FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model-based Deployment and Provisioning of Applications to the Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

David Madner
Matrikelnummer 0926741

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel
Mitwirkung: Projektass. Dipl.-Ing. Alexander Bergmayr

Wien, 04.12.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Model-based Deployment and Provisioning of Applications to the Cloud

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

David Madner
Registration Number 0926741

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel
Assistance: Projektass. Dipl.-Ing. Alexander Bergmayr

Vienna, 04.12.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

David Madner
Kaiserbrunnstraße 68/6, 3021 Pressbaum

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank my advisor Alexander Bergmayr for his continuous support and helpful advices for writing this thesis. Especially, when I was on the other side of the world, we were faced with additional challenges, such as different time zones or a slow internet connection. Special thanks go to Prof. Gertrude Kappel for her additional feedback and for giving me the opportunity to use the institute's infrastructure during the master's thesis evaluation.

Furthermore, I want to thank my parents Helga and Johann for their mental and financial support throughout the whole duration of my studies. With your help and useful advices, you made my life in difficult times a lot easier.

Moreover, my thanks go to my girlfriend, who gave me motivation and was patient with me in moments, when I dedicated more time to my master's thesis than to her – especially during the time of finishing the thesis.

Finally, I want to thank my family, friends and study colleagues all over the world for being part of my life, especially Tobias, as my brother and volleyball buddy, Jakob, as an "all-rounder-mate" in various aspects, such as sports and traveling, and Gregor for his friendship through school and university time and for his unique talent of imitating people.

Abstract

Cloud computing had and still has a major impact on how applications are made accessible for the users. Due to the advantages cloud computing has, there is a demand to migrate applications to the cloud. Unfortunately there does not exist general guidelines how to define the required application execution environments and deployment requirements so that they can be interpreted by any arbitrary cloud provider.

In the last years, cloud providers came up with approaches to be able to describe cloud resources in form of an interpretable template. Just recently, in November 2013, OASIS published the open standard TOSCA [44], which aims to unite existing proprietary approaches and standardise them. Approaches following a declarative way of describing orchestrated cloud resources are quite recent and are extended frequently, as it is a promising possibility of illustrating complex dependencies and limitations of computing resources in a way that can be read by human beings as well.

This thesis firstly discusses model driven engineering and cloud computing separately and afterwards, how they can be combined. The main aim is to create a model that contains enough information about dependencies, limitations and application specific requirements, which can support the migration of the application to the cloud.

Furthermore, the master's thesis proposes a process, which is subdivided into two parts: Deployment and Provisioning. The first step is about creating UML models and refining them with UML extensions (classifiers, profiles and stereotypes), which consists out of cloud computing specific attributes. The second step converts the model into a template, by means of applying model to text transformations, in order to be interpretable and executable by cloud providers.

Existing solutions only address partial aspects of the whole problem, focusing on other objectives. One of the main goal of this thesis is the creation of a unified and model-based solution, whose processes and tools support the application modeler and make a (semi-)automatic execution of the deployment and provisioning of an application in the cloud possible.

Kurzfassung

Cloud-Computing hatte und hat noch immer einen großen Einfluss darauf, wie Applikationen Benutzern zur Verfügung gestellt werden. Aufgrund der überwiegenden Vorteile, die Cloud-Computing mit sich bringt, besteht ein großes Bestreben, Applikationen in die Cloud zu migrieren. Leider existieren derzeit keine allgemeinen Richtlinien, in welcher Form die benötigten Ausführungsumgebungen und die Erstellungsspezifikationen einer Applikation definiert werden sollen, damit diese von jedem x-beliebigen Cloud Betreiber verwendet werden können.

In den letzten Jahren wurde vermehrt der Fokus auf Ansätze gelegt, die es ermöglichen, Ressourcen in der Cloud in Form eines interpretierbaren Templates zu beschreiben. Erst letztes Jahr, im November 2013, wurde von OASIS ein offener Standard TOSCA [44] veröffentlicht, der versucht bereits existierende propitäre Ansätze zusammen zu fassen und zu vereinheitlichen. Die Entwicklung hin zu einer deklarativen Beschreibung von orchestrierten Ressourcen in der Cloud sind noch Neuland und werden ständig erweitert, da es sich um eine vielversprechende Möglichkeit handelt, komplexe Abhängigkeiten und Einschränkungen von Computerresourcen in einem für den Menschen lesbaren Format zu beschreiben.

Diese Arbeit beleuchtet zuerst die Welten von Model-Driven-Engineering und Cloudcomputing und wie diese beiden kombiniert werden können. Das Ziel besteht darin ein Modell zu erstellen, das ausreichend Information über Abhängigkeiten, Einschränkungen und applikationsspezifische Anforderungen beinhaltet, damit es bei der Migration der Applikation in die Cloud unterstützend verwendet werden kann.

Im Rahmen dieser Arbeit wird weiters ein Prozess beschrieben, der in zwei Schritte unterteilt ist: Erstellung und Bereitstellung. Im ersten Schritt werden UML Modelle erstellt und mit UML Erweiterungen (Typen, Profile und Stereotypen), welche Cloud-Computing spezifische Attribute beinhalten, verfeinert. Der zweite Schritt besteht darin, die Modelle mit Hilfe von Transformationen in Templates zu konvertieren, damit diese vom entsprechenden Cloud Betreiber interpretiert und exekutiert werden können.

Bestehende Ansätze adressieren nur Teilaspekte des Migrationsproblems und fokussieren sich auf Teillösungen. Einer der Hauptziele dieser Arbeit ist, die Erstellung einer vereinheitlichten und modell-basierenden Lösung, dessen Prozesse und Tools den Applikationsmodellierer unterstützen und einen (semi-)automatischen Ablauf der Erstellung und Bereitstellung einer Applikation in der Cloud ermöglichen.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Aim of Thesis . . . 3
  - 1.3.1 Motivating Example . . . 3
  - 1.3.2 Cloud-based Application Deployment . . . 4
  - 1.3.3 Provisioning to the cloud . . . 5
- 1.4 Approaches and Methodologies Applied . . . 6
- 1.5 Structure of Thesis . . . 7

2 Cloud Computing meets Model Driven Engineering **9**
- 2.1 Cloud Computing Principles . . . 9
  - 2.1.1 Cloud Computing Service Models . . . 10
  - 2.1.2 Potential Risks and Challenges When Moving to the Cloud . . . 11
- 2.2 Model Driven Engineering . . . 13
- 2.3 Model Driven Engineering for the Cloud . . . 15
  - 2.3.1 Unified Cloud APIs . . . 17
  - 2.3.2 Templates . . . 18
- 2.4 Building an Application Stack in the Cloud . . . 18
  - 2.4.1 Search Based Software Engineering . . . 20

3 Deployment and Provisioning Process **23**
- 3.1 The Model-based Deployment and Provisioning Process . . . 23
  - 3.1.1 The Cloud Metamodel (CMM) . . . 24
  - 3.1.2 Scaling Rules . . . 26
  - 3.1.3 Modelling Library and Extensions for Model Refinements . . . 28
  - 3.1.4 Model refinement . . . 30
  - 3.1.5 Model to Model transformation . . . 32
  - 3.1.6 Model to Text Transformation . . . 34
  - 3.1.7 Provisioning Engine . . . 34
- 3.2 UML Profile Enrichment . . . 34

4 Prototypical Implementation **37**

4.1 Used technologies . . . 37
4.2 Implementation of Deployment and Provisioning Process . . . 38
4.2.1 Deployment Process . . . 38
4.2.2 Provisioning Process . . . 40
4.3 Implementation of UML Profile Enrichment Process . . . 44
4.3.1 Solution . . . 44

5 Evaluation **47**
5.1 Blueprints . . . 47
5.2 Case study . . . 48
5.2.1 Cloud Providers Used For Evaluation . . . 49
5.2.2 Calender Application . . . 50
5.2.3 PetStore Application . . . 53
5.2.4 JBoss Ticket Monster . . . 59
5.3 Results of the Case Study . . . 61

6 Related Work **63**
6.1 Related technologies . . . 63
6.1.1 Template formats . . . 63
6.1.2 Unifying Cloud APIs . . . 64
6.1.3 Proprietary Approaches to Describe Deployment Requirements . . . 65
6.2 Similar approaches . . . 67
6.2.1 Managing the Configuration Complexity of Distributed Applications . . . 67
6.2.2 Uni4Cloud . . . 69
6.2.3 CloudMIG . . . 70
6.2.4 CloudMF . . . 72
6.2.5 Aeolus . . . 73

7 Conclusion **75**
7.1 Critical Reflection . . . 75
7.1.1 All-Embracing Solution . . . 75
7.1.2 Automation . . . 75
7.2 Current Limitations and Possible Improvements . . . 76
7.3 Potential Extensions . . . 77
7.3.1 Scaling rules for PaaS . . . 77
7.3.2 In-place Transformation for Requirements Matching . . . 77
7.3.3 TOSCA Integration . . . 77
7.3.4 Chef Opscode integration . . . 78
7.3.5 Docker . . . 79
7.3.6 CloudSim . . . 81
7.4 Lessons Learned . . . 81
7.4.1 Different Strategies of Virtual Image Configuration . . . 82
7.4.2 Getting Familiar with Offerings and Available Technologies . . . 82

A Cloud Metamodel **83**

B Scaling Metamodel **85**

List of Figures **86**

Listings **88**

Bibliography **89**

CHAPTER 1

Introduction

1.1 Motivation

Cloud computing offers new possibilities for IT companies, to provide services to their customers. There is no need to plan for peak loads in advance, as additional resources can be acquired instantly. Therefore, moving applications to so-called cloud providers, which operate data centres all over the world, are an appealing opportunity to save costs and minimize the expenses for own IT infrastructure.

In the last years cloud computing became really popular and following a recently published forecast analysis, the size of the cloud computing market will constantly increase in the next years. Currently the market size is said to be 153.6 Billion of Dollars worth, whereas in 2016 a market size of 206.6 Billion of Dollars is expected [3]. For that reason, more and more companies started their own business providing cloud computing facilities and are trying to get market share. Big advantages [5] for cloud consumers are the charging principle (pay-per-use) and the elasticity of cloud environments to automatically acquire resources (processing power, storage, bandwidth) or release them, depending on current circumstances such as work load or amount of requests. Those benefits of cloud computing can only be exploited, if the application is moved to the cloud.

The ARTIST project [10], works towards a model-based migration process to move existing applications to cloud environments. An excerpt of this process [9] is shown in Figure 1.1, whereas this master's thesis aims at supporting the last two phases: *Prepare Deployment* and *Execute Provisioning*.

The whole project follows a Model Driven Engineering (MDE) approach, so a model-centric solution for a deployment and provisioning process in this master's thesis suggests itself. When using a model-based approach during software development, models are not used only for documentation, but are major artefacts of the software development process. Deployment requirements, system constraints and cloud service dependencies of an application can be represented in a model on a higher level of abstraction and thus in a structured way. In doing so, the infor-

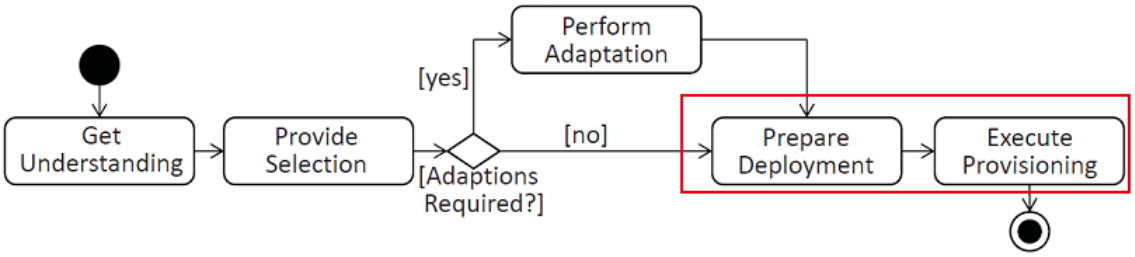

Figure 1.1: Context of master's thesis within ARTIST [10]

mation can be converted by means of model transformations into a format that is interpretable by a cloud provider, which can be used to support and automate the provisioning process.

As the title of this masters thesis mentions as well, the transition from a non-cloud to a cloud environment from the viewpoint of an application can be split into two parts: deployment and provisioning, which is also discussed by Eilam et al. in [23]. The deployment is about the modelling part that captures the necessary cloud-specific information in form of models. The provisioning is the subsequent step, which translates the enriched deployment model into a format that is interpretable by cloud providers and can help to automatise the process.

1.2 Problem Statement

There exist formats, such as CloudFormation from Amazon, which can be used to define cloud resources requirements that can be interpreted by cloud providers. Those scripts are hard to maintain and tedious when parts have to be adapted. When application deployment requirements are specified on a higher level of abstraction, only deployment relevant data are represented in those models and contribute to a better understanding. For this reason the first research question is: How can cloud application deployments be expressed in terms of models?

Shifting an application to the cloud causes new challenges to be solved. Depending on the service abstraction layer, a cloud computing provider is working on, there are specific problems, which have to be taken into account. In the case of *Infrastructure as a Service (Iaas)* especially the orchestration of multiple virtual machines, the structure of the required execution stacks and the intercommunication have to be analysed. When choosing a cloud provider who offers *Platform as a Service (PaaS)*, more adaptations have to be done to the application itself, as the provider operates on a higher abstraction layer and infrastructure-related properties or execution environments are predefined.

An application has specific constraints or dependencies on services or environments that are captured in a structured way in form of models. As cloud providers offer APIs to provision cloud resources, it suggests itself to automate the provisioning process as well. Thus, the second research question is as follows: How can the resource provisioning for cloud applications be automated?

The information of a model that expresses deployment requirements on a higher level of abstraction, can be used to automatically provision required cloud resources by using advanced MDE techniques [12] such as model transformations and useful conventions encoded by them.

Several existing approaches introduced meta languages or additional languages in order to describe application deployments [25, 47, 46]. The proposed solution should use approaches that are familiar to application modelers, such as diagrams from the UML standard. This standard is widely accepted and should be tested for its applicability in modelling and supporting the deployment process. Especially structural models like class diagrams, component diagrams and deployment diagrams should be considered. Hutchinson et al. [35, 36] discuss advantages and benefits when integrating UML in the software development process.

By following a MDE approach, models can be automatically transformed into formats that are interpretable by cloud providers, such as TOSCA[1]. This standard introduces a way to describe virtual appliances in a universal way, to ensure interoperability, to provision cloud resources across multiple cloud platforms easily which prevent vendor lock-ins. It should be analyzed, if these standards or similar approaches can be integrated into the deployment and provisioning process.

1.3 Aim of Thesis

This section describes by means of a motivating example the aim of this master's thesis and discusses the steps *Deployment* and *Provisioning* and how the solution looks like.

1.3.1 Motivating Example

For a better understanding, Figure 1.2 shows a model of the components and deployment topology of the Java PetStore application[2], which is a relatively simple e-commerce application that uses Java technologies such as JavaServerPages (JSP), Enterprise JavaBeans (EJB) and Java Message Service (JMS). It contains three packages: *Petstore Components*, *Petstore Deployment* and *Deployment Library*. *Petstore Components* exemplarily depicts parts of used classes and their relationships. The artifacts *PetstoreBusiness* and *PetstoreData* manifest those components. The second package *Deployment Library* is one of the results of this thesis. The library defines modeling concepts to express cloud-oriented deployment topologies at the type level and is imported by the third package *Petstore Deployment*, which contains the deployment requirements. This definitions are used to decide which and how many cloud resources have to be created during the provisioning process.

In this example both, the data store and the application container reside on the same virtual machine, which has requirements such as Debian as the operation system or a CPU power of 2.6 GHz. Furthermore, the container is meant to be a Java Enterprise Edition execution environment (for instance a JBoss web server).

The aim is to develop modeling facilities to let modelers express these dependencies and requirements in an abstract way to support the application modeler during the definition of de-

[1] `https://www.oasis-open.org/committees/tosca/`

[2] `http://www.oracle.com/technetwork/java/petstore1-3-1-02-139690.html`

ployment requirements for the application. Furthermore, this information can be converted into a format that supports the provisioning process.

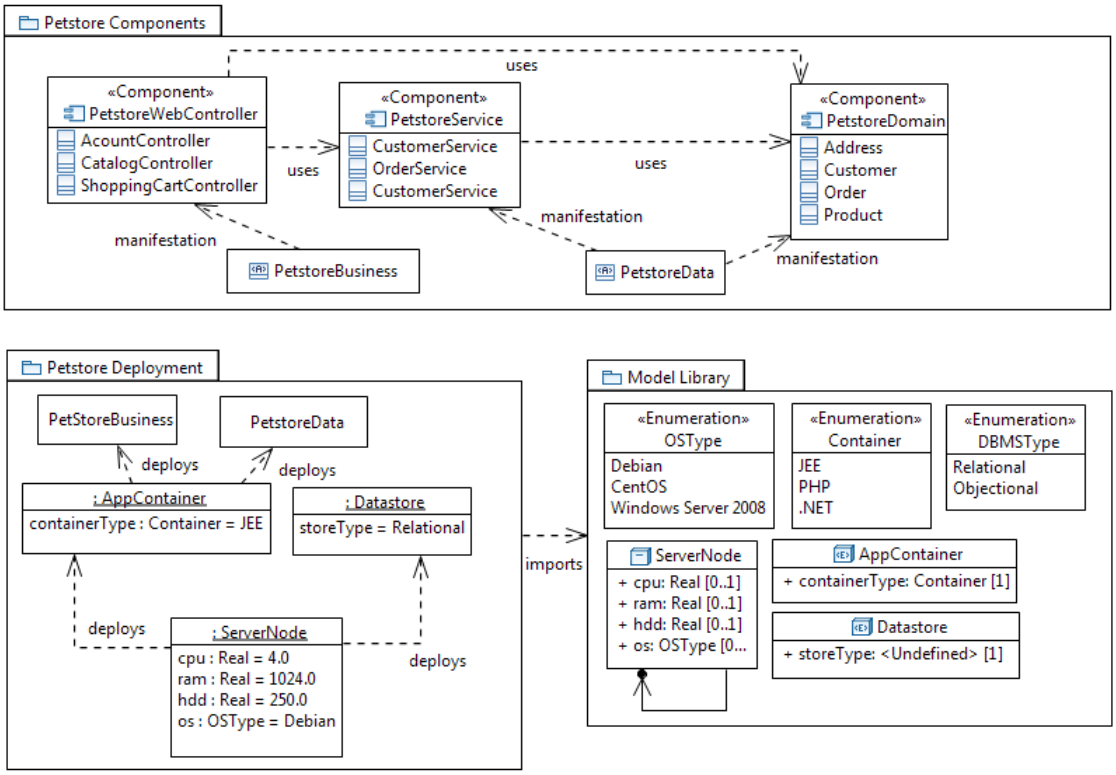

Figure 1.2: Deployment Requirements of Sample Application

In the following two sections, the aims of the thesis is described that try to solve the previously mentioned challenges:

1.3.2 Cloud-based Application Deployment

To facilitate application deployment to the cloud and to express information on a higher level of abstraction a model-based approach is used in this master's thesis. According to Selic [48], when designing a domain specific language that should be based on UML, the first step is a metamodel, which is developed isolated from any UML idiosyncrasy. In this work, the metamodel contains general cloud computing concepts, how cloud resources are arranged and concepts that are specific for IaaS in order to facilitate the modelling process of an application and its deployment requirements.

To provide application modelers a familiar environment, the metamodel is used for a UML-based solution, which is the CAML library discussed in [8]. Furthermore, the application modeler can include predefined templates of common execution stacks and blueprints of best-practise

examples for orchestrating cloud resources. The UML library contains profiles and stereotypes that can be applied to model elements to enhance the expressiveness.

The metamodel is designed in a way to comply with the following points:

- Cloud provider independent: The structure of the metamodel is cloud provider agnostic, which means it does not contain specific concepts from one cloud provider, but introduces a generic way of describing cloud resources.

- Focus on IaaS: As already mentioned, the metamodel contains generic cloud computing concepts, with a focus on IaaS, including individual configuration of cloud resources, such as virtual networks and virtual machines

- Differentiation between PIM and PSM: the differentiation between platform-independent model (PIM) and platform-specific model (PSM) can alleviate the model process even more [41], as a PIM contains generalized cloud service definitions, which are independent of the cloud provider and so does not restrict the modeler to one specific vendor.

Apart from defining required cloud resources, the definition of scalability behaviour of virtual machines is important, as elastic scaling is one of the advantages of cloud computing. The metamodel for defining scaling rules has the following characteristics:

- Focus on IaaS: The metamodel facilitates the definition of scaling rules on the IaaS cloud abstraction layer. It contains various scaling criteria and scaling statistics that are used to trigger a scaling operation.

- Cloud provider independent: The design of the scaling metamodel is cloud provider independent and it is possible to express scaling rules once and use them for different cloud providers.

