cit. on pp. 17, 22).

[FSR+14] N. Ferry, H. Song, A. Rossini, F. Chauvel, A. Solberg. "CloudMF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications". In: 2014. DOI: 10.1109/UCC.2014.36 (cit. on pp. 39–41).

[GR92] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902 (cit. on p. 66).

[Han12] Y. Han. "Cloud Computing: Case Studies and Total Cost of Ownership". In: *Information Technology and Libraries* 30, 2012. DOI: 10.6017/ital.v30i4.1871 (cit. on p. 22).

[HBBL+14] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann, et al. "Automatic Topology Completion of TOSCA-based Cloud Applications." In: *GI-Jahrestagung*. 2014, pp. 247–258 (cit. on p. 47).

[HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. "Pattern-based Deployment Models and Their Automatic Execution". In: *11$^{th}$ IEEE/ACM International Conference on Utility and Cloud Computing UCC 2018, 17–20 December 2018, Zurich, Switzerland*. IEEE Computer Society, 2018, pp. 41–52. DOI: 10.1109/UCC.2018.00013 (cit. on p. 47).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications". In: *Business Process Model and Notation*. Ed. by J. Mendling, M. Weidlich. 2012, pp. 38–52. ISBN: 978-3-642-33155-8. DOI: 10.1007/978-3-642-33155-8_4 (cit. on p. 33).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery – Modeling Tool for TOSCA-based Cloud Applications". In: *11$^{th}$ International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (cit. on pp. 18, 31).

[Ley09] F. Leymann. "Cloud Computing: The Next Revolution in IT". In: *Photogrammetic Week '09*. Wichmann Verlag, 2009, pp. 3–12 (cit. on p. 25).

[LFS+15] M. Lushpenko, N. Ferry, H. Song, F. Chauvel, A. Solberg. "Using Adaptation Plans to Control the Behavior of Models@Runtime". In: *MRT 2015:10th Int*. Ed. by N. Bencomo, S. Gotz, H. Song. Vol. 1474. Workshop on Models@run. time, co-located with MODELS 2015: 18th ACM/IEEE Int, 2015 (cit. on p. 40).

[LR99] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 1999. ISBN: 0130217530 (cit. on p. 25).

[LRH+13] F. Leymann, M. Rutkowski, A. Hohl, M. Waschke, P. Zhang, eds. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Jan. 31, 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html (cit. on p. 29).

[MBJ+09] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, A. Solberg. "Models@ Run.time to Support Dynamic Adaptation". In: *Computer* 42.10, 2009, pp. 44–51. ISSN: 0018-9162. DOI: 10.1109/MC.2009.327 (cit. on p. 40).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011 (cit. on pp. 17, 21).

[Mor16] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. 1st. O'Reilly Media, Inc., 2016. ISBN: 1491924357 (cit. on pp. 17, 22).

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*. Specification. Organization for the Advancement of Structured Information Standards (OASIS), 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (cit. on pp. 25, 27).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 18, 26, 27, 68, 69).

[OMG11] OMG. *Business Process Model and Notation (BPMN), Version 2.0*. Object Management Group, 2011. URL: http://www.omg.org/spec/BPMN/2.0 (cit. on pp. 25, 27).

[OPTO20] *OpenTOSCA Online Documentation*. 2020. URL: https://opentosca.github.io/ (cit. on p. 36).

[RBL17] M. Rutkowski, L. Boutier, C. Lauwers, eds. *TOSCA Simple Profile in YAML Version 1.2*. Aug. 31, 2017. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html (cit. on pp. 26, 44, 70, 71).

[RR07] L. Richardson, S. Ruby. *RESTful Web Services*. First. O'Reilly, 2007. ISBN: 9780596529260 (cit. on p. 34).

[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, 2017, pp. 247–258. ISBN: 978-989-758-243-1 (cit. on p. 47).

[SBKL18] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns". In: *Papers From the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC'18)*. IBM Research Division, 2018, pp. 43–53 (cit. on p. 46).

[SHB+14] M. Seckler, S. Heinz, J. A. Bargas-Avila, K. Opwis, A. N. Tuch. "Designing Usable Web Forms: Empirical Evaluation of Web Form Improvement Guidelines". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. Toronto, Ontario, Canada: Association for Computing Machinery, 2014, pp. 1275–1284. ISBN: 9781450324731. DOI: 10.1145/2556288.2557265 (cit. on p. 72).

[SMB17] D. Stahl, T. Martensson, J. Bosch. "Continuous practices and devops: beyond the buzz, what does it all mean?" In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2017. DOI: 10.1109/seaa.2017.8114695 (cit. on p. 22).

[SNP15] J. Smeds, K. Nybom, I. Porres. "DevOps: A Definition and Perceived Adoption Impediments". In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by C. Lassenius, T. Dingsøyr, M. Paasivaara. 2015, pp. 166–177. ISBN: 978-3-319-18612-2. DOI: 10.1007/978-3-319-18612-2_14 (cit. on p. 22).

[STOB12] M. Seckler, A. N. Tuch, K. Opwis, J. A. Bargas-Avila. "User-Friendly Locations of Error Messages in Web Forms: Put Them on the Right Side of the Erroneous Input Field". In: *Interact. Comput.* 24.3, 2012, pp. 107–118. ISSN: 0953-5438. DOI: 10.1016/j.intcom.2012.03.002 (cit. on p. 72).

[WBB+14a] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "Streamlining Cloud Management Automation by Unifying the Invocation of Scripts and Services Based on TOSCA". In: *Int. J. Organ. Collect. Intell.* 4.2, 2014, pp. 45–63. ISSN: 1947-9344. DOI: 10.4018/ijoci.2014040103 (cit. on p. 36).

[WBB+14b] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, M. Zimmermann. "Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA". In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. CLOSER 2014. Barcelona, Spain: SCITEPRESS - Science and Technology Publications, Lda, 2014, pp. 559–568. ISBN: 9789897580192. DOI: 10.5220/0004859005590568 (cit. on p. 36).

[WBKL18] M. Wurster, U. Breitenbücher, O. Kopp, F. Leymann. "Modeling and Automated Execution of Application Deployment Tests". In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, 2018, pp. 171–180. DOI: 10.1109/EDOC.2018.00030 (cit. on p. 47).

[WINE20] *Eclipse Winery Online Documentation*. 2020. URL: https://winery.readthedocs.io/en/latest/index.html (cit. on p. 32).

[XZ15] T. Xu, Y. Zhou. "Systems Approaches to Tackling Configuration Errors: A Survey". In: *ACM Comput. Surv.* 47.4, 2015. ISSN: 0360-0300. DOI: 10.1145/2791577 (cit. on p. 19).

[YOF15] H. Yoshida, K. Ogata, K. Futatsugi. "Formalization and Verification of Declarative Cloud Orchestration". In: *Formal Methods and Software Engineering*. Ed. by M. Butler, S. Conchon, F. Zaïdi. 2015, pp. 33–49. ISBN: 978-3-319-25423-4. DOI: 10.1007/978-3-319-25423-4_3 (cit. on p. 46).

All links were last followed on June 15, 2020.

**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