1.3.3 Provisioning to the cloud

Once all deployment requirements have been manifested in form of models, they have to be transformed into a format that can be interpreted by cloud providers. In other words, the information related to deployment requirements should leverage the automatic provisioning of required cloud resources.

In this work, automation is achieved by model transformation rules. There are two types of transformation rules that are used in this master's thesis:

- Model to Model (M2M) transformation rules: The application modeler defines deployment requirements in form of UML models and applies profiles and stereotypes to model elements to enhance the expressiveness. The UML models have to be converted by means of applying M2M transformations into a model that conforms to the metamodel presented in this master's thesis.

- Model to Text (M2T) transformations rules: Based on the metamodel discussed earlier, M2T transformation are implemented targeting two formats that are interpretable by cloud providers and can support the provisioning process: CloudFormation from Amazon and Heat from OpenStack.

1.4 Approaches and Methodologies Applied

The master's thesis is based on design science introduced by Hevner et al. [33], following a constructive approach. The three phases (analysis, design and evaluation) are furthermore subdivided into the following points:

Analysis phase. The evaluation of existing approaches and technologies, which probably can be integrated or adapted, is crucial, as others may have already identified similar problems and presented solutions for them, which could be the basis for the approach proposed in this work. Critically analysing their strengths and weaknesses of each evaluated approach is important, in order to be able to learn from their conclusions and probably to avoid similar problems.

APIs, which try to unify proprietary cloud provider APIs, such as deltacloud[3], jClouds[4] or libCloud[5] are promising candidates to be part of the developed approach in this master's thesis. The applicability and usability has to be investigated.

Open standards, which try to describe cloud resources in a cloud provider interpretable way, such as the Open Virtualisation Format (OVF) or TOSCA[6], standardised interfaces such as the Open Cloud Computing Interface (OCCI) [7], or proprietary formats such as CloudFormation from Amazon, could be helpful for the provisioning process.

It is important to know, which cloud provider offers what kind of cloud service and how they can be configured through their respective APIs, as their offering can differ from each other.

Design phase. In the design phase, the process is planned conceptually, which includes both sub processes: the deployment and the provisioning. For each activity of the process input and output are specified. Furthermore, a metamodel for describing cloud deployments and scaling rules are is developed. Afterwards, the metamodel is used in a UML context in form of a model library and UML profiles. As soon as the metamodel is defined, transformation rules can be created (M2M and M2T).

Evaluation phase. In the evaluation phase, a tool support is provided based on a prototypical implementation of the proposed approach. The expressiveness of the approach and its feasibility is evaluated based on three representative case studies.

[3] http://deltacloud.apache.org
[4] https://jclouds.apache.org/
[5] http://libcloud.apache.org/
[6] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca
[7] http://occi-wg.org

1.5 Structure of Thesis

The master's thesis is structured as follows: In Chapter 2 a brief introduction to cloud computing and MDE is given. Furthermore, as a model-based approach is applied in this work, the question of how cloud computing and MDE can be combined is discussed. Chapter 3 introduces a high level process to support deployment and provisioning, the metamodel that defines the structure of deployment descriptions and scaling rules, to define scaling behaviour of virtual instances. In this chapter, the focus is set on how each step of the process is realised and how predefined execution stacks are included during the deployment. In Chapter 4, a prototypical implementation of the mentioned process is discussed. The first part mentions technologies and frameworks that are used for implementing the solution. In the second part of this chapter, the prototypical implementation is discussed by emphasising on technical details. Afterwards in Chapter 5, the solution is evaluated by means of representative case studies. Chapter 6 covers similar approaches and compares them with the approach proposed in this work. Finally, Chapter 7 draws a conclusion and mentions potential extensions. Moreover, a critical reflection including limitations of the developed approach is given.

CHAPTER 2

Cloud Computing meets Model Driven Engineering

2.1 Cloud Computing Principles

Is cloud computing only a new word used for marketing purposes to describe already existing technology? Back in 2008 the CEO of Oracle, Larry Ellison said at the Oracle OpenWorld conference *The interesting thing about cloud computing is that we have redefined cloud computing to include everything that we already do ... But I do not understand what we would do differently in the light of cloud.* [17], but Armbrust et al. try to invalidate in [5, 4] the assumption that cloud computing is an invention of the marketing department. When speaking about hardware provisioning and pricing structures, there are three main differences compared to conventional computing[4, 28]: (i) there "exists" infinite computing resources that are available when needed, which means additional capacity can be added whenever necessary (ii) companies do not have to commit themselves in advance to hardware infrastructure considering it as variable costs rather than depreciated capital and (iii) payment of used resources, such as processing power and storage, is done on a short-time basis, which means renting 100 virtual machines for one hour costs the same than renting one instance for 100 hours.

Another important point is the distinction between cloud consumer and cloud provider. A company, which relies on cloud computing infrastructure to be able to offer their services, is a consumer and at the same time can be a provider on a higher abstraction layer. The different abstraction layers will further be discussed in 2.1.1.

The National Institute of Standards and Technology characterised cloud computing with the following capabilities [6]: (i) on-demand self-service: A consumer can ask for additional computing facilities or network storage without interacting with a sales-person of the cloud provider, (ii) broad network access: Cloud computing resources are accessible over the internet, (iii) resource pooling: The provided computing and storage facilities are located in big data centres spread all over the world. Consumers share physical and virtual resources, which can dynamically be assigned or reassigned over time. Users do not have the ability to determine the exact

location of their resources, except on a higher abstraction (e.g. region of data centre is *eu-west-1*, which could be Amsterdam, Dublin, Frankfurt, London, Madrid, Milan, Paris or Stockholm in the case of AWS[1]), (iv) rapid elasticity: Additional resources can be provisioned automatically and within a short time period. Although physical computing capabilities are limited, in theory they are unlimited available to every user. Rapid elasticity also means automatic down-scaling of resources, if the workload drops. Scaling often can be configured time or load depended and (v) measured service: Cloud resources are constantly monitored and outages or performance issues are reported to the provider and the user of the services.

Harman et al [28] think the main argument to deploy software into the cloud is a question of optimisation and efficiency: "Optimisation of resource usage can be achieved by consolidating hardware and software infrastructure into massive data centres, from which these resources are rented by consumers on-demand".

2.1.1 Cloud Computing Service Models

Mainly, there exist three different cloud computing abstraction layers, namely Software as a Service, Platform as a Service and Infrastructure as a Service. Figure 2.1 illustrates this architecture of layers, whereas each abstraction layer has its own characteristics [6]:

- Software as a Service (SaaS): Consumers can access an application through a thin client, like a web browser or a mobile application. The underlying cloud resources are managed by the provider. Cloud storage providers such as Dropbox[2] or GoogleDrive[3] are examples for SaaS.

- Platform as a Service (PaaS): Consumers have the ability to deploy and run self-created or purchased software in the cloud. Often the platform is tied to a couple of specific programming languages, for instance Google App Engine[4] currently supports Java, Python, PHP and Go and faces some limitations concerning installed libraries. Furthermore, specific platform services may not be available within other PaaS clouds, which leads to vendor lock-in (see Section 2.1.2).

- Infrastructure as a Service (IaaS): Although the physical hardware is still maintained by the provider, consumers can use virtual instances, network storage and other virtualised resources on which software can be deployed. Even the operating system of virtual machines can be configured and tailored. Amazon is according to [31] with *Amazon Web Services* the leading company in IaaS.

Borders between IaaS and PaaS are becoming indistinct, the authors of [4] even refuse to differentiate among IaaS and PaaS, as they think that general accepted definitions still vary broadly. They prefer to distinguish between utility computing (IaaS and PaaS) and SaaS. Harman

[1] `http://aws.amazon.com/about-aws/globalinfrastructure/`
[2] `https://www.dropbox.com/`
[3] `http://drive.google.com/`
[4] https://developers.google.com/appengine/

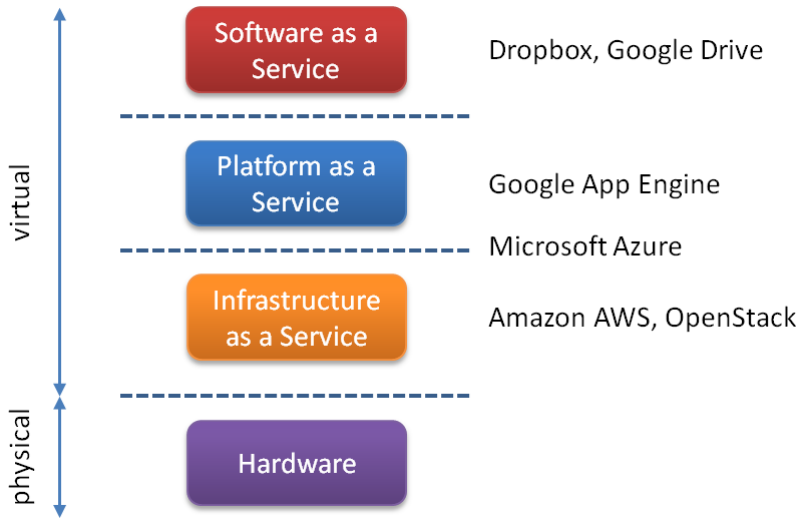

Figure 2.1: Cloud Computing Abstraction Layers, Source: Author with ideas from [28]

et al. [28] speak of *artificial distinctions*, which should be avoided as for instance data storage services can be assigned to one of the three abstraction layers depending how they are used.

Although Amazon initially started with a service to provide virtual machines and storage, they try to gain market share in higher abstraction layers of cloud computing as well, as with AWS they try to offer a *Swiss army knife*. Based on their original cloud computing infrastructure, they constantly introduce new services, which are free-of-charge, as only for the underlying cloud resources one has to pay for. For instance, Elastic Beanstalk[5] abstracts away the configuration of virtual machines and creates a chosen environment automatically, in which users can run their applications. This can be compared to the Google App Engine.

2.1.2 Potential Risks and Challenges When Moving to the Cloud

Although in theory the migration of applications to the cloud are seamlessly, in practise one can face serious problems and risks. An application could be designed to run in a local environment or on dedicated server infrastructure and therefore cloud computing technologies and principles have not been considered. Moving to the cloud can be beneficial, but according to [32] the main reasons why companies decided not to take the risk to move to the cloud, were concerns about security and integration and unexpected costs. In this section some of the problems and risks are described in detail.

Cloud Costs. Beside the financial benefits (no commitment to hardware in advance, rapid elasticity, per-use principle), which already have been mentioned earlier, there exist some drawbacks as well. The main question is about overhead costs during migration, which can occur

[5] http://aws.amazon.com/elasticbeanstalk/

during redevelopment of parts of the application [55], as well as the reoccurring operating costs for cloud resources [38].

Technology Gap. Tran et al. [55] exemplified a migration scenario of a .NET application to the azure cloud. The application was developed a few years ago and hence relied on outdated technologies, whereas usually in the cloud one has to build on the latest technologies. In this concrete example the gap was between an SQL Server 2005, the application was based on, and SQL Azure (comparable to SQL Server 2008), which was not compatible to older versions. Due to the evolution of services there may be no direct method to migrate from an old version to the newest one, however detours through intermediate migration steps have to be accepted.

Vendor Lock-In. To be able to migrate an application to the cloud, some parts have to be re-engineered, such as the database layer. Starting from IaaS, the higher the cloud abstraction level is, the bigger the dependency on provided services and APIs gets. For instance, if a virtual machine is created with Amazon AWS, the configuration of the application stack can by done ad libitum. As soon as more things are abstracted away, less flexibility and customizability is provided to the consumer. This means in general, on a PaaS layer more application re-development has to be done than on IaaS [6].

From a consumer perspective it is clear that the migrated application should not be executable only on one specific cloud, but with low adaption effort on an arbitrary one. Especially when using PaaS, the provided APIs greatly differ from each other, which means once an application is developed against API A, changing to another cloud provider with API B would be time consuming and would result into additional costs.

One mitigation strategy is to unify different cloud APIs and use one homogenous API (see Section 6.1.2), against an application is programmed to that enables interoperability and easy application migration to other cloud providers. This provider agnostic interface, hides provider-specific characteristics but at the same time takes off the ability to use provider-specific unique features, as it can only support the "lowest common denominator" set of capabilities [6].

Whereas heterogeneity may be attractive for cloud providers, a unified API would lower the switching costs of consumers, but will not necessarily cut down the provider's profit as standardization would enable consumers to run their applications in both private and public cloud environments, which could expand the market [4].

Replication and Scalability. Legacy applications possibly were not developed with a hidden agenda of replication, which means it can not be replicated without further modifications. There are two types of parallelism: user-level (users accessing the same service) and data-level (data can be handled in parallel) [28]. Before migration, one has to think about implications and consequences if the application still runs faultlessly if instances are replicated to meet the current workload. If the workload can not be parallelised an additional virtual instance would be worthless.

The authors of [1] focus on data management in the cloud. The replication of data among large geographic distances stands in conflict with the ACID principle, which is provided by common

transactional database systems. Mostly, consistency is loosened in favor of availability, as both can not be guaranteed as the CAP theorem shows [26].

Data Privacy. Especially nowadays with the disclosure of the NSA surveillance program PRISM [54], data privacy in the cloud is a well-known issue. Application data were formerly stored on companies internal servers, which were under the control of them. Confiding critical or secret data to a third party should not be done without thinking about proper encryption [53]. Another important question is the location of the stored data, as by definition *cloud* can mean everywhere on the world. Companies may be worried about the physical location of their data, as they become subject of the local data-protection laws, which may be in conflict with laws of the home country or contracts with their customers [15]. Transactional databases normally contain detailed operational data of a company and its customers, which must not be disclosed under any circumstances, as this would result into data privacy violations [1].

2.2 Model Driven Engineering

The Object Management Group (OMG) published in 2000 a paper about a strategy of a Model Driven Architecture (MDA), which is a special initiative of the common principle of MDE [11]. There are three main characteristics of the MDA strategy [49]:

Higher levels of abstraction. The problem as well as the solution are modeled on a higher level of abstraction. This ensures that irrelevant and distracting information of a complex system is stripped away. The concepts of the problem domain ideally can be described with modelling languages, which hide the underlying implementation technologies. This leads mostly to non-text representations which is easier to comprehend.

Automation. Due to a higher level of abstraction, it is easier to implement a computer-based automation to support analysis, design and implementation. Any reoccurring task, which can be accomplished by computers better than by humans, is suitable for automation. Model transformations or conversion of high-level abstracted models to the program level (source code generation) are predestinated for automation. With modelling languages such as eUML[6], it is even possible to execute them and evaluate correctness and suitability of the modeled system. A designated intention of MDA is that the transformation of models should be at least semi-automatic, to minimise the effort of keeping the models updated and to lower the maintenance costs [37].

Industry standards. With the usage of industry standards and best practises, a common basis for communication and collaboration of people from different fields is provided. To prevent lock-in to one specific tool or vendor, MDA uses open industry standards (such as UML), which provides the possibility of exchanging models between diverging tools. With UML, a model can be "constructed, viewed, developed, and manipulated in a standard way at analysis and

[6] `http://www.soyatec.com/euml2/`

design time" [51]. This means a system can be analysed and criticised before starting with the implementation and so, structural changes are still easier and economically feasible.

When object oriented programming languages got popular and replaced procedural ones, the key principle was to think about "everything is an object", which simplified the way of writing new software. The widely accepted software paradigm of object orientation may have reached its boundaries and has to be replaced with a new one, which is MDE and which changes the key principle of thinking to "everything is a model" [11]. As mentioned in [51], "Companies that adopt the MDA gain the ultimate in flexibility: the ability to derive code from a stable model as the underlying infrastructure shifts over time. ROI flows from the reuse of application and domain models across the software lifespan". If this argument delivers in practise what it promises, is still a matter of discussions and personal opinion. Reasons why MDE still has not replaced the object oriented paradigm are listed in Selic [49] and can be grouped into three categories:

Technical hurdles. The usability of available tools for MDE is still poor and causes a decrease of the learning curve for developers. Although the functionality of such tools can be quite matured, complex or counterintuitive tool sequences make the application of such tools unattractive. The other major problem, among technical issues, is the lack of amply theoretical background for MDE. Most MDE technology was created to solve individual problems targeting specific issues, which stands in contrast with programm-oriented methods that come with a broad coverage of theoretical background knowledge and solid patterns to avoid common problems.

Cultural hurdles. Even if people are aware of potential advantages and benefits when using a model driven approach, the inhibition level of introducing new technologies and tools into a functioning development environment is still high. The additional overhead cannot be disputed. A far more critical factor are software developers, who tend to think in general conservatively. For most technologies, not an insignificant time has to be brought up to get an expert, which implies a rejection of a new technology, even if it could offer more features.

Economic hurdles. Shareholders are interested in profit and it is not easy to justify an introduction of new development methods and tools, especially if it cannot be assured that the investment will be worth it.

Making one step further and moving away from the object oriented paradigm, where the relationship between objects and classes can be described with *instanceOf* and the connection between classes with *inheritsFrom*, towards to a MDE approach results into the ability to describe a certain view or detail of a system in form of a model. The way a model describes a system is defined in its metamodel. The linkage between a system and its model is called *representedBy*, and a model *conformsTo* a metamodel [11].

PIM versus PSM in MDA. The idea behind PIMs is to describe a system in such a neutral way that it does not contain any platform or implementation specific constraints and characteristics

[11]. Miller et al. mention in [41] advantages of PIMs, such as an easier validation of correctness of models or integration and interoperability among different systems "can be defined more clearly in platform-independent terms, then mapped down to platform specific mechanisms". The additional expenses when separating application aspects into PIM and PSM are compensated due to the increased flexibility as one can rely on stable platform-neutral models from which PSMs can be derived. A PSM contains both technical details, which can differ depending on the underlying platform, and business semantics, which originate from the related PIM. The better the PSM reflects the execution platform, the better application code can be generated automatically. Another advantage of PIMs is the resistance against shifting enterprise boundaries: Developed modules would have to be re-developed from scratch if the underlying technology changes, which can be prevented when using PIM [51].

It is unquestionable that the correct tools have to be provided to make MDE effective. It should be clear that models are not only used for documentation purposes, but are artefacts that have to be maintained like program source code. This can be only achieved, if the additional time expense results into a perceivable benefit, because if not, cultural and economic hurdles (as discussed earlier) will prevent the establishment of a MDE approach [37].

2.3 Model Driven Engineering for the Cloud

After having explained both topics separately, it is time to focus on the question if and how they can be combined. Is it possible to use MDE principles in the context of cloud computing deployments and in which way could they be supportive and benefit from each other? Deployment requirements and dependencies ideally should be describable in a structured manner, which serves as an interface between the MDE world and the cloud computing world.

What is the best way to define requirements an application has on the underlying computer infrastructure? Is there a way to use this definition not only for documentation purposes, but also for a better understanding of the application infrastructure? Can this model of application requirements be created in a way to support the maintenance and evolution of applications towards the cloud?

It turns out that a model driven approach, which was explained in 2.2, is a very good solution for this specific problem. There is definitely a need of "an advanced high-level programming model for building Cloud-oriented business solutions in a multi-provider environment" [52]. Ideally an application developer has the ability to execute the deployment and provisioning of the application on her own, without having a profound knowledge of complex server configurations.

Application requirements can be categorised into the following group of constraints [52]:

Hardware constraints. Requirements such as hard disk size, amount of CPU cores, processor architecture or amount of working memory are examples for hardware constraints. In some cases, a cloud provider only offers a certain combination of hardware specifications and groups them together into instance types or flavours. For instance, Amazon has a great variety[7] of

[7] `http://aws.amazon.com/ec2/instance-types/#instance-details`

different instance types. *m1.xlarge* for example provides 4 CPU cores and 15 GB of working memory. It is clear that hardware requirements mostly can not be fulfilled accurately, but the best compromise has to be determined.

Software constraints. Starting with the operating system, all software that is needed to run the guest application can be considered as software constraints. On every cloud abstraction layer, the constraints in relation to the required execution environment may differ or even may be limited by the cloud provider. On an IaaS level pre-baked images, which contain pre-installed software, can be used to initialise the virtual machine, whereas additional software has to be installed once the machine has booted.

Storage constraints. An application needs to store data, such as log files or raw data. Constraints may include the storage location or the required disk size.

Data constraints. An application normally has a persistence layer to read and write data. If this source has to be moved to the cloud as well, there may be some data constraints. For example the following requirements may be interesting: type and version of the database management system, database engine, size of database, backup strategy or type of storage solution (key-value, relational or document-oriented).

Security constraints. If an application has special security requirements, it is important to know, if a certain cloud provider is offering solutions to fulfil them. Furthermore, policies in form of firewall rules and access restrictions are constraints that have to be considered.

Performance constraints. A company, which provides an application, can have service level agreements with their customers. For instance, the application must be available 99% of the time. With the help of deploying an application to multiple data centres or the usage of elastic scaling and load balancing, such requirements can be fulfilled.

Cost constraints. Not all cloud providers have the same cost structure and some of them may be cheaper than other ones. A company's application could have a limited budget that must not be exhausted. Appropriate cloud resources have to be acquired to ensure to stay within budget.

Compliance constraints. These constraints are about legal requirements an application may have. The question is, if a set of cloud resources are available to satisfy regulations.

It is possible that constraints or part of them are not relevant for a certain cloud abstraction layer. Given the example of software constraints, one may not be able to choose the operating system of a virtual machine, if working on a PaaS level.

Furthermore, the description of all constraints should be free from technical details and should abstract away any vendor specific attributes. In this way, requirements matching can be done with multiple cloud providers and the best one eventually could be selected.

In some cases a developer may not be aware of all capabilities and services a cloud provider offers. This results into the necessity of semi-automatic search capabilities of available resources. Strategies of how this can be achieved are described in [30, 29, 56].

2.3.1 Unified Cloud APIs

To the best of our knowledge, standardised cloud computing interfaces targeting IaaS, which aim to be implemented by the cloud provider directly, are not widely adapted. This raises once again the question which format or strategy should be used to fill the gap between MDE and cloud computing.

There are various community driven open source libraries under development, which introduce an additional layer between cloud consumers and cloud providers and which homogenise cloud provider specific APIs. With such a library it would be possible to create a self-implemented application, which reads the deployment requirements from a file that has been extracted from the cloud deployment models in a previous step. As the library provides one unified interface for all supported cloud providers, the application would not have to consider any idiosyncrasy of a proprietary API, which is not necessarily compatible with other APIs. Because of the speed new features and capabilities are added to the APIs, it is unlikely that in the future there will be one common interface, as it is the strategy of cloud providers to set themselves apart from others with unique services [30].

This is an important fact: As another abstraction layer between the provisioning engine of cloud providers and the application modeler is introduced, the availability of new features in such libraries can be limited or delayed.

When provisioning cloud resources to the cloud, inconsistent states, where half of the cloud resources haven been provisioned and some may have not because of an error, are suboptimal. There exist only two desired states: Either everything goes well and all cloud resources are available or in the case of an error, already deployed resources should be deleted to guarantee a consistent state. This functionality is not provided by these libraries, in fact, each resource has to be created through a separate API request and subsequent requests cannot be linked to former ones. DeltaCloud or jClouds, which are describe in 6.1.2, would be examples for unified cloud APIs.

2.3.1.1 Middleware

A middleware, such as Cloudify (described in 6.1.3.3), is a software which is installed on a dedicated controller node, which on the one hand provides a vendor agnostic API to the user and on the other hand communicates with and manages all provisioned virtual instances. Each of them run an agent service, which provides the controller node with statistics about CPU load or hard disk write operations. Based on this information the controller node can decide if a scaling (starting or terminating virtual machines) has to be performed. Furthermore, software updates or any other operations the user desires, are propagated by the controller node.

The middleware constitutes another layer of abstraction and therefore is not limited to one cloud provider. Technically in the case of Cloudify, jClouds is used to communicate transparently with cloud provider such as Amazon AWS. Often the middleware provides a way to use

a template description of needed cloud resources to be provisioned. The unification once again comes along with a limited feature set.

2.3.2 Templates

The most promising approach of describing a set of necessary cloud resources is in form of a templates. One of the main advantages is that the template can be parsed by the cloud provider in advance and any contradictions concerning configuration can be identified before the provisioning takes place. Furthermore, cloud providers process templates transactional, which means that if the provisioning of one resource fails, everything gets rolled back and any already provisioned appliance will be deleted. The other way around, once the system was able to provision all resources contained in the template and terminates successfully, it is guaranteed that all resources are available. This means there does not exist an inconsistent state.

There exist different template formats, whereas some of them originate from or where inspired by others. Examples are HOT, CloudFormation or OVF. An example how such a template can look like, is given in Listing 2.1. It defines a simple virtual machine, with certain properties such as flavour, type of image and the name. The description of cloud resources in a structured way can be read by both, humans and machines.

Listing 2.1: Sample Template of a HOT Template in YAML Syntax

```
1 heat_template_version: 2013-05-23
2
3 resources:
4   compute_instance:
5     type: OS::Nova::Server
6     properties:
7       flavor: m1.small
8       image: F18-x86_64-cfntools
9       name: Example Server Node
```

Cloud providers, who offer an API that is capable of interpreting such templates, often speak of orchestration of cloud resources. A template does not define how resources should be provisioned, but what kind of resources are necessary. This makes it easy to keep templates cloud provider independent, provided that the cloud provider supports the format. This is one of the most crucial points, as if a format or a standard is not supported by the majority of cloud providers, its practical applicability is limited.

Another advantage of templates is that models can be converted by model transformations to any arbitrary template format that can be interpreted by cloud providers. Moreover, a template can be checked into a version control system as they are simple text files.

2.4 Building an Application Stack in the Cloud

Once deployment requirements (including hardware and software requirements) have been captured in form of models, there is still the question how the required execution stack, which is necessary to run the cloud application, can be created automatically. This strongly depends on which cloud abstraction layer is targeted. As already mentioned, IaaS provides the most flexible

solution and virtual instances can be customized all-embracing. When speaking about PaaS, the dependency on APIs and available libraries is stronger and detailed information about hardware requirements (i.e. amount of CPU cores) can not be taken into account during migration. On the other hand, there are software requirements, which can be categorised into different layers, each of them depending on underlying layers and so on. All layers grouped together can be seen as a stack, which enables an application to be run in the cloud.

There are different approaches to address the problem of how to semi-automatically create and setup execution stacks on cloud resources. Figure 2.2 illustrates four different possibilities of a virtual image configuration to ensure that a Java-based application (such as the PetStore), can be executed on the virtual machine. The application *Cloud Application* needs a database, a web server and a Java execution environment, which are already part of the virtual image or have to be installed once the virtual machines has booted.

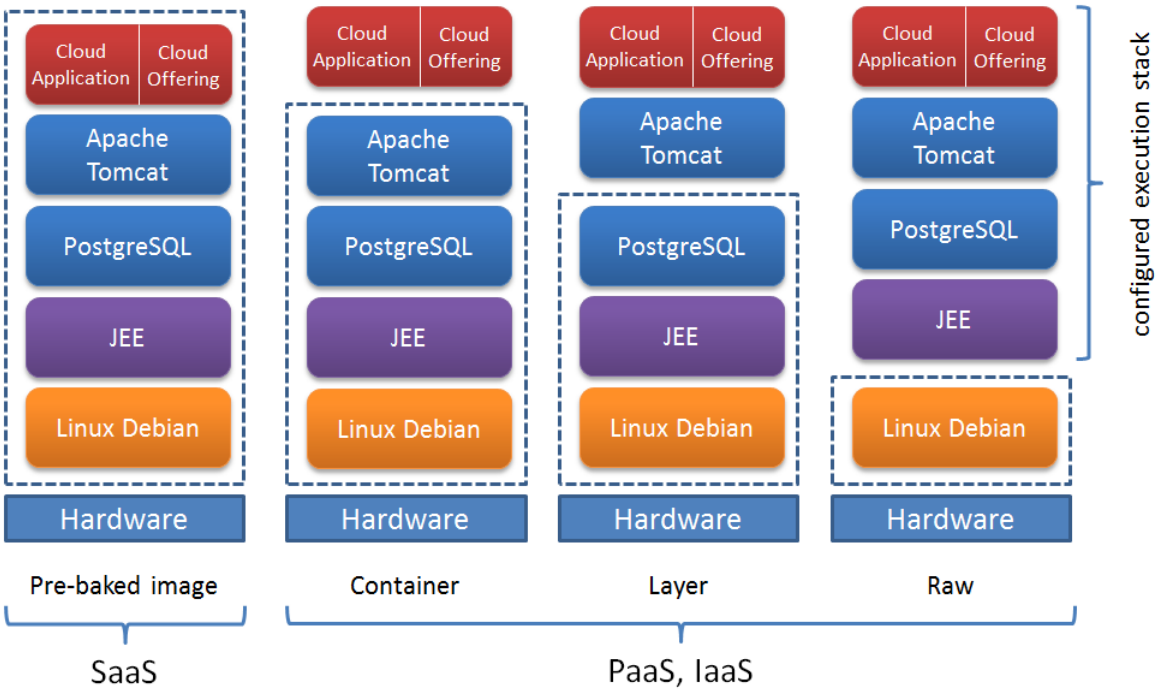

Figure 2.2: Virtual Image Configuration Options. Source: [2]

Pre-baked image. All dependencies, inclusively the application, are baked into one virtual image. If the existing application is already running in a virtual environment it is easier to create a snapshot of the server in its current state and transfer it to the cloud. The main drawback is when components have to be changed, such as updating or patching the web server, the operating system or the application itself. To persists such changes, the virtual images needs to be recreated. The advantage is, as soon as a virtual machine is initialised with the image, it is ready for operation instantly.

Container approach. Market places like *AWS market place*[8], which are tightly integrated with their provided cloud services enable consumers to look for pre-configured virtual machines images, which are ready to be deployed onto a virtual appliance. The creator can determine the hourly rate she wants to charge, which is billed to the consumer additionally to the normal expenses of using virtual resources. Although this approach is comfortable and does not include a lot of configuration it is quite inflexible. There may be some installed software, which are not necessary to run the application that would result into wasted resources. If the technology stack is unavailable, one would have to search for the most accurate alternative and eventually re-configure it afterwards. Compared to the pre-baked image strategy, the application code is decoupled from the rest of the software stack, which is handy when the application needs to be updated.

Layer approach. Compared to the previous two approaches, this is one is more agile. The strategy is to create images in advance only with the most basic layers, which do not have to be updated frequently. All layers above are described in an abstract way, often called recipes. These have to be interpreted and executed when the virtual machine is initially booted and before it can be used for productive purposes. The flexibility of such recipes, which describe everything needed to install and configure a specific software, lies in the definition of configuration parameters such as version numbers or user credentials.

An application that is capable to interpret and run recipes is Opscode Chef, which is explained in more details in Section 7.3.

Raw. As the image just contains the operating system, all configuration has to be done from scratch, every time a new virtual instance is booted. In relation to application scaling this may not be effective, as the installation of needed software may consume more time than the peak period, where an additional instance is needed, lasts. On the other hand, the used image does not have to be maintained and kept updated, as the majority of cloud providers provide images of up-to-date operating systems.

2.4.1 Search Based Software Engineering

The idea of search based software engineering (SBSE) was already considered in 2001 by Harman et al. in [27] and has been successfully applied to a various of different software engineering problems such as test data generation or automated patching [28].

The authors of [28] explore the possibility of how SBSE "can help to optimise the design, development, deployment and evolution of cloud computing for its provider and their clients" [28]. In the following the focus lies mainly on the client side, as this master's thesis describes the problem from a consumer's point of view.

In general SBSE can be seen as a problem solving method in software engineering, where "computational search and optimisation techniques" [28] are used to find an optimal solution among other potentially correct solutions, which may be in conflict. There are two steps involved to convert a common software engineering problem into one which can be solved by the

[8] `https://aws.amazon.com/marketplace/`

SBSE approach. (i) reformulation of the problem, which means the definition of candidates representing possible solutions and (ii) definition of an evaluation function, which fulfils the ability to determine the better solution, out of two given ones.

Concerning the previously mentioned container approach, [28] addresses the problem in detail and provides a solution in applying SBSE. When using pre-configured virtual images, some software components may not be used at all, which would waste unnecessarily resources and may affect both: client (additional costs, higher response time) and provider (demand of physical hardware). The main question is to determine the trade-off between the frequency of using a module and the possible reduction of allocated hard disk space when removed.

Partial evolution, which has a long history and was applied already in 1977 for specialising programs [7], is one method to identify those modules which are unused and therefore could be removed in favor of smaller machine images. In this case, dependencies between modules are represented as a graph, where single nodes or a small group of nodes can be striped away, which makes the graph smaller, but at the same time lowers the functionality.

The authors of [50] and [34] describe an approach of slicing unused parts of the source code to increase performance on the application layer. In general there are three methods: static slicing (without executing the program), dynamic slicing (considering program input) and conditioned slicing (bridging the gap between static and dynamic). [28] claim that the statical method of this approach can be applied to cut virtual images as well.

The third approach is a searched based approach to identify parts which can be deleted or even better to search for a way to create a new image, out of the old one and recording the intermediate necessary steps to assure reproducibility in the future.

CHAPTER 3

Deployment and Provisioning Process

After having discussed MDE and cloud computing and how they can be used together, the next step is to have a look on an approach that can be used to merge the gap between them. In the following, a process is described that aims to use deployment models to support the deployment and provisioning of applications and the required cloud resources.

During developing the modelling process, our goal was to keep it as flexible as possible regarding to extensions and target formats. The reason is simple: As it was elaborated in Section 2.3, there exist efforts to make a description of cloud resource vendor agnostic, but such approaches are rather new and still under development. In such a case, the process all the more should be easy to extend to target new formats or to define new ways a cloud deployment is described.

3.1 The Model-based Deployment and Provisioning Process

Figure 3.1 illustrates the whole process, which is proposed in this master's thesis. In general, the process can be split into two sub parts: Deployment and Provisioning. The deployment is about modelling application requirements and refining model elements with cloud computing concepts towards a selected cloud environment. Starting with general definitions and requirements of an application, a PIM model gets converted through any arbitrary amount of refinement iterations to a PSM model. In this context, platform independent means that the model is provider agnostic and does not contain any cloud provider specific information.

The provisioning on the other hand, takes a model that conforms to the CMM as an input and transforms it to an orchestration template, which is interpretable by cloud providers to perform the provisioning of the cloud resources.

In the following, we introduce the Cloud Metamodel (CMM) and discuss the deployment and provisioning process more detailed: (i) creation of modelling library and extensions for model refinements, (ii) model refinement, (iii) model to model transformation, (iv) model to text transformation and (v) provisioning engine, whereas each step consists out of sub-activities.

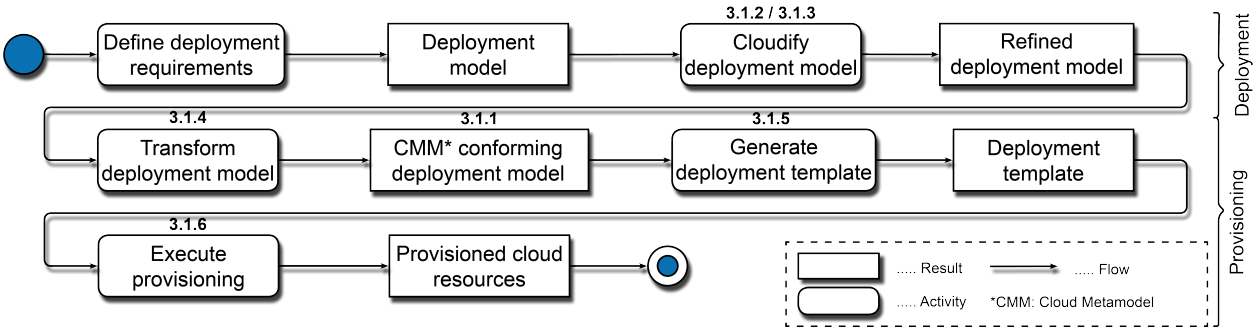

Figure 3.1: Deployment and Provisioning Process

From an application modeler point of view, the process is initiated by modelling the cloud application, which eventually results into deployed cloud resources, with certain provisioning information fed back into the model.

3.1.1 The Cloud Metamodel (CMM)

According to Selic [48] when designing a domain specific language that should be used as a UML extension in form of profiles or stereotypes, it is important to concentrate on the concepts and functionalities the DSL should have and to create a metamodel that is treated isolated from any UML idiosyncrasy. The mapping to UML correspondent elements is done in the next step.

Furthermore, the process in Figure 3.1 was designed from the end to the beginning, as firstly we had to know what kind of functionalities and possibilities cloud providers offer, to be able to design the CMM and to make realistic assumptions.

As already mentioned, a model conforming to the CMM is used as an starting point for the provisioning sub process. The CAML library, discussed by Bergmayr et al. in [8], is a CMM representation in UML, a so-called internal DSL manifested in the UML metamodel. This enables application modelers to model their application and cloud resources in a familiar UML environment.

Furthermore, CMM is the basis for all model to text transformations in the process, and adds additional flexibility to the provisioning process: Even if a model was not created with UML, it is possible to use the provisioning sub process. Through model to model transformations a model can be transformed into a model that conforms to the CMM.

In the following paragraphs, we would like to give a detailed overview of the CMM. Various cloud providers such as Amazon AWS, Rackspace and HP Cloud (both OpenStack), as well as middleware providers (for instance Cloudify) have been analyzed and a generic metamodel that enables the description of any arbitrary cloud deployment model was developed. The emphasis is clearly on IaaS clouds, and introduces concepts to describe infrastructure related cloud resources, network aspects and firewall rules.

Cloud Resource. Figure 3.2 shows the relation between a cloud resource and virtual appliances it consists of. A virtual appliance can have dependencies on other appliances, and pos-

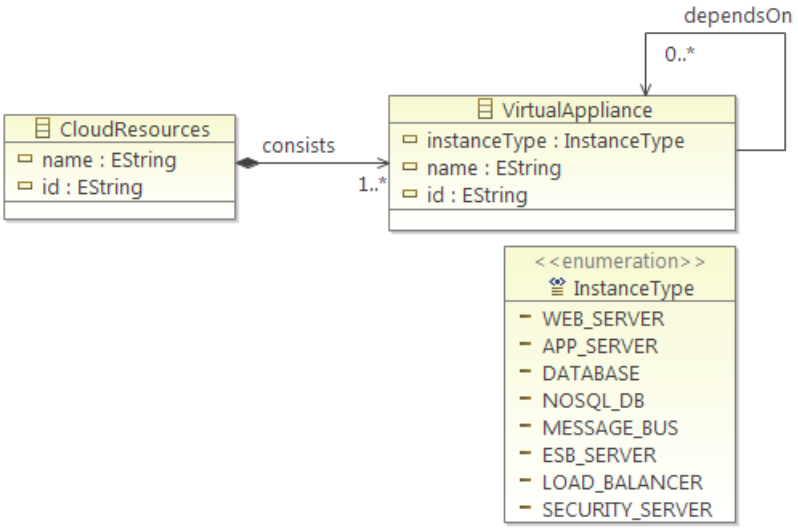

Figure 3.2: Relation between a Cloud Resource and Virtual Appliances

sesses an attribute, which roughly categorises the appliance into one of the listed instance types.

Virtual Appliance. Apart from the instance type, a virtual appliance can have requirements (see Figure 3.3), which have to be considered during provisioning. Region and availability zone are related to the physical location of the data centre of a cloud provider. Such definitions can be important for reasons like data privacy or resilience policies. Hardware specifications are normally summarized as an instance flavour (for example a F4 Google App instance has 512MB of working memory and a CPU of 2.4GHz). The image ID is used as a unique identifier for an image that should be used for booting the virtual instance. This can be a basic Linux installation or a pre-configured system, which already contains installed software.

Apart from requirements each virtual appliance has a stack, which is described by stack properties (for a more detailed explanation see the paragraph *Stack property*). We assume that a virtual appliance consists at least out of one stack property.

Stack property. A collection of hardware and software properties forms the stack of a virtual appliance. As already mentioned, in most cases hardware requirements are expressed as instance types, which would mean that information about the hard is redundant. In any case we decided to keep this information, as it may be useful in the future. Furthermore, not in all cases an instance flavour can be found for arbitrary hardware requirements.

As shown in Figure 3.4, the following hardware characteristics can be defined: CPU, HDD, Memory, the process architecture and the amount of virtual CPU cores. In the case of software it gets more complex. Figure 3.4 does not show all references regarding the software element, as this will be discussed in the next paragraph. Software can depend on other software, for instance a Tomcat6 server has a dependency on Java. In other words with those references a dependency

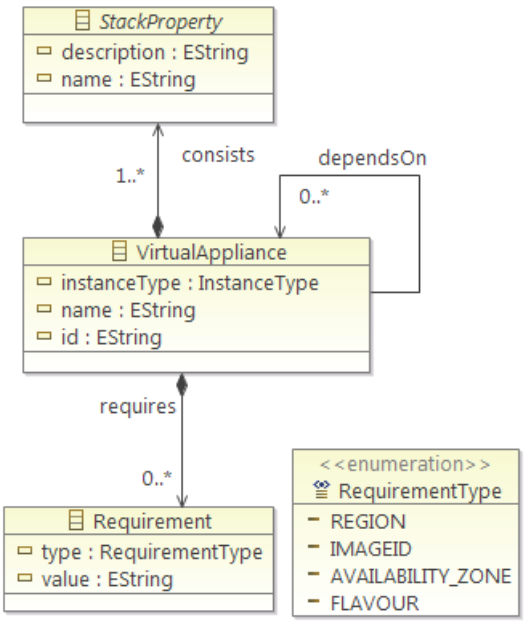

Figure 3.3: Requirements of a Virtual Instance

tree can be constructed. Most of the software can be obtained by package managers, such as apt or yum for Linux or MSI for Windows machines.

Software. Software packages can provide services that are exposed to other applications, services or users. The connection between a software package and a service is shown in Figure 3.5. Furthermore, each service comes with a set of firewall rules. This is necessary, as normally a newly created virtual machine cannot be reached from outside and any installed service listening for incoming connections would be unreachable. A firewall rule can be seen as a definition of how and under which circumstances traffic is let passing through the firewall. It is possible to define the type of IP protocol and the type of application protocol separately, as well as the traffic direction (i.e., if it is incoming or outgoing traffic). There also exists the functionality to define port mappings, which means for example incoming requests to a certain port are forwarded to the port of the virtual machine the service is running on. A rule also can be defined for specific IP addresses or IP ranges, so that the companies public IP may have access to services, which are only designated for internal usage and are not available for public.

3.1.2 Scaling Rules

Scaling rules are difficult to be represented in form of stereotypes. We decided to define an own domain specific language, which can be included into UML models. The goal was to design the DSL in such a way to make it cloud provider independent targeting IaaS clouds. The underlying

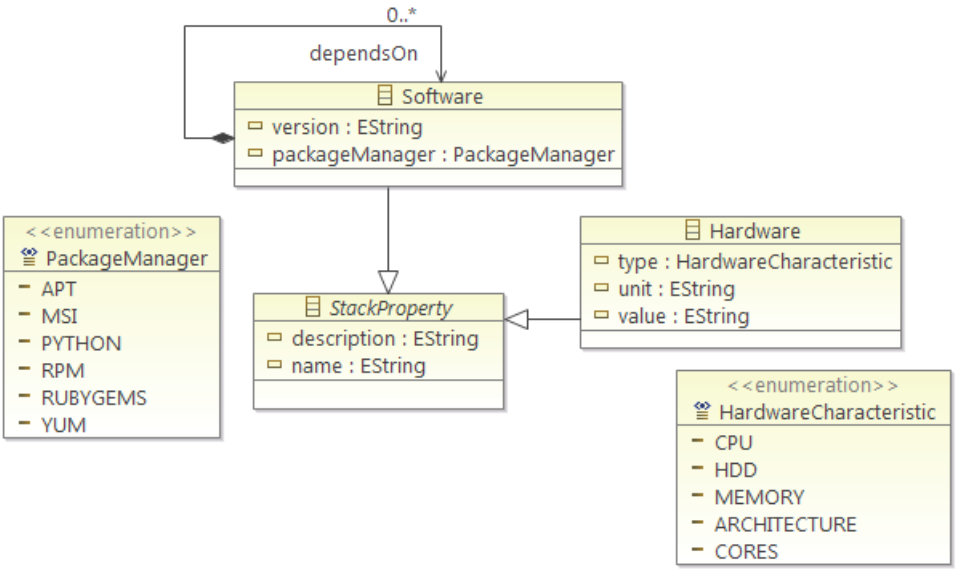

Figure 3.4: Definition of Stack Properties

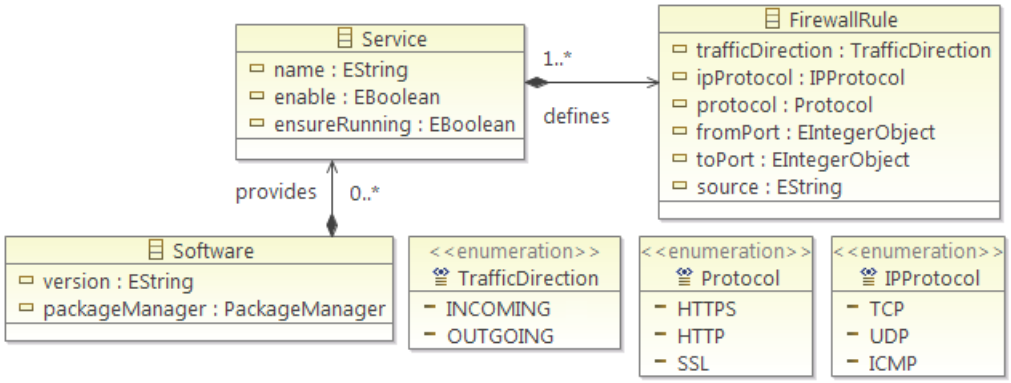

Figure 3.5: Software and Firewall Rules

model of such a DSL definition can be used as an input for model transformations, which makes it easy to process the information and integrate them into the provisioning process.

Scaling rule. A virtual resource can have an arbitrary amount of scaling rules, which contains definitions concerning the scaling behaviour. An excerpt of the scaling metamodel concerning rule can be seen in Figure 3.6. For instance, a scaling rule can use various scaling criteria, such as CPU usage or disk write operations, a cool down time in which no further scaling operations are performed and different scaling statistic that define how samples are aggregated. To keep the figure compact, not all enumerations are included, but in Section B a complete picture of the scaling metamodel is given.

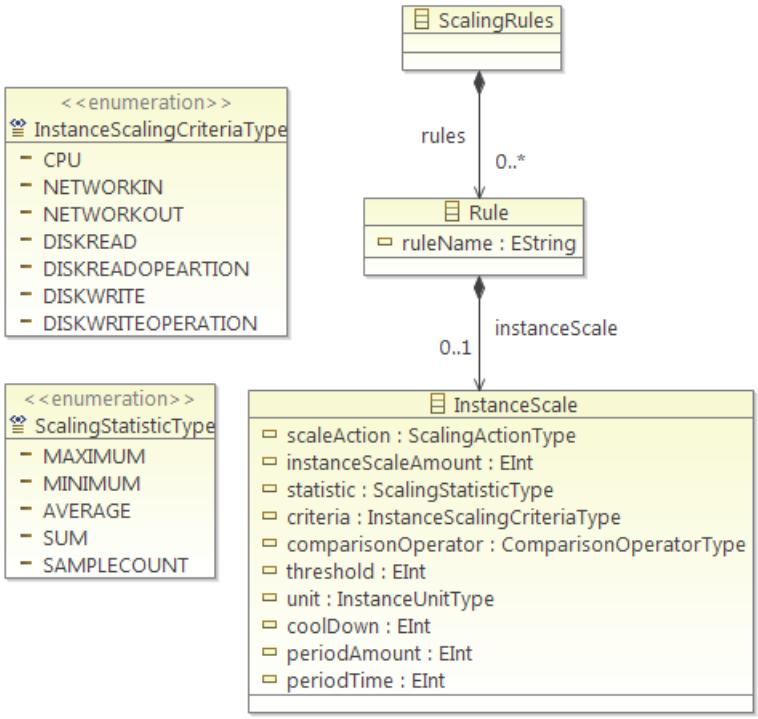

Figure 3.6: Composition of Scaling Rule

Furthermore, there are some scaling requirements that do not have to be defined for every rule. For instance, to limit the expenses the maximum amount of concurrent virtual machines can be defined. Another example would be the initial amount of virtual machine that should be created during provisioning.

3.1.3 Modelling Library and Extensions for Model Refinements

Once the CMM is defined, it can be used in the context of UML. In our case the CAML library from the ARTIST project, is a UML internal language based on the CMM.

A standard deployment model in UML contains elements, such as *Node* or *ExecutionEnvironment*, which are promising for modelling cloud resource deployment descriptions in an abstract way. Certainly they do not offer the whole scope of expressiveness we are striving for, but just represent the basic structure. The CAML library consists of profiles and stereotypes, which can be applied to model elements to add new functionalities and information to increase expressiveness. Furthermore, the library contains best practise scaffolds of possible deployment scenarios, to accelerate the deployment.

The aim is to start the process with a model, which is as much cloud provider-independent as possible. In this case different cloud providers and deployment options can be evaluated during the model refinement, which is described in the next section.

Furthermore, the library offers a mechanism for extensions, which can be applied to model

components during a refinement iteration, in order to reach a higher specificity of the model. There exist extensions for each abstraction level, which means some of them represent general cloud computing concepts, whereas others reflect possible configurations of one specific cloud provider. Latter ones consist partly of static information, which can be obtained through APIs, offered by the cloud providers and can be integrated automatically into the appropriate extension. This kind of data, such as availability zones or virtual machine characteristics, is subject to change and should be updatable by the modelling library maintainer in a controlled and semi-automatic way (see Section 3.2 for further explanations).

Deployment Blueprints. An application modeler may not be aware of all available cloud services and how they could be combined. With the definition of best practise deployment configurations, which can be used as deployment scaffolds, the modelling process can be facilitated and accelerated. Those blueprints mainly target the composition of cloud resources and the communication links between them.

For a better understanding, Figure 3.7 illustrates a blueprint for a common web application deployment. There are two load balancers, one distributes the workload among the web servers and the other one takes care that the application servers' load is balanced. For an improved reliability the servers may be deployed in different availability zones (geographical location of a data centre).

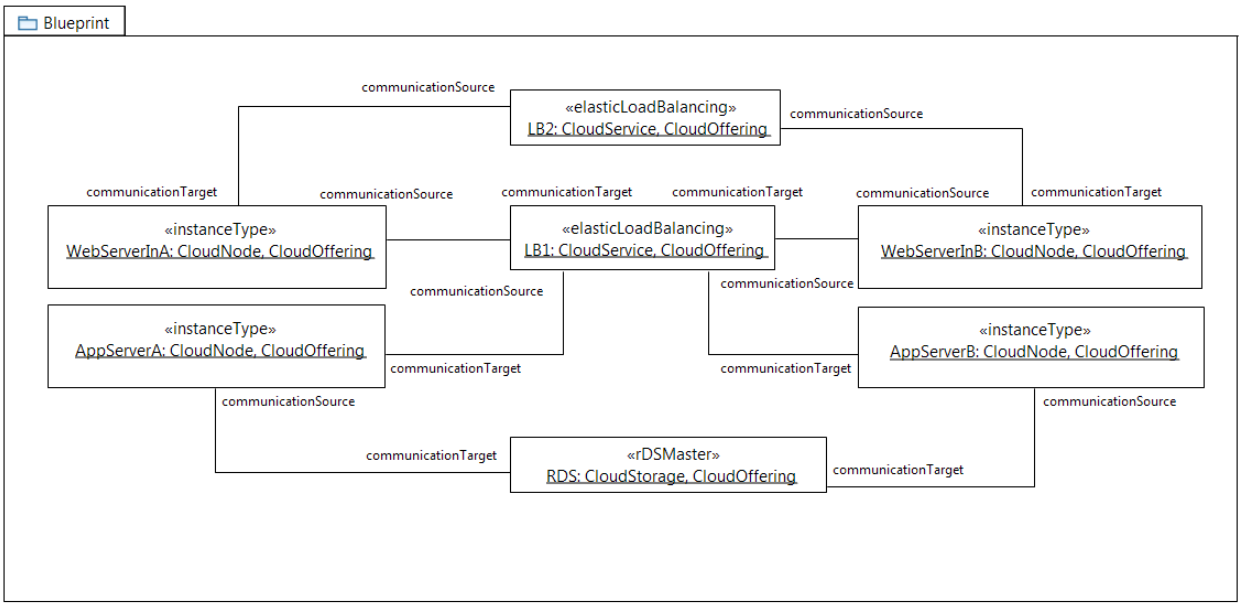

Figure 3.7: Blueprint for a Common Web Application Deployment

Execution Stack Blueprints. On top of an operating system, an application modeler may want to define constraints concerning the execution stack, to be able to run the application. There can

be limitations or only a couple of options the modeler can choose from (this strongly depends on the cloud abstraction layer). An execution stack consists out of software packages, which can depend on each other. Such dependency trees are suited to be modeled as blueprints, which only have to be referred and used by the modeler.

Figure 3.8 depicts a software stack for the Apache Tomcat 6. The software package *Tomcat6* has dependencies on other packages, and exposes a software service, which can be used by the entitled. For each software package, the package manager can be defined to configure how it should be retrieved and installed. Furthermore, specifications about the version and the language type a software package satisfies can be provided.

In the case of a software service, operating ports and protocol types are useful information in order to configure firewall rules or to establish port-forwarding mechanisms.

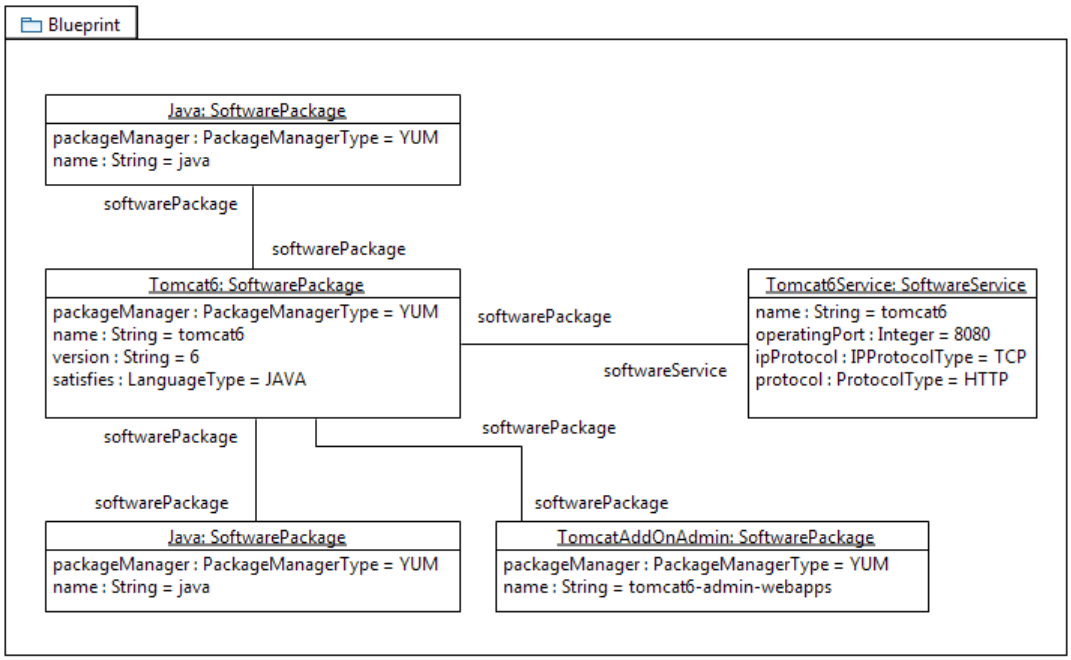

Figure 3.8: Blueprint for a Tomcat Software Stack

3.1.4 Model refinement

Coming back to the sample application that was introduced in 1.3, the generic deployment model has to be refined. In other words, additional stereotypes will be applied the model element *ServerNode*. Further refinements are: (i) applying stereotypes to other model components, (ii) executing predefined transformation rules, (iii) defining scaling behaviour, (iv) integrating supplemental data provided by the modeler. As the rules are defined by the maintainer of the modelling library, it is not the scope of responsibility of the modeler to adapt or change them. Nevertheless transformation rules are parameterized by considering manually provided information by the modeler and by offering choices in the case of multiple feasible solutions for defined

requirements. During this procedure, the deployment model eventually contains cloud provider specific data and is ready to be converted into a model, which conforms to the already discussed CMM.

Refining PIMs to PSMs. It is hard to draw a line and to say when a model is platform independent and when it gets platform dependent. Apart from the first and the last model, which obviously can be categorized as PIM and PSM, for intermediate refined models, it is every time a question of the point of view. Under certain aspects, a PSM still can be seen as PIM and vice versa. If the deployment model contains already specific data for one cloud provider, it may still be open, which software stack is going to be used by the virtual machines. We are aware of and accept this unclarity, as it does not affect the way of how models are refined.

To keep the graphical representation of the process simple, the refinement is illustrated as a one-time activity. This could lead to the wrong assumption that multiple refinement iterations are not possible, but the opposite is the case: An application modeler would perform small and fine-grained refinements that are summarised as one general refinement operation.

Coming back to the running example, which was introduced in the previous chapter, Figure 3.9 shows the transition from a model, which holds the information in an idiosyncratic way to a representation that uses stereotypes and elements from the model library, which can be used for further refinement iterations. Not all information has been included yet, but this will take place in further iteration, where more and more deployment relevant constraints are added.

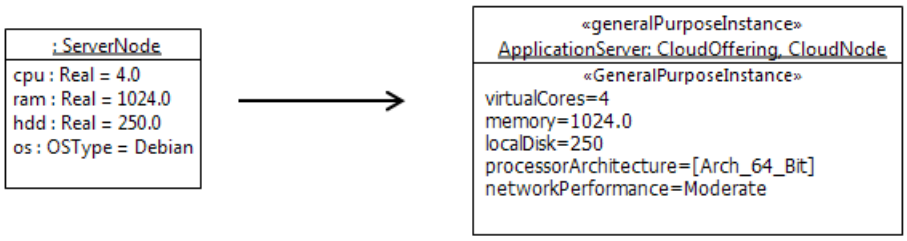

Figure 3.9: Transition to a Model that uses Model Library Stereotypes and Elements.

A possible result of the first refinement iteration is illustrated in Figure 3.10. The model is getting slowly platform specific, as a concrete instance type in form of a stereotype from Amazon AWS has been applied to the application server element. *m1_large* fulfils all specified hardware requirements and introduces new slots that can be defined and customised, such as *region*, *operatingSystem* or *availabilityZone*.

The diagram of the running example (see Figure 1.2) contained some application specific requirements, such as a Java execution environment and an application server than can serve the PetStore application (for instance Apache Tomcat). This information will be included in further model refinement iterations, in this iteration only in a general way though, as Figure 3.11 shows. The deployment target of the defined software stack is a language environment that requires Java as its language type. This is a rather broad description of application specific requirements, but facilitates the evaluation of different software stacks, which fulfil the language type.

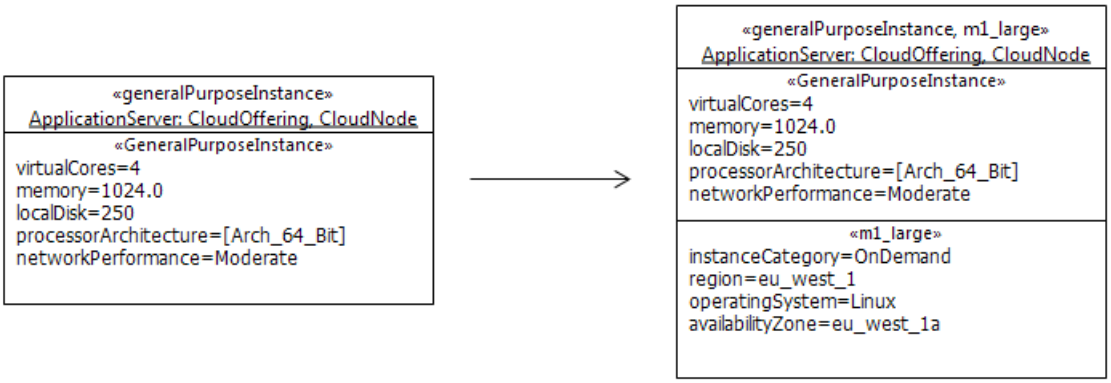

Figure 3.10: First Model Refinement Iteration.

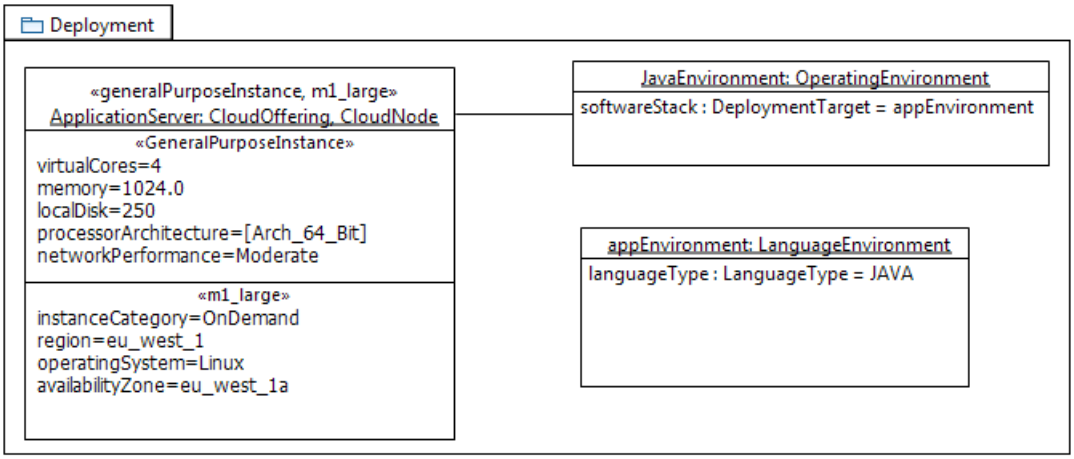

Figure 3.11: Second Model Refinement Iteration.

Following this iterative approach it is easier to find alternatives for some requirements, if it is not possible to completely satisfy them. As an example there may be a blueprint for a JBoss server, which meets the application language requirement and that could be used, if there was no blueprint for an Apache Tomcat web server.

Fortunately there was a blueprint available, which was included into the model, as Figure 3.12 depicts. After three refinement iterations, the model contains enough information, to be passed to the next step of the process.

3.1.5 Model to Model transformation

Conceptually this step has some similarities with to last one: The model gets transformed into another one. Nevertheless something unique and distinctive is happening, as in this step no

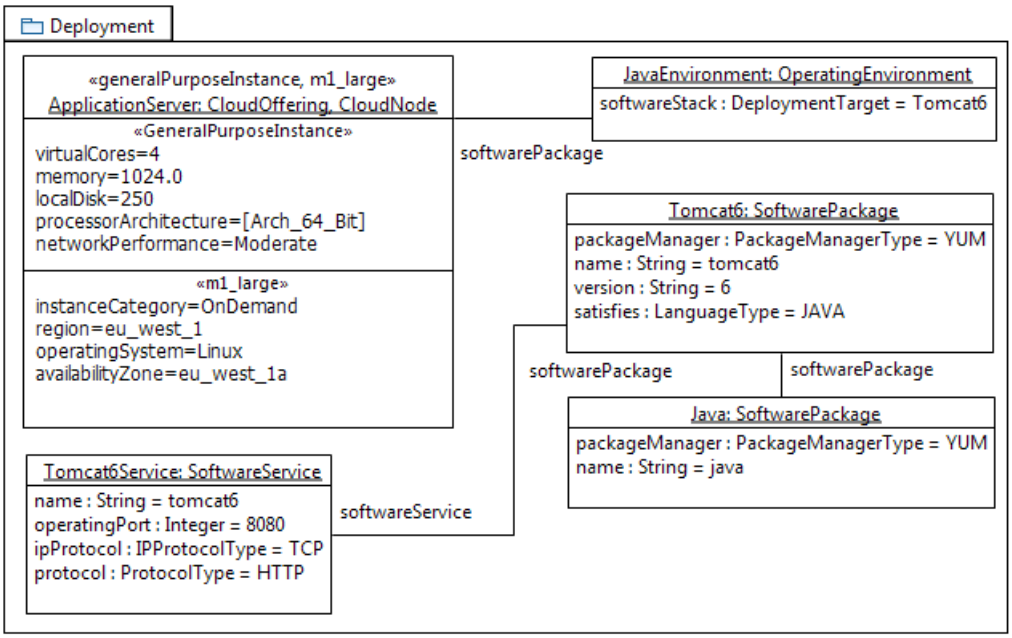

Figure 3.12: Third Model Refinement Iteration.

further refinement is done, but the model gets converted into a model that conforms to the CMM (see Section 3.1.1), which proposes a way of describing specific cloud deployments without restrictions of which cloud provider is being targeted.

One may argue that this step is unnecessary, as a refined model could be transformed directly to a representation, which can be interpreted by the cloud provider. The whole process has been designed in a way to be as flexible as possible, to make it even easier to enter the process at any desired step.

Transformation into a CMM conforming model. The introduction of the CMM enables anybody to convert their models, not necessarily created by previous steps of the process, into a model which conforms to CMM. It would be even possible to derive a domain specific language from the metamodel bypassing the whole UML modelling process. We think these arguments are reasonable enough, to introduce this extra step and justify additional efforts. Furthermore, it reduces the complexity of the next step, as model to text transformations can rely on only one metamodel and do not have to consider cases in which other metamodels are used. We want to emphasise once again that the deployment process does not have to be done in UML, but any other approach, persisting application requirements in form of models, is possible. The only condition is, to write model to model transformations to obtain a model that conforms to the CMM.

3.1.6 Model to Text Transformation

A representation of deployment requirements in form of a model cannot be parsed and consumed directly by a cloud provider, but has to be converted into a format that can be interpreted by a cloud provider. The chosen image strategy, which has been already discussed in Section 2.4, influences directly the structure of the orchestration template. If a pre-backed image strategy is used, less information about the required application stack has to be included in the template. The model to text transformation rules are designed by following the principle of convention over configuration, which means that reasonable defaults are used and adaptions have to be configured.

3.1.6.1 Transformation into a Cloud Provider Readable Format

Independently of which format a model is converted into, the principle of creating transformation rules stays the same: As the source model conforms every time to the CMM, a specific structure and relations between model elements can be assumed and facilitates creating rules. If a transformation to a specific format is required, which has not been implemented yet, it is easy to create those rules and/or extend existing ones. As the deployment information is represented in a format-agnostic model, structural changes in the target format or adaption requirements can be achieved by modifying the relevant transformation rules.

The strategy is to design a modular system by means of a plugin system. In other words, the core consists out of transformation rules that produce a valid orchestration template but does not consider fine-grained details. Each loaded plugin contributes its own rules and/or extends existing ones, which results into an orchestration template that also contains detailed specifications of cloud resources, such as the behaviour for elastic scaling.

3.1.7 Provisioning Engine

The last step of the suggested process is about the provisioning engine. Ideally the resulting representation of the model, also called the template, is interpretable by the cloud provider itself (through a web portal or a proprietary API).

The provision engine uses the template as an input and takes care that the resources are created in the desired cloud. The important requirement is that the creation of cloud resources is done transactionally: Either all parts of the orchestration template are created or as soon as the creation of one element fails, already generated resources are deleted thus the transaction gets rolled back.

After a successful creation, provisioning specific information such as IP addresses or DNS names can be fed back into the deployment model and represented graphically to the application modeler.

3.2 UML Profile Enrichment

The second process, which is illustrated in Figure 3.13, is about maintaining the modelling library. The process is not executed by the application modeler, but by the supplier of the library.

Figure 3.13: UML Profile Enrichment

Lets take the CAML library as an example: It contains static data that can change over time, for instance information about instance types, available instance flavours or pricing information. Such data can change on a regular basis and are suited for integrating them into the modelling library in an automatic way.

In the first step, the data gets extracted through proprietary APIs that are offered by the cloud providers and is stored locally in a structured way. The data is used as an input for the next step, which takes care about the enrichment of the respective UML profiles programmatically.

The goal of this process is to facilitate the update of the UML library in a automatic way, as manual alteration and extension could be cumbersome and error-prone.

CHAPTER 4

Prototypical Implementation

In the previous Chapter 3, it was described how a model-based deployment and provisioning process of applications could look like. In this chapter, a prototypical implementation of the proposed process is presented, to show the feasibility. Moreover, the prototype is used for evaluation purposes. In the following section, a short overview, about the technologies that were used to implement the prototype, is given.

4.1 Used technologies

For the deployment and provisioning process we used the Eclipse Modeling Framework (EMF[1]) as a basis, as it ships with a lot of tools and extensions that are useful for implementing the prototype.

We created the CMM in Ecore, which is a reference implementation of EMOF. Other extensions of the Eclipse Modelling Framework are perfectly aligned with Ecore. For instance, all model-to-model transformation rules were written in ATL[2] and ATL/EMFTVM[3] was used for in-place transformation, which is an extended version of the ATL compiler. In contrast to conventional ATLAS transformation rules, with in-place transformations all elements from the source model, which are not captured by a rule, are copied to the output model without further modifications. For model to text transformations, Acceleo[4] is used, which offers the functionality to convert a model to a textual representation. So-called templates contain static sections and dynamic sections that are filled with data from the model.

Part of EMF is the Xtext[5] framework, which can be used to define own domain specific languages (DSL). Xtext describes the grammar of the DSL and how an Ecore model is represented

[1] http://eclipse.org/modeling/emf/
[2] https://eclipse.org/atl/
[3] http://wiki.eclipse.org/ATL/EMFTVM
[4] https://eclipse.org/acceleo/
[5] http://www.eclipse.org/Xtext/

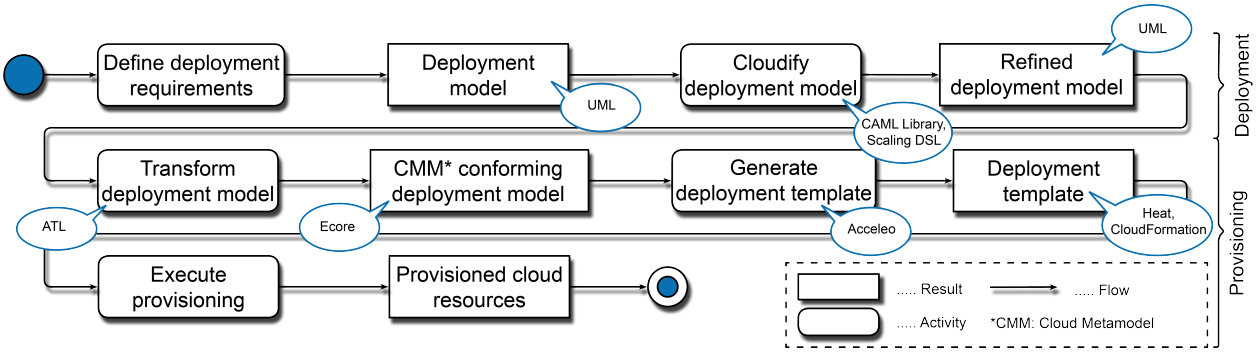

Figure 4.1: Deployment and Provisioning Process

in a textual way. We used Xtext to define our own scaling DSL, to express scaling behaviour and scaling parameters in a descriptive and textual manner.

For the implementation of the process of enriching UML profiles, a Node.js[6] script was used, to extract information from cloud provider APIs and other sources. The enrichment of the UML profiles was implemented as a Java program that uses libraries of the EMF, which facilitates the adaption and extension of UML models programmatically.

4.2 Implementation of Deployment and Provisioning Process

To make it easier to follow the implementation explanations of each step of the deployment and provisioning process, Figure 4.1 illustrates the process from the previous chapter with additional information about used technologies in each step.

4.2.1 Deployment Process

The deployment sub process mainly consists of model refinement operations. For the prototype, we used the CAML library [8], which is a UML internal language based on the concepts provided by the CMM, as we think that application modelers should use tools and environments they are already familiar with, like the UML.

4.2.1.1 Model refinement

The refinement activity of the deployment process is about the transition from a PIM to a PSM through an arbitrary amount of iterations. The application modeler imports profiles provided by CAML in her deployment model, applies stereotypes to model elements and configures slot values of stereotypes. Obviously this step is done manually and hardly can be automated.

An additional feature are validation rules, which can be defined for specific model elements. Figure 4.2 shows an example of an OCL constraint applied to the stereotype *InstanceType*. With

[6] `http://nodejs.org/`

such constraints, it is possible to validate the model before a transformation takes place. In this particular case, it is checked if the selected availability zone is within the selected region.

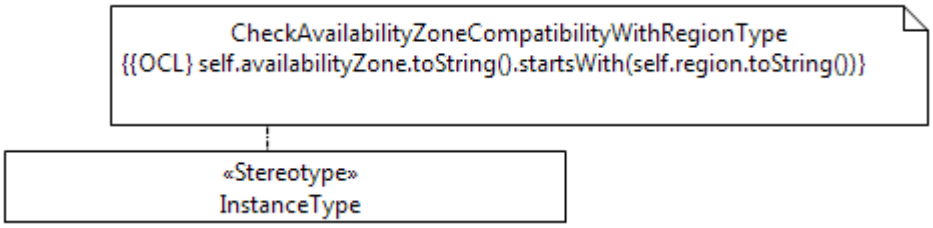

Figure 4.2: OCL constraint used for Model Validation

4.2.1.2 Scaling Rules DSL

As already mentioned, we used Xtext to define our own DSL to express scaling behaviour in a textual way. Rules about scaling behaviour are hard to be modeled as slot values of applied stereotypes. The DSL grammar defines how scaling rules have to be described in a textual way and at the same time facilitates the persistence in form of an Ecore model. Xtext automatically reflects any textual changes in the underlying model and vice-versa. This allows us to integrate an inline editor in Papyrus, which offers syntax highlighting and auto completion according to our Xtext grammar. Furthermore, as the information is stored in form of a model, which conforms to a metamodel, it easily can be used for a model transformation.

We analysed possible scaling configurations from Amazon AWS, OpenStack and Cloudify to consider. On an IaaS abstraction layer, scaling rules are defined in the scope of a virtual instance. For instance, average CPU utilization, outgoing network traffic or disk read operations are common scaling statistics.

Figure 4.3 illustrates an example of a definition of a scaling rule for a virtual node. We decided to align the Xtext grammar with the natural language, to make the rule more readable and understandable. Each scaling rule has a name that should be unique among other rules for the particular virtual instance. Because of the previous mentioned differences between application centric and instance centric scaling rule definitions, we decided to separate them as well in the Xtext grammar. Keywords are formatted as bold green text, whereas variables and values are displayed in a blue font. General configurations that are not rule specific, are annotated in the *config* block.

Listing 4.1 shows the instance scaling part of the scaling DSL grammar. Multiple rules can be defined for one instance, each consisting out of instance specific and/or application specific scaling behaviour definitions. The general scaling configuration is defined once for all rules.

4.2.1.3 Model to Model transformation

After the model has been refined with sufficient cloud specific information, the model has to be converted into a model that conforms to the CMM. Figure 4.4 shows an example of how slot values of stereotypes applied to the cloud node *CN*, are extracted by an ATL transformation rule.

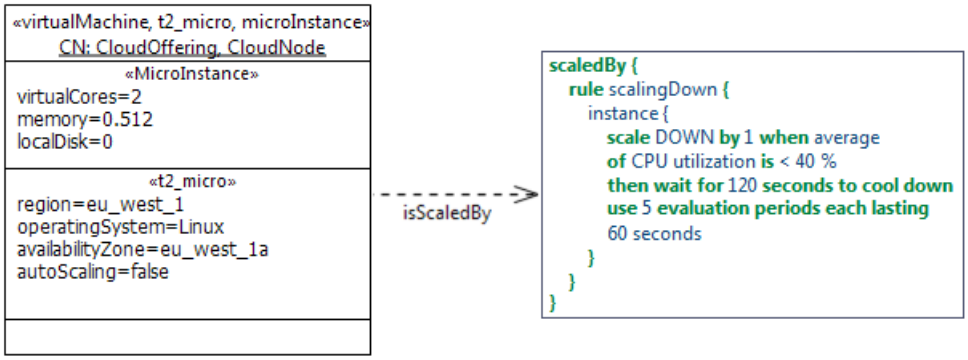

Figure 4.3: Scaling Rule Definition

Listing 4.1: Excerpt of the Scaling DSL Grammar

```
1  grammar ac.at.tuwien.big.dsl.ScalingDsl with org.eclipse.xtext.common.Terminals
2  generate scalingDsl 'http://www.at.ac/tuwien/big/dsl/ScalingDsl'
3
4  ScalingRules:
5          'scaledBy {' rules+=(Rule)* config=ScalingConfiguration '}'
6  ;
7  Rule:
8          'rule ' ruleName=ID ' {'
9                  ('instance {' instanceScale=InstanceScale '}')?
10                 ('application {' applicationScale+=ApplicationScale '}')?
11         '}'
12 ;
13 InstanceScale:
14         'scale ' scaleAction=ScalingActionType ' by ' instanceScaleAmount=INT
15         'when ' statistic=ScalingStatisticType ' of '  criteria=InstanceScalingCriteriaType
16         ' is ' comparisonOperator=ComparisonOperatorType threshold=INT unit=InstanceUnitType
17         'then wait for ' coolDown=INT ' seconds to cool down' 'use ' periodAmount=INT
18         ' evaluation periods each lasting ' periodTime=INT ' seconds'
19 ;
```

The information is persisted in a model, which conforms to a structure that was defined in the CMM.

With a model that conforms to the CMM, the deployment sub process is finished and the subsequent provisioning process is ready to be executed, taking the CMM conforming model as an input.

4.2.2 Provisioning Process

The aim of this step is to use the CMM conforming model and turn it into action. This can only be achieved through model to text transformations, as a model hardly can be interpreted by a provisioning engine or a cloud provider.

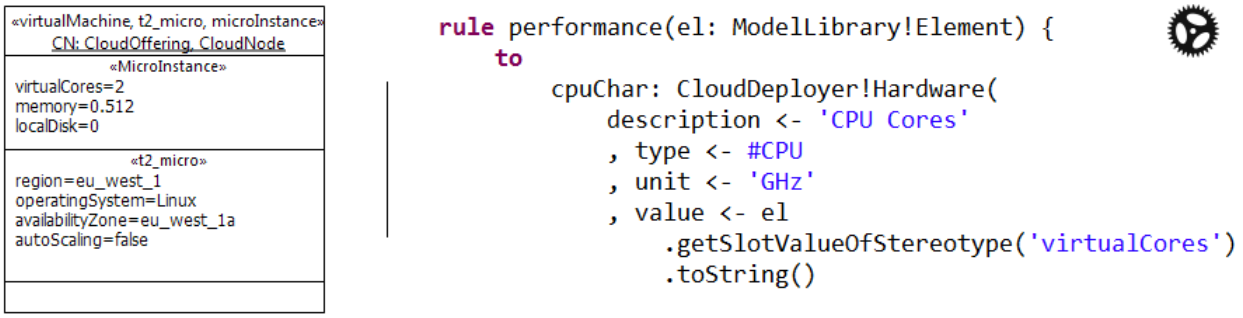

Figure 4.4: Model to Model Transformation

4.2.2.1 Preliminary Considerations

Making the right decision concerning the target format is crucial, as for each format separate transformation rules have to be generated. We evaluated open and/or best practise standards, if they would be suitable as a target format.

Although OVF is an official standard, there is a lack of support and integration among cloud providers. With Amazon AWS, which is one of the most popular providers on the IaaS level, it is not possible to use OVF templates or OVA archives to import or export virtual machines. Rather, they try to push their own implementations and make them defacto-standards (see Section 6.1.3.1). Moreover, OVF does not support the specification of the required software stack, which however is essential for a unified description of both, hardware and software.

We evaluated DeltaCloud (described in Section 6.1.2) quite a while, as the support of OVF sounded promising. Unfortunately it turned out that beside of a Git commit[7] of a DeltaCloud community member, which supports OVF for VMware vSphere[8], there was no OVF support.

For this reason, we discarded the idea of converting our models into OVF format, as even through a unified API such as DeltaCloud, it was not possible to use OVF. We decided to use three different target formats: Heat from OpenStack, CloudFormation from Amazon AWS (both described in Section 6.1.3) and a plain JSON file that is interpretable by our own provisioning engine.

4.2.2.2 Model to Text transformation

Heat and CloudFormation are orchestration interfaces of the respective cloud providers, which can interpret textual descriptions and configurations of desired cloud resources (called templates) and provision them. Figure 4.5 shows the relation between the cloud node *CN* and how an interpretable representation should look like. In this case, CloudFormation is used as a target format. Beneath the model element, the respective model transformation rule is shown.

[7] https://github.com/dkoper/deltacloud-core/commit/69d7d6f70169af07a1b73
[8] http://www.vmware.com/products/vsphere/

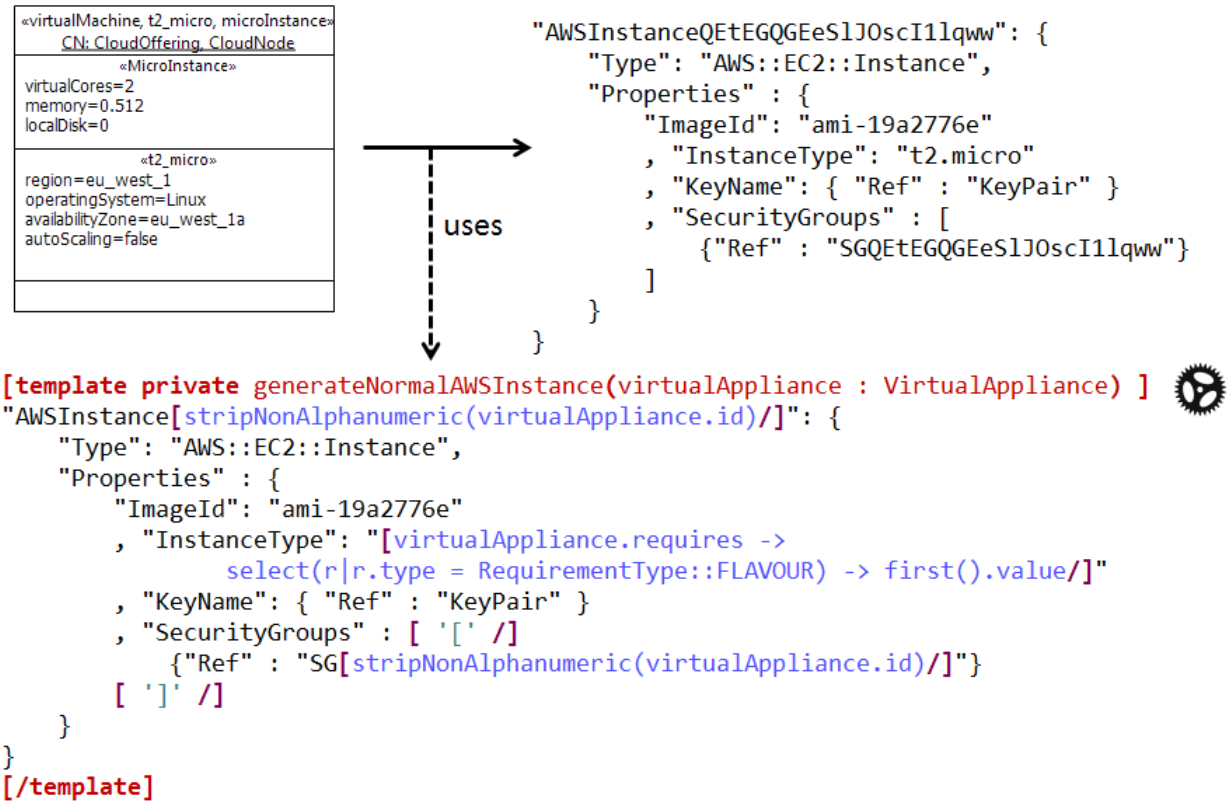

Figure 4.5: Model to Text Transformation with CloudFormation as the Target Format

The figure probably leads to the assumption that the UML element gets directly converted into an interpretable representation, but this is not the case. Just for a better comprehension we decided to leave out the CMM conforming model in this figure.

The Acceleo template extracts data from the model, to define the instance type, the security groups and a unique identifier to create a resource name. Some data, such as the image ID, is statically defined, as the manifestation of the rules depend on the chosen image strategy as well.

4.2.2.3 Provisioning Engine

Concerning the provision engine, we evaluated various approaches with different complexity levels:

Cloud resource deployer. As described in Chapter 6.1.2, there exists a few number of libraries, which try to unify heterogenous cloud provider APIs into one consistent one. Our first idea was to transform our deployment model into JSON notation, as it is lightweight and easy to read for humans (see sample output in listing 4.2).

Listing 4.2: Sample Output of Textual Model Used by Own Deployer

```
{
  "virtualInstances": [{
    "name": "ServerA",
    "HWProperties": [{
      "description": "Working memory",
      "type": "MEMORY",
      "unit": "GB",
      "value": "0.4"
    }],
    "Requirements": [{
      "type": "REGION",
      "value": "eu-west-1"
    }, {
      "type": "AVAILABILITY_ZONE",
      "value": "eu-west-1b"
    }, {
      "type": "FLAVOUR",
      "value": "t1.micro"
    }]
  }]
}
```

We build a small JAVA application, which parsed the JSON file and used jClouds to deploy our cloud resources to one of the supported cloud providers. One of the biggest drawbacks of libraries, which unify proprietary APIs, is that they mainly implement functionality, which is offered by all cloud providers. This means that we were able to provision cloud resources and do basic configuration, but we soon reached the boundaries of the library, especially when more complex configuration was necessary.

Cloud Resources Orchestration Interface. To add more functionality to our deployment models, we decided to create interpretable representations of our models in the CloudFormation format or Heat format. In this case, the provisioning engine part of the process is optional, as the file can be uploaded to the web backend of Amazon or OpenStack. Nevertheless for a more convenient way of testing and evaluating, we implemented a Node.js script, which connects to the API of the orchestration module, uploads the template and initiates the creation automatically.

Dynamic variables such as IP addresses, operating ports or DNS names can not be predetermined and are a result of the provisioning process. Fortunately with CloudFormation and Heat, it was possible to define special output parameters, which were of interest in regards to the deployment model. A post-deployment routine in form of a small JAVA programm was implemented, to feed the information back into the deployment model.

Figure 4.6 depicts an example of a deployed RDS database in the Amazon cloud, with deployment information such as operating port and public DNS name. In this way the model can be used for documentation purposes as well.

To be able to assign deployment information to its respective modelling component, we used the *xmiID*, which is unique for each element within a model. We are aware that during model transformations this id can change, but in our case no model alteration happens during the provisioning process.

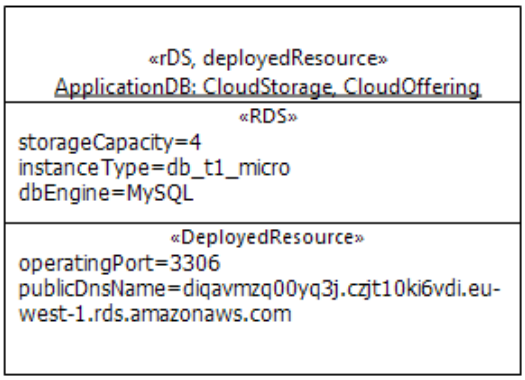

Figure 4.6: Deployed RDS Database with Deployment Information

4.3 Implementation of UML Profile Enrichment Process

To the best of our knowledge, information about instance types or availability zones can not be extracted by using a unified API such as jClouds, but for each cloud provider its own proprietary API has to be used. We mostly used the REST interface of the APIs, which meant that we had to build all HTTP request from scratch, in order to retrieve the required information. For this purpose, we decided to use a Node.js script, as it is lightweight, relies on clear asynchronous programming patterns and supports JSON out of the box.

4.3.1 Solution

Figure 4.7 has already been discussed in the previous chapter, but for a better understanding we decided to list the process once again, as the explanation of the solution will refer to each step of the process. Furthermore, we added additional information about used technologies in each step.

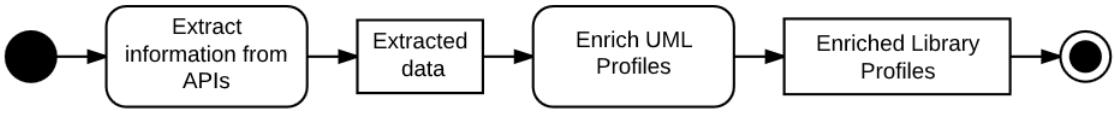

Figure 4.7: UML Profile Enrichment

Extract Information. The first step is about using proprietary cloud provider APIs to extract useful information that can be integrated into the UML library. In the prototype implementation a Node.js script connects itself to three cloud providers: Amazon AWS[9], Rackspace[10] and

[9] http://aws.amazon.com/
[10] http://www.rackspace.com/

TryStack[11] and saves the result in JSON notation to a file (see listing 4.3 and 4.4 for sample output).

Listing 4.3 shows a sample output for Amazon AWS, which contains information about regions and their respective availability zones and hourly prices depending on the region (in general, instances hosted in the USA are cheaper), the operating system (for Windows instances there may incur licence fees) and the instance type.

Listing 4.3: Sample Output of the Information Extraction Script: Regions and Prices

```
"aws": {
    "regions": {
        "eu-west-1": {
            "regionName": "eu-west-1",
            "zones": [
                "eu-west-1a",
                "eu-west-1b",
                "eu-west-1c"
            ]
        }
    },
    "prices": {
        "eu-west-1": {
            "Linux": {
                "m1.small": [{
                    "price": "0.065",
                    "currency": "USD"
                }]
            },
            "Windows": { ... },
        }
    }
}
```

In listing 4.4, available flavours at TryStack and Rackspace are shown. For hardware requirements matching it is important to know that only predefined instance type configurations are available and certain discrepancies cannot be avoided. The most suitable flavour has to be selected among others.

Enrich UML Profiles. The extracted data is used in the final step as an input to enrich the UML profiles with new or updated data. This is done through a Java programm, which parses the JSON file and modifies the UML profile with the help of libraries provided by the EMF.

As an example, Figure 4.8 shows a UML enumeration and an instance flavour, which has been created automatically based on the extracted data that were received through the API. This information can be helpful in various ways, such as (semi-) automatical requirements matching (regarding to costs or characteristics) or improvement of overall expressiveness of the model.

[11] http://trystack.org/

Listing 4.4: Sample Output of the Information Extraction Script: Flavours

```
"trystack": {
    "flavors": {
        "m1.small": {
            "name": "m1.small",
            "memory": 2048,
            "virtualCores": 1,
            "localDisk": 20
        }
    }
},
"rackspace": {
    "flavors": {
        "512MB Standard Instance": {
            "name": "512MB Standard Instance",
            "memory": 512,
            "virtualCores": 1,
            "localDisk": 20
        }
    }
}
```

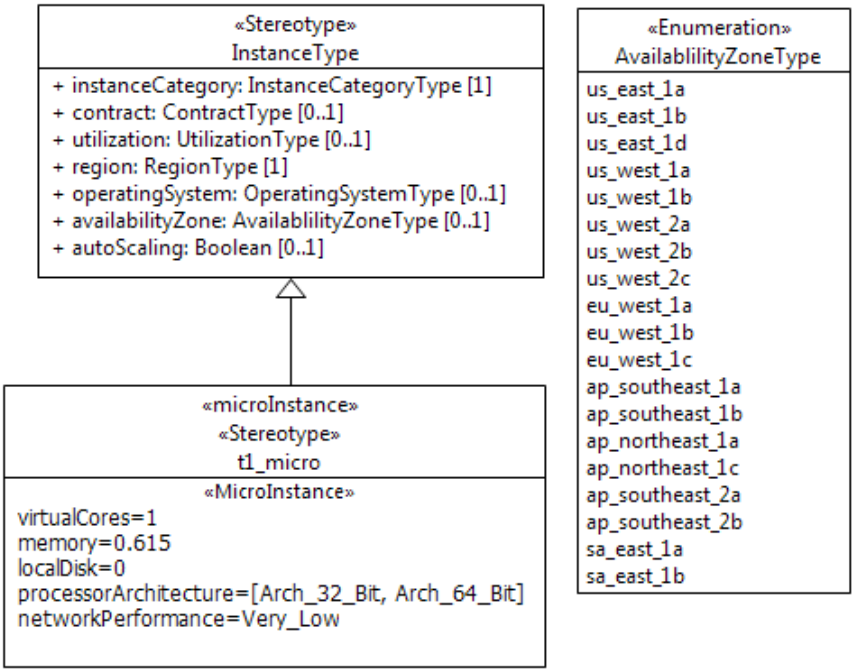

Figure 4.8: Extracted Availability Zones and Instance Type from Amazon AWS

CHAPTER 5

Evaluation

In this section, the expressiveness and the applicability of the presented solution is evaluated.

5.1 Blueprints

In order to facilitate the start of the modelling process, we created best practise blueprints of cloud resource orchestrations. These blueprints can be used as a basic scaffold, which later can be altered or extended. To provide blueprints, which correspond to real-world deployment scenarios, we used reference architectures from Amazon[1] and recommendations from Rackspace[2]. Although they are originally described in a way to promote their own cloud products, we generalised them and introduced a generic terminology, which means that the blueprints can be used independently of the target cloud provider. Nevertheless as cloud computing is a fast moving environment some services may not be available for all cloud providers and replacing the service with alternative cloud resources may be necessary.

Blueprints for various scenarios and application domains, such as web application hosting, online games or e-commerce applications, have been created. We identified four main categories in which all cloud resources can fit into: *Computing and Networking Service*, *Utility Service*, *Data Processing Service* and *Storage and Content Delivery Service*. Computing and Networking Service contains cloud resources such as load balancers, DNS services or virtual machines. We categorised email gateway, monitor service and identify service as Utility Service. Under Data Processing Service fall service like search engine service, map reduce service, workflow service and queue service. The last category Storage and Content Delivery Service is all about providing data to applications and store them safe, secure and efficient. It contains cache service, CDN (content distribution network) service, block storage service, object storage service and database service.

[1] `http://aws.amazon.com/architecture/`

[2] `http://www.rackspace.com/knowledge_center/article/rackspace-open-cloud-reference-architecture`

In table 5.2 a summary of how many elements per cloud resource category per blueprint is stated.

	Cloud Resource Categories			
Blueprint name	**Comp. & Network**	**Utility**	**Data Processing**	**Storage and CD**
Advertisement Serving	5	-	1	2
Batch Processing	2	-	2	2
Content And Media Serving	3	-	-	3
E-Commerce Checkout Service	4	1	1	1
E-Commerce Marketing Recommendations	4	1	1	3
E-Commerce Web Frontend	7	-	1	4
File Synchronization Service	3	2	-	2
Financial Services Grid Computing	3	-	1	4
Media Sharing	6	-	3	1
Online Games	3	1	1	3
Web Application Hosting	7	-	-	3
Web Log Analysis	1	-	1	3
Basic Cloud Architecture	1	1	-	2
Content Management System Architecture	2	1	-	2
Reverse Proxy Cloud Architecture	4	-	-	1
Tiered Cloud Configuration	3	1	-	4
Web Application Configuration	7	1	-	4

Table 5.1: Amount of model elements per category per blueprint

5.2 Case study

In the following section, three different applications with different complexity level will be deployed via the process, which has been discussed in this master's thesis. The aim is to investigate the practical applicability of the process in practise, if additional configuration work is necessary and which possible extensions could be subject of further work.

The three applications have the following characteristic:

- **Calendar application**: Requires an application server and a database. The application is provisioned through the provisioning client and CloudFormation. A layered-image strategy is used, where some software is installed after the machine has booted.

- **PetStore**: Requires an application server, a database and a load balancer. The deployment model contains scaling rules and the application will be provisioned through Heat and CloudFormation. As an image strategy, a container approach is used.

- **Ticket Monster**: Requires as well a database, a load balancer and an application server with scaling rules. The performance differences between a pre-baked image strategy and a raw-image strategy in combination with elastic scaling are evaluated. Furthermore, Heat and CloudFormation are the target formats.

In theory everything always looks easier and one does not take all modalities into account. Especially the configuration of application servers and the adaption of the hosted software can be cumbersome, as the devil is in the detail.

5.2.1 Cloud Providers Used For Evaluation

During our evaluation we decided to sign up for an Amazon free-trial account[3], which allows the usage of certain AWS resources for a limited time for free. The reason why we chose Amazon was beside the economical incentives that Amazon is the market leader in IaaS, as already mentioned.

As a second cloud infrastructure, we chose to install and configure OpenStack on our own hardware as a private cloud. The following paragraphs will discuss the used configuration of OpenStack in detail.

Firstly, lets have a look on the network topology, which is illustrated in Figure 5.1, whereas we followed a three-node example architecture described in the OpenStack installation manual[4]. Essentially, there are three isolated physical networks responsible for different purposes: management network, virtual instances network and external network. The management network is used for managing and configuring all three physical nodes. The API of OpenStack, which is exposed by the controller node is accessible through this network. All virtual machines are assigned to a subnet of the virtual instance network. The network node is capable of creating virtual routers and virtual switches, to establish communication paths between them. The network node has also one network interface connected to the external network. Normally virtual instances launched within OpenStack are not reachable from outside. Through public floating IP, a virtual instance can provide services that are accessible from the internet as well. Floating IPs are mapped by the network node to a specific virtual instance within OpenStack.

We used the following services of OpenStack:

- Basic services: Keystone for Authentication, Glance for hosting virtual images that can be used for launching new instances, Nova for managing computing nodes, which provide processing power to start new virtual machines, Horizon for a web based management interface and Neutron for advanced networking capabilities.

- Additional services: Heat for the orchestration of cloud resources, which is capable of parsing Heat templates (similar to CloudFormation)

As an operating system CentOS 6.5 was installed on all three servers. Furthermore, the following hardware was used:

For evaluation, all three provisioning strategies that were implemented and integrated in the process and which were already mentioned in 4.2.2.3 are considered. In some examples, not all strategies could be used, wether because of compatibility problems or technical reasons.

[3] `http://aws.amazon.com/free/`

[4] `http://docs.openstack.org/icehouse/install-guide/install/yum/content/ch_overview.html#architecture_example-architectures`

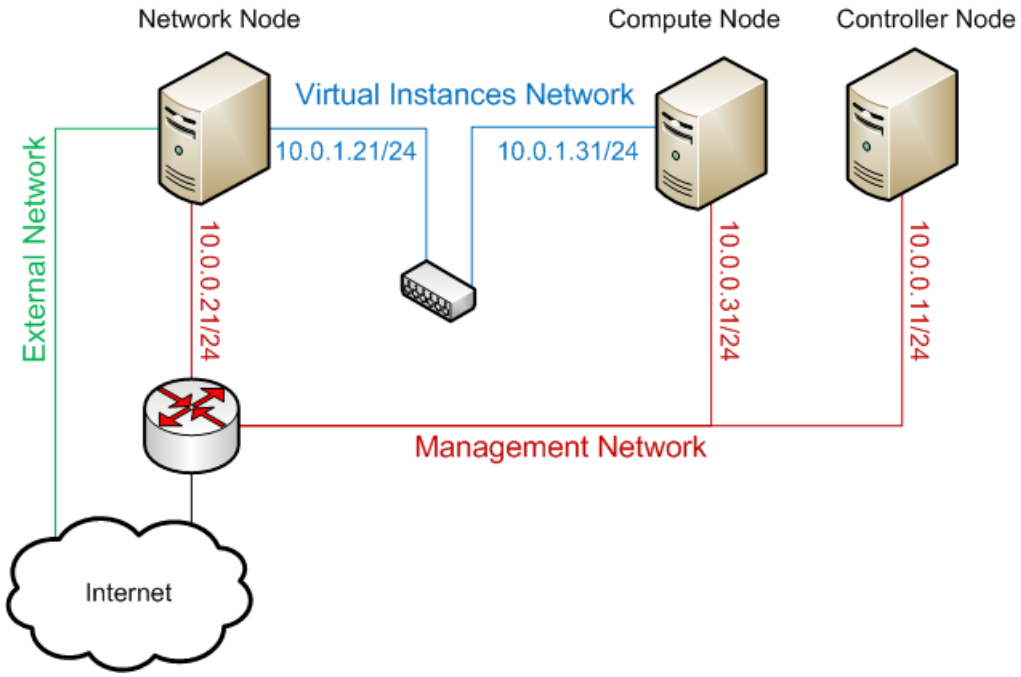

Figure 5.1: Network Topology for the Private OpenStack Cloud

	Hardware specifications			
Node	**CPU**	**Working memory**	**HDD**	**Network**
Network Node	Intel Xeon DP E5440 @ 4x 2.83GHz	16GB	160GB	100Mb/s
Compute Node	Intel Xeon DP X5482 @ 4x 3.20GHz	16GB	250GB	100Mb/s
Controller Node	Intel Xeon DP L5410 @ 4x 2.33GHz	16GB	500GB	100Mb/s

Table 5.2: Hardware Specifications for the OpenStack Environment Provided by an Intel (R) Modular Server

5.2.2 Calender Application

The first example is a calender application, which uses a MySQL database for persisting new events. The deployment requirements are illustrated in Figure 5.2 and consist out of an application server and a database. The process is initiated with the basic modelling of cloud resource requirements, for instance a virtual machine is used for the application server. Furthermore, a generic stereotype *microInstance* is applied to *CalAppServer* and general hardware characteristics such as one virtual core and working memory of 512MB.

As we decided to deploy our application to the AWS cloud, we perform a model refinement that automatically applies AWS specific stereotypes on selected model elements (see Figure 5.3). In this case, the hardware requirements, which were defined in the previous step, are satisfied by a *t1.micro* virtual instance from Amazon and therefore the appropriate stereotype was added. We want the application to be hosted in the western European region in availability zone *1a*. As

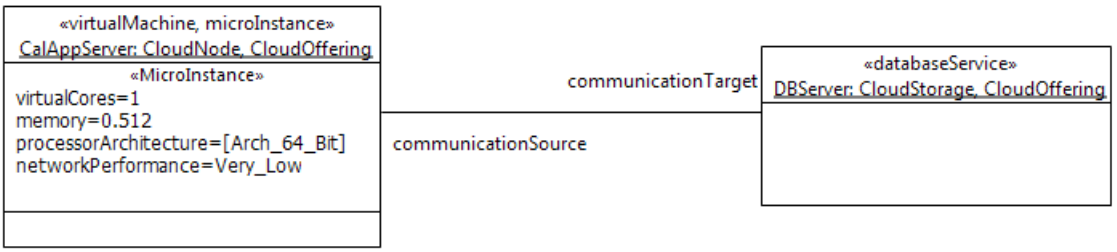

Figure 5.2: General Deployment Requirements for Sample Application #1

we are not using any load balancer, we do not want the application to be scaled. Concerning the database we choose the smallest available instance type and MySQL as the database engine.

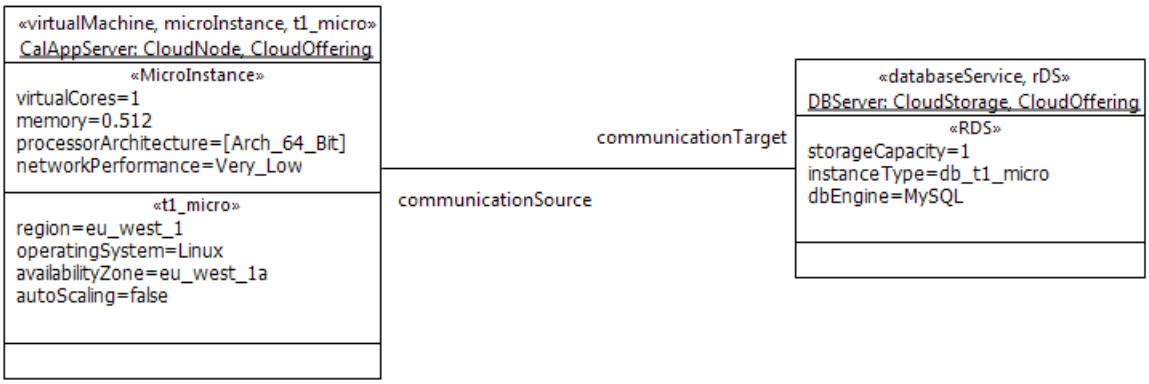

Figure 5.3: AWS Specific Deployment Requirements for Sample Application #1

Apart from the fact that Linux is used as the operating system, an operating environment is defined to ensure the execution of the calender application (see Figure 5.4). Each operating environment consists of a deployment target, which in our case is Apache Tomcat6. As already mentioned in Chapter 3, the modelling library provides predefined execution stacks with all their dependencies of other software and services. Those dependencies Apache Tomcat6 is relying on, will be taken into account in the next transformation step. In this example we are following a layered approach, as additional software is not baked into the image, but is installed afterwards.

In the next step the model is transformed into an interpretable representation. We are going to use our own provisioning client and CloudFormation and will discuss the results separately. Heat can not be used, as Amazon AWS does not offer an interface for Heat.

5.2.2.1 Provisioning Client

As our self-implemented provisioning client relies on jClouds, the functionality is limited. Apart from the fact that creating a database are not supported by jClouds at all, the creation

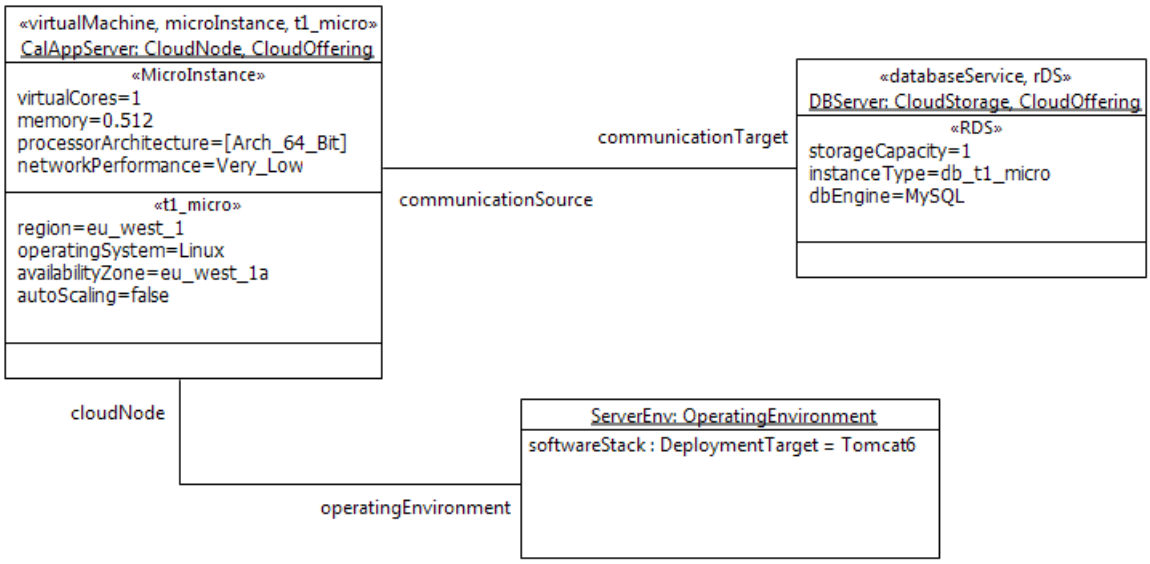

Figure 5.4: Operating Environment Requirements for Sample Application #1

of *CalAppServer* involved some troubles as well. The provisioning clients parses the JSON configuration and concatenates all hardware requirements and other requirements, such as the region. Firstly, only the region can be specified, but no the availability zone within a region. Furthermore, the library is pretty inflexible concerning requirements matching, as it return choices that are matching exclusively all requirements. Concerning the operating system type, we could determine some inconsistencies, as defining *Linux* as an operating system, systems with *Ubuntu* were not listed. In the case of Amazon, the version number was not used to specify a ceratin release such as *Ubuntu 14.04*, but was used to specify the date the image has been built. This implies at the same time that although jClouds provides the ability to match by version strings, it is not practicable nor intuitive why Amazon uses the build date instead of the version number. The consequence is that the operating system can not be further specified.

Concerning the hardware requirements, jClouds can be used to do basic requirements matching and provisioning. This is limited to virtual instances, as additional cloud resources such as a database can not be created. In the case of required software and dependencies that have to be installed on the virtual machine, jClouds lacks of support. To the best of our knowledge, jClouds does not provide a common way of installing software, abstracting away the differences and idiosyncrasies of an operating system, apart from executing shell scripts through jClouds. As discussed in Section 7.3, jClouds integrates well with Chef Opscode though.

5.2.2.2 CloudFormation

The other strategy we evaluate with the Calendar application example, is the provisioning through CloudFormation from AWS. The functionality and possibilities to define constraints and express yourself is broader. Compared to the provisioning client, availability zone and specifications

concerning the services to be installed can be annotated. With CloudFormation, we were able to provision the virtual instance, as well as the MySQL database. Figure 5.5 illustrates the end result (for instance public DNS names for resources), embedded in the deployment model.

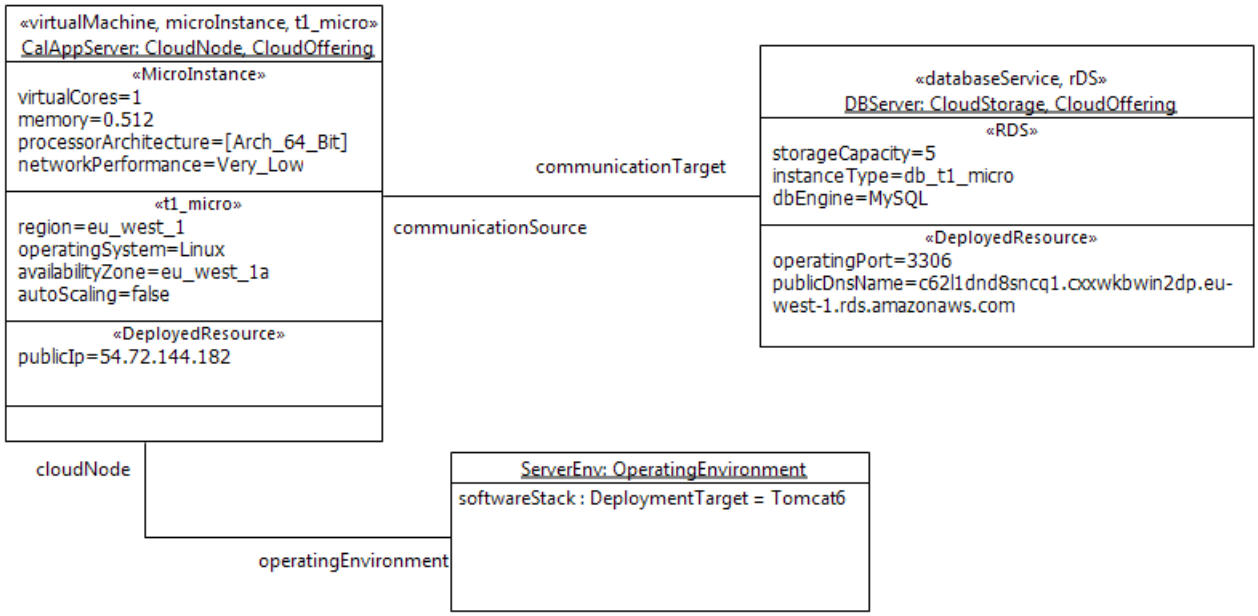

Figure 5.5: Provisioned Cloud Resources for Sample Application #1

Although software dependencies can be specified, the solution is still not flexible enough as it has to be predefined through which package manager the installation should take place. For instance, the Debian distribution uses *apt-get* to install new software packages, whereas rpm-based versions of Linux such as CentOS use *yum*. We used for this example as an operating system Amazon Linux, which has been tailored for the usage on AWS infrastructure. As the calendar application archive was publicly available, we added manually an instruction to copy the war-archive, during provisioning, into the respective application folder, so that it can be served instantly after the machine is running. Apart from that, we also had to manually configure the database settings of the application, such as IP address, username and password, so that the application could connect to the database.

5.2.3 PetStore Application

The second example used for evaluation was the PetStore application, which has already been introduced in previous chapters. Figure 5.6 shows the general deployment requirements the applications has. A load balancer is used to distribute incoming requests. The request load of the application can not be determined in advance. For this purpose scaling rules for virtual instances have to be defined in further steps. These rules are used, to perform elastic scaling automatically. The third element is a database that is used for persistence purposes.

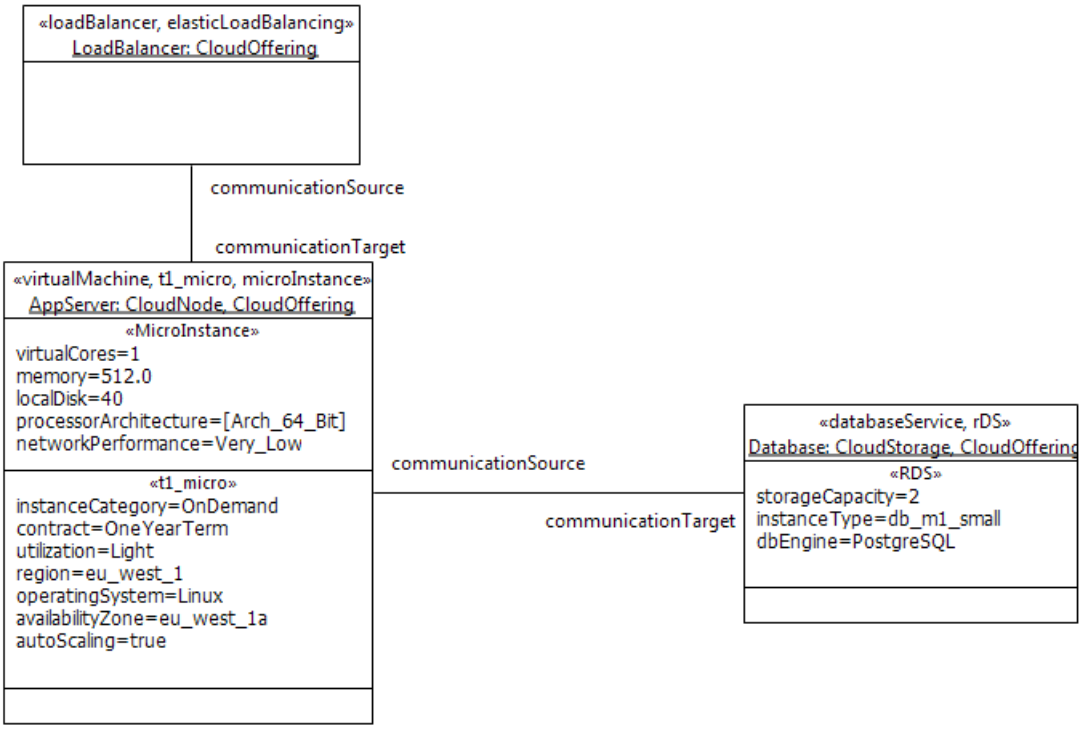

Figure 5.6: General Deployment Requirements for Sample Application #2

As already described in the first evaluation example, the next step consists out of the definition of hardware requirements and configuration parameters. As in this example, a load balancer combined with scaling rules are used, Figure 5.7 depicts the scaling configuration we used for the application server. We defined two scaling rules, both on the instance level. The first rule *scalingDown* defines the behaviour, when the average CPU utilization of all running instances is lower than 40%. In this case one of the machines will be terminated (=scale down). To calculate the average CPU usage statistic, a sample lasting 60 seconds is taken 5 times. Out of this data it is determined if the scaling has to be performed or not. To prevent a bias within the samples, after a scaling has taken place, the scaling engine will be inactive for 120 seconds, before it returns to take samples. The rules for scaling up looks similar. The only difference is that once a threshold of 80% CPU utilization is reached, a new instance will be launched. In the general configuration section we define settings that are related to all scaling rules. In this example we want to launch two instances at the very beginning, have at least one instance and at a maximum five instance running at the same time.

The first example showed that the abilities while modelling a software stack with the library is limited. Furthermore, using a layered approach, where additional software has to be installed after the machine has booted, in combination with a load balancer and scaling rules, could be problem. Let us assume that there is a temporary request peek and the scaling engine decides to scale up and trigger the creating of a new virtual instance. If the creation of a new instance needs

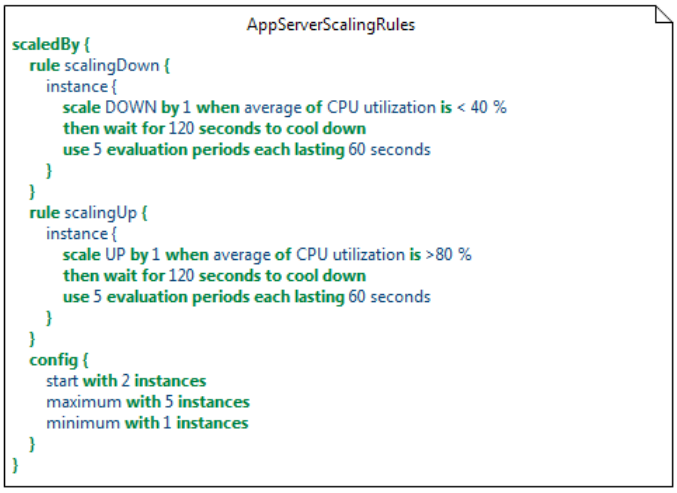

Figure 5.7: Scaling Rules For the Application Server

more time than the period of time the additional instance is needed, the idea of elastic scaling is taken ad absurdum.

For that reason, we use a container approach in this example. The image, the virtual instance are booted from, is provided by Bitnami, which offers a huge variety of pre-configured virtual images that can be used for local or cloud installations. We retrieved an image that contained Ubuntu and JBoss and all its dependencies from the Bitnami website[5]. For our private OpenStack installation, we downloaded the image, converted it into a flatten single vmdk-image and uploaded it through the image service of OpenStack *glance* to the image registry. In the case of Amazon, the image was available through their image repository and could be referenced by its ID.

During our preliminaries it turned out that the image we used for OpenStack did not have *cloud-init*[6] installed, which is capable of configuring a cloud instance at an early stage, such as injecting SSH keys for public/private key authentication. For this example, we used two formats the model was transformed to: CloudFormation and Heat. As we did not model the software stack in our model, settings for security groups and port forwarding information for the load balancer were not available. For that reason, we manually added certain firewall rules to the Ecore model, so that they could be considered during the model to text transformation. Our experiences with both formats, will be described in the next two sections.

5.2.3.1 CloudFormation

The second example used for evaluation is more complex, which means that the CloudFormation template could not be created completely automatically. As already mentioned, we used an image from Bitnami, which contained a pre-configured JBoss application server. All Bitnami

[5] `https://bitnami.com/stack/jboss`
[6] `http://cloudinit.readthedocs.org/en/latest/`

images can also be accessed via the image repository of AWS and are identified by a unique AMI ID, which had to be inserted into the CloudFormation template manually. We did not model any software stack requirements, as a pre-configured image with an application server and all dependencies was used. As a consequence the firewall rules for incoming traffic for the load balancer and the virtual machines could not be configured automatically and some definitions had to be configured manually.

In order to run the PetStore application and to be able to communicate with the database, the JBoss server has to be configured. JBoss manages database access through data sources, defined in an XML configuration file. As new application server instances are created automatically (according to the workload following the defined scaling rules), the configuration of the JBoss server has to reside in the template definition as well as manual configuration after the instance has been launched takes the idea of elastic load balancing ad absurdum.

Listing 5.1 shows the instructions embedded in the CloudFormation template, which are necessary to create a data source and deploy the application. In more detail, two files are created in the home directory of the user *bitnami*. The first file (*/home/bitnami/PetStoreJEE6.war*) is copied from an external source and contains the application and its library in a war archive. The second file (*/home/bitnami/conf*) is created dynamically as information about the MySQL database are included, such as connection URL, operating port and user credentials. It also refers to the first file in order to deploy the application. Apart from creating new files, in the section *UserData*, a custom bash script is defined, which triggers the bootstrapping process after a virtual instance has been started. During this process the previously mentioned files are created. After the instance has been bootstrapped, we call the JBoss CLI script, to execute all commands defined in */home/bitnami/conf*.

Listing 5.1: Configuration of JBoss Server within CloudFormation

```
"files" : {
  "/home/bitnami/PetStoreJEE6.war": {
    "source": "http://web.student.tuwien.ac.at/~e0926741/PetStoreJEE6.war"
  },
  "/home/bitnami/conf" : {
    "content" : { "Fn::Join" : ["", [
      "connect\n",
      "data-source add --name=PetStoreDS --jndi-name=java:jboss/PetStoreDS --user-name=root --
          password=8wkqNMHC --connection-url=jdbc:mysql://",
      {"Fn::GetAtt" : ["iDqEoDjEeSMg6X7n3H9g", "Endpoint.Address"]},":",
      {"Fn::GetAtt" : ["iDqEoDjEeSMg6X7n3H9g", "Endpoint.Port"]},
      "/iDqEoDjEeSMg6X7n3H9g --driver-name=mysql-connector-java-5.1.12-bin.jar\n",
      "data-source enable --name=PetStoreDS\n",
      "deploy /home/bitnami/PetStoreJEE6.war\n"
    ]]},
    "mode"  : "000644",
    "owner" : "bitnami",
    "group" : "bitnami"
  }
},
"UserData": {
  "Fn::Base64" : { "Fn::Join" : ["", [
    "#!/bin/bash -ex\n",
    "/usr/local/bin/cfn-init -s ", { "Ref" : "AWS::StackId" },
    " -r LaunchConfigurationgW8EIDjEeSMg6X7n3H9g --region ", { "Ref" : "AWS::Region" }, "\n",
    "/opt/bitnami/jboss/bin/jboss-cli.sh --file=/home/bitnami/conf\n",
]]}}
```

Amazon provides images of a customised Linux for AWS, which also contains a package called *cfn-init* that is capable of installing new software packages, creating new files (in this

example */home/bitnami/PetStoreJEE6.war* and */home/bitnami/conf*) and and start new services according to the configuration of the CloudFormation template. The image provided by Bitnami did not have *cfn-init* pre-installed and had to be loaded in order to be able to bootstrap the virtual machine. Listing 5.2 shows the necessary steps to retrieve and install the *cfn-init* package.

Listing 5.2: Installation of the Package cfn-init

```
"UserData": {
  "Fn::Base64" : { "Fn::Join" : ["", [
    "#!/bin/bash -ex\n",
    "apt-get update\n",
    "apt-get -y install python-setuptools\n",
    "wget -P /root https://s3.amazonaws.com/cloudformation-examples/aws-cfn-bootstrap-latest.tar.gz
        ","\n",
    "mkdir -p /root/aws-cfn-bootstrap-latest","\n",
    "tar xvfz /root/aws-cfn-bootstrap-latest.tar.gz --strip-components=1 -C /root/aws-cfn-bootstrap
        -latest","\n",
    "easy_install /root/aws-cfn-bootstrap-latest/","\n"
]]}}
```

The creation of all cloud resources took about 10 minutes and another 5 minutes to register the virtual machine in the load balancer pool. This strongly depends on how the health check of virtual instances is configured. The load balancer checks every five seconds if the virtual machine is still alive, and retrieves current CPU usage, network load and other metrics. We set the (un)healthy threshold to 2, which means a virtual instance is considered as (un)healthy as soon as 2 health checks fail/succeed. With this configuration, we decreased the response time of the load balancer to a minimum.

As stated previously, we defined two scaling rules, which use CPU usage statistics as a basis to decide if scaling has to be performed. To simulate workload and be able to evaluate the functionality of the defined rules, we connected to the first virtual machine via SSH, and created 6 detached processes all executing the *yes* command writing any result to */dev/null*, like so: *yes > /dev/null &*. This resulted into a CPU utilisation of 100% and after 5 minutes (5 evaluating periods of 60 seconds) a new instance was spawned.

5.2.3.2 Heat

The deployment of the PetStore application in our private OpenStack cloud required some preliminary work to be done, as we used the image from Bitnami that was designated for virtual machine players, such as VMWare or VirtualBox. As the image was split into various files, we flatten it and converted into the *.vmdk* format that is readable by Glance, the image service of OpenStack. After this step, we added the image to the registry by executing the following command on the console:

Listing 5.3: Adding a New Image to the Glance Image Repository

```
glance image-create --name "JBoss Application Server" --disk-format vmdk  --container-format bare
    --is-public True --progress < jboss_7_11_bitnami.vmdk
```

We did not consider that the previously mentioned image did not contain the required software package *cloud-init* for injecting SSH keys and creating files during boot time. For that reason we abandoned the idea of using the pre-configured image from Bitnami and created our

own. The process was straightforward: After launching a new instance with a plain CentOS 7 installation, we installed the JBoss application server manually. After finishing the configuration step, we made a snapshot of the running machine and used that image for the virtual machines defined in the heat template.

Although the API of the orchestration platform of OpenStack is based on CloudFormation and still can interpret templates in the CloudFormation syntax, Heat introduces new concepts with every new release of OpenStack. For instance, Heat is not limited to the scaling of virtual instances, but every cloud resource can be considered for scaling. Furthermore, a load balancer in Heat is a plain proxy that forwards all request to a pool of virtual instances. The definition of the pool itself contains all required information, such as the load balancing method a monitor that test the availability of each registered pool member. Amazon AWS assigns by default public ip addresses to every virtual resource that is created, in OpenStack we had to define IP address assignments explicitly.

One of the biggest differences is that Heat provides the functionality of defining own resource types that can be used for instance for the purpose of auto scaling. This is achieved by creating sub-templates, which can be deployed as stand-alone templates as well and are embedded in the main template.

Similar to CloudFormation, we created custom bash scripts that are executed during boot time to create a data store for JBoss and to deploy the application automatically. As HOT uses the YAML syntax it, the created template is easier to read and looks cleaner. Instead of brackets and commas, indentations are used to structure the content of the template. Whereas incorrect usage of indentation leads to a parsing error, model to text transformations are easier, as closing tags, quotation marks or similar are not necessary.

As we did not use the OpenStack Trove (Database as a Service), we configured the application to reuse the database that was created with CloudFormation. According to our observations, with OpenStack virtual instance got recognised as healthy instance, to which traffic can be forward to, faster and the registering process did not last as long as in the case of AWS. In order to simulate a heavy workload and to trigger a scaling operation, we proceeded with the same strategy of spanning new processes as we did in the case of CloudFormation. With the telemetry service of OpenStack *ceilometer* we could reconstruct in more detail when and why a scale-in or a scale-out happened, which has not always been the case with AWS.

5.2.3.3 Observed Issues and Challenges From an Application Perspective.

Application Data Management. The PetStore application was taken as an example for the proof of concept and was not further optimised in relation to cloud specific data management. The application uses Hibernate as an ORM mapper and populates the necessary database structure automatically upon deployment. In combination with elastic scaling and the automated creation of new instances this would have been suboptimal. For that reason, we decided to initialise the database once and ensured that the application did not change the database structure upon deployment.

Application Container Configuration. Another issue we stumbled over, was the session handling among multiple application server. For instance, if the authentication is forwarded by the

load balancer to server A, but further requests are handled by server B, the user would have to re-authenticate. One solution on the load balancer level, would be a cookie stickiness, which means the load balancer keeps record which server issued which cookies and forwards further request to the appropriate server. Another solution would affect the configuration of all application servers, which would be grouped in a cluster, knowing that they are operating in a cloud environment. Nevertheless this issues are out of scope of this master's thesis.

5.2.4 JBoss Ticket Monster

The Ticket Monster application is a reference implementation that uses state of the art technologies developed by the creators of the JBoss application server. It was designed in a way to be runnable on cloud infrastructure and the data storage can be in-memory or any arbitrary database. The deployment and provisioning process starts as usual with the definition of requirements, as shown in Figure 5.8. The virtual machine *ComputeNode* contains a JBoss application server as an operating environment, which deploys the Ticket Monster application. If the chain is viewed from the other end, a dependency graph can be derived: because the Ticket Monster application requires a JBoss application server, the virtual machine has to provide an appropriate operating environment. The virtual machine should use *CentOS* as the operating system, combined with an instance flavour of *m1.medium*. Apart from the virtual machine, two further cloud resources are required: a load balancer and a database server that defines requirements such as storage capacity or database engine. Detailed specifications about scaling behaviour are modelled in subsequent steps.

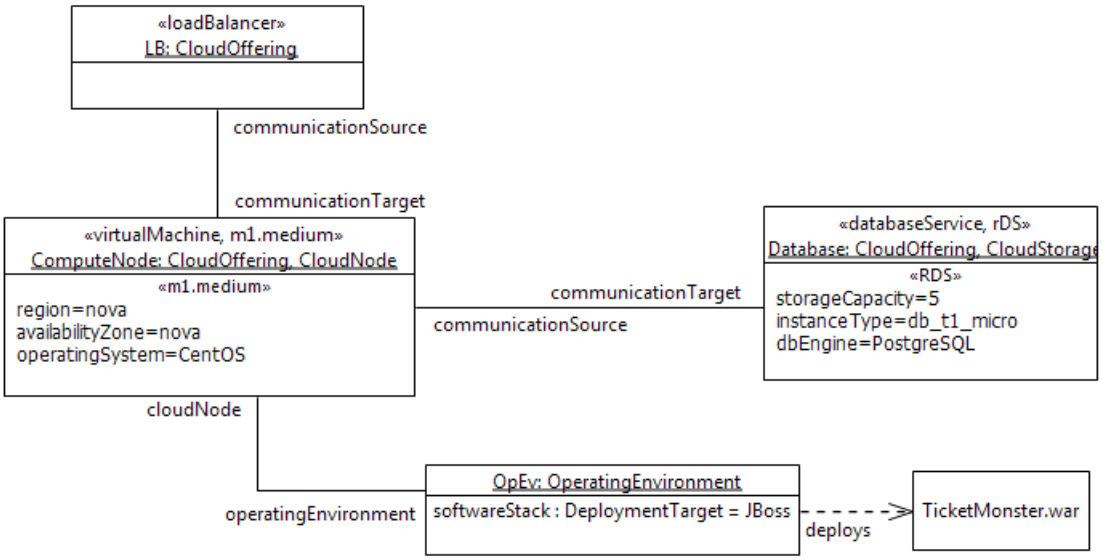

Figure 5.8: Deployment Requirements of Required Cloud Resources for the Ticket Monster Application

One of the interesting questions is, which image strategy should be used, as various ap-

proaches are possible. The model illustrates the dependencies and requirements, and can be used as a decision support. In the following we want to evaluate the raw image strategy and the pre-baked image strategy in combination with load balancing. They are the opposite of each other, whereas a raw image is lightweight and the whole installation workload is done once the machine is booted, a pre-baked image is bigger, as all dependencies and the application itself are already contained, but as soon as the virtual machine has been booted, it is ready to handle incoming requests. The virtual image has to be created and configured beforehand though.

For the third example we used for both image strategies our private cloud based on OpenStack to deploy all cloud resources except for the database, which was created beforehand and externally (in our case in the Amazon AWS cloud). The reason was that we had to configure the application in the pre-baked image before taking a snapshot and using it for the scaled virtual instances.

Elastic Scaling with Pre-baked Image. According to the specified scaling rules, we simulated a high CPU load to trigger a scale-out. As we used a fully configured pre-baked image, the instance was able to serve request instantly after it had booted. One of the main drawbacks is the configuration of the image, as keeping it updated requires additional maintenance time and as soon as the application or one of its dependencies change a new image has to be prepared.

Another issue related to the JBoss server was the fact that sometimes the deployment timeout was too short and therefore the deployment failed. After adapting the server configuration and creating a new image, the problem could be solved.

Overall, the created Heat template was less complex, as no configuration scripts had to be included (this was already done in a previous step). This means, the resulting template form the model to text transformation could be used without any further modification.

Elastic Scaling with Raw Image. For the raw image, we used CentOS 7 as the operating system, which only contained general software packages. Apart from installing Java and the JBoss application server, which includes downloading a couple of hundred MBs, we had to deploy database drivers in form of jar-archives, as shown in Listing 5.4 (URLs have been minified for a better readability of the template). It basically downloads a MySQL driver and a PostgreSQL driver from a publicly accessible server to a temporary directory and afterwards copies them to the deployment directory of the JBoss application server, which takes care of automatic deployment.

Listing 5.4: Configuration Scripts for *ComputeNode*

```
jBossDownload:
  type: OS::Heat::SoftwareConfig
  properties:
    group: ungrouped
    config: |
      #!/bin/sh
      wget http://red.ht/1eMZTLO -P /var/tmp/jboss-as.tar.gz
      tar -zxf /var/tmp/jboss-as.tar.gz -C /opt

driverDownload:
  type: OS::Heat::SoftwareConfig
  properties:
    group: ungrouped
    config: |
      #!/bin/sh
      wget http://bitly.com/1wFnqvj -P /tmp
      wget http://bit.ly/ZZeKDm -P /tmp
      cp /tmp/mysql-connector-java-5.1.12-bin.jar /opt/jboss-as/standalone/deployments/
      cp /tmp/postgresql-9.1-903.jdbc4.jar /opt/jboss-as/standalone/deployments/
```

It is important to know that this bootstrapping process has to be done every time a new machine is started. The same applies for instances that are created through elastic scaling. Having these limitations in mind, our expectations of a poor scaling behaviour were confirmed. We could observe that the sum of sample taking time and cool-down time was too short for provisioning an additional instance. This triggered the initiation process of another instance, as the used sample statistic (in our case CPU utilization) was still above the defined threshold.

5.3 Results of the Case Study

The presented deployment and provisioning process has been evaluated by using a prototypical implementation of it and by applying the process to three different applications. Table 5.3 gives a short overview about how many elements were necessary to model the required cloud resources and how long the deployment and the provisioning took time.

The amount of model elements does not include elements from profiles or model libraries that were used, but represent the count of elements in the custom model only. Concerning the deployment time, it strongly depends on how familiar one is with the used tools and the model library itself. Quite frankly, it is an advantage to know how transformation rules are written, as modelling is more intuitive and gets more effective as less errors are made. The provisioning time on the other side strongly depends on the modeled cloud resources: For instance, as soon as a relational database is part of the orchestration, we observed an extra time of 8 minutes for the provisioning process.

Application	Deployment time	Provisioning time
Calendar	2 min	6 min
Petstore	3 min	13 min
Ticket Monster	4 min	15 min

Table 5.3: Summary of the evaluation outcomes of the three applications

Feasibility. We proved with the case study and the three applications that the discussed model-based deployment and provisioning process is feasible. We successfully deployed and provisioned the application and its dependencies to two different cloud providers. As already mentioned, the deployment time strongly depends how familiar the application modeler is with the modelling tools and the used modelling library. As the implementation of the process is a prototypical one, an application modeler has to know how the constellation of all cloud resources and the application stack should look like, in order to be able to convert the model into a executable textual representation.

Automation. In some parts, the deployment and provisioning process can be seen as semi-automatical. Especially the deployment subprocess is dominated by manually refining the model with cloud specific concepts such as applying profiles and stereotypes. Apart from blueprints, an application modeler can use as a starting point, the automation of this step would be really hard, as refining strongly depends on the application needs and how the orchestration should look like. As there does not exist a homogenous application, we followed the principle of *convention over configuration*, which means that reasonable defaults are used during model transformations. Nevertheless especially when considering application provisioning, constraints are application specific and, as already discussed in the previous section, sometimes manual tweaks of the orchestration templates are necessary.

Provisioning of software stack. As already discussed in Section 2.4 and illustrated in Figure 2.2 there exist different virtual image strategies, which have an indirect impact on the scaling behaviour and the provisioning of new instances. There are two extremes: The raw image and the pre-baked image approach. According to our evaluations, the raw image approach is not applicable for rapid elastic scaling as the provisioning time of the machine takes too long. Especially in the field of software stack provisioning there exists a potential for extensions, which are discussed in Section 7.3.

CHAPTER 6

Related Work

6.1 Related technologies

This section describes some related technologies and formats. Some of them haven been used for the process that has been developed in this master's thesis.

6.1.1 Template formats

Apart from CloudFormation and Heat, there are other template formats, which help to describe a virtual machine in a textual and interpretable way. This subsection discussed some of them.

6.1.1.1 Open Virtualisation Format (OVF)

This format was originally designed by Dell, HP, IBM, Microsoft, VMware, XenSource and was proposed in 2007 to DMTF as an open standard [20]. With OVF it is possible to describe virtual appliances in a generic way to be as compatible as possible to any cloud provider. Having a glimpse on the structure of an OVF file, it is XML based and consists out of various human readable entries such as virtual nodes or network and disc specifications. Starting from virtual network cards and CPU criteria up to virtual DVD drives, a wide range of hardware specifications can be described with OVF. Referenced files to virtual hard disks or disk images can be packed together within one archive which is called an Open Virtualisation Appliance (OVA) archive. Additionally, server certificates and scripts, which have to be run once the server has been setup, can be included.

The current version of the OVF standard is 2.1.0 [21], which was published in January, 2014. It is important to know that out of the box without custom extensions it is not possible to describe software requirements with OVF, which are on top of the operating system, such as application environments or database management systems. This means, OVF provides a unified way of describing virtual appliances targeting attributes, which are mostly relevant for

the IaaS abstraction layer. Although OVF sounds quite promising, to our best knowledge it is not supported by OpenStack or Amazon.

No matter how good a standard is, the usage has to be justified in terms of functionality and applicability. If major cloud providers refuse to integrate the standard, other solutions have to be found. DMTF also published a specification of a Cloud Infrastructure Management Interface (CIMI) [19], which is also capable of processing OVF files. This means, once cloud providers have implemented CIMI, it would be possible for consumer to communicate in a unified way with each cloud provider.

A comparable approach is the Open Cloud Computing Interface from OGF, which core is described by the authors of [43] and according to their website[1] further specifications are in progress. They propose a protocol and an API, which defines a way of how to interact with the OCCI core through restful HTTP requests. The OCCI working group is not responsible for OCCI reference implementations, but this is done by the community. There exists an implementation for Openstack, but according to the last commit to the git repository it does not seem active though.

6.1.1.2 TOSCA

TOSCA[44] (Topology and Orchestration Specification for Cloud Applications) is a standard that has been published by the Organization for the Advancement of Structured Information Standards (OASIS) in November, 2013. A YAML version has been released in March, 2014 as a draft[45] and v1.1 is, to the best of our knowledge, still under development. Although it introduces unique features, some concepts can be compared to AWS CloudFormation or Heat from OpenStack. Palma et al. introduce in [44] a way of describing service components and definitions of how they depend on each other, in a structured way using a service topology. Through orchestration processes, TOSCA provides the ability to describe how such resources are created or modified. The combination of service components and orchestration processes are grouped into service templates, which provide sufficient information to enable application deployment to different environments of various cloud providers and to support the management of the application throughout the whole software lifecycle, such as updating or elastic scaling.

6.1.2 Unifying Cloud APIs

Unified APIs hide proprietaries characteristics of an arbitrary cloud provider API exist for different program languages, which are discussed in the following.

6.1.2.1 DeltaCloud

DeltaCloud[2] can be used as a Ruby library, but ships with its own server, which is a top-level project of the Apache Software Foundation[3] and provides a unified REST interface to perform commands on the IaaS abstraction layer. It supports major cloud providers[4], such as Amazon

[1] `http://occi-wg.org/about/specification/`
[2] `http://deltacloud.apache.org/`
[3] `http://www.apache.org/`
[4] `http://deltacloud.apache.org/drivers.html#drivers`

AWS, OpenStack or IBM SmartCloud. The server can be seen as an intermediate between the developer or a program and a specific cloud provider. It abstracts away singularities of cloud computing APIs through cloud specific drivers and provides a unified interface, which can be addressed in three different ways. Firstly, Delatcloud provides a backend web interface, which can be accessed via a browser. As Amazon is one of the leader in cloud computing, their EC2 API[5] is supported as well. Furthermore, CIMI (Cloud Infrastructure Management Interface) from DMTF[6] is implemented, which provides the possibility to use OVF as well.

Other than Deltacloud, the following libraries have to be embedded into an application. Programs are written against the library interfaces, which offer a generic way of accessing the API of an arbitrary supported cloud provider. Among others, libCloud and jClouds are examples for such libraries:

6.1.2.2 libCloud

libCloud[7] can be used in Python programs. The library supports the most popular cloud providers[8] and implements four components: *Compute*, which is mainly used to manage virtual servers and to run deployment scripts. *Storage* to access cloud storage services, *Load Balancer* to create, delete and maintain virtual load balancers (Load Balancer as a Service) and *DNS* to manipulate DNS zones or add a DNS entry. Furthermore, libCloud offers a special debug mode, where all incoming and outgoing HTTP requests can be analysed. As libCloud is thread safe, it can be used in multi-threaded applications as well.

6.1.2.3 jClouds

jClouds[9], which is written in Java, can be seen as the small Java pendant to libCloud, because it unifies only compute and storage APIs[10]. For application debugging, stubs for both APIs are available, which enables developer to test their source code without targeting an existing cloud provider, which would imply costs. jClouds also offers a basic requirements matching functionality, to find the most appropriate instance type or machine image by providing hardware characteristics such as amount of CPU cores, working memory or hard disk size.

6.1.3 Proprietary Approaches to Describe Deployment Requirements

Most of the proprietary approaches to describe deployment requirements are declarative. This means the template contains instructions about what cloud resources are necessary, but it is not described how they should be provisioned. This is the responsibility of the provisioning interface, which consumes such template specification files.

---

[5] `http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html`

[6] `http://dmtf.org/standards/cloud`

[7] `http://libcloud.apache.org/`

[8] `http://libcloud.apache.org/supported_providers.html`

[9] `http://jclouds.incubator.apache.org/`

[10] `http://jclouds.incubator.apache.org/documentation/reference/supported-providers/`

6.1.3.1 Amazon AWS Approaches

With Amazon Web Services, Amazon builds a variety of additional cloud services around their original service of Elastic Cloud Computing (abbreviated EC2). Some of them try to minimise the complexity of the creation of new virtual appliances, whereas in the background EC2 instances are still used. This should not be an advertisement for AWS, but as Amazon is the leading company among IaaS platform providers [31], it is a good idea to analyze and evaluate some of their approaches.

CloudFormation. With CloudFormation, Amazon introduced a template language to describe multi-tier applications with AWS specific resources. This human-readable definition of cloud resources, software packages, hardware requirements and configuration in JSON notation can be extended with embedded console scripts, which can contain installation instructions, additional configurations for software or any arbitrary console command. Amazon provides ready-to-deploy templates for different applications[11]. Although the whole template language is related to AWS, Heat (see Section 6.1.3.2) for OpenStack offers an interface, which is capable of parsing CloudFormation templates and use them for deployments.

Amazon OpsWorks. Amazon OpsWorks takes a different approach and introduces a stack with different layers. Each layer is responsible for a specific purpose and can be customized with Opscode Chef recipes, which are grouped together into cookbooks[12]. In most of the cases, Chef cookbooks are maintained by the community via a Git repository. More technical, each cookbook consists out of various Ruby scripts (where the recipes are defined) and aim to work on different platforms (Windows included).

The drawback is that the cook books cannot be taken directly from OpsWorks, but have to be modified to work with AWS EC2 instances. One would have to adapt recipes or rely on the predefined layers from the AWS engineers, which makes it unattractive for a unified deployment solution.

6.1.3.2 Heat Openstack

Openstack follows with Heat[13] a similar way, compared to Amazon's CloudFormation approach. Heat is the orchestration module of Openstack and can parse templates in two different formats: YAML and JSON. Templates, which are in the YAML format, can use two different Heat template syntaxes, which have minor differences. The JSON format is compatible with the Amazon CloudFormation syntax and was the first format Heat supported. Although the HOT syntax is the official template format, there are still parallels between both formats and some concepts have been adapted. Whereas JSON uses brackets to define a tree structure, YAML relies on pure indentation.

[11] `http://aws.amazon.com/cloudformation/aws-cloudformation-templates/`
[12] `http://community.opscode.com/cookbooks`
[13] `https://wiki.openstack.org/wiki/Heat`

Heat can be seen as a proof that Amazon's approach with CloudFormation has the potential to get a de-facto standard, if more and more cloud providers implement the principle of describing required cloud resources as a template in an interpretable format.

Openstack's support for Heat is still in its early stages and in general this approach of describing a whole cloud resource deployment in form of a template is novel. The first commit[14] to the Git repository, announcing CloudFormation support in the context of Openstack, was in March, 2012.

6.1.3.3 Cloudify

Cloudify is a middleware, which can be used to manage cloud resources transparently. In the background, it uses jClouds to communicate with the supported cloud providers. Essentially, Cloudify consists out of a management machine, which takes care of and communicates with all virtual resources, which are part of the application. Initially, each node gets bootstrapped and runs a Cloudify agent, which can receive commands from the management machine. Deployment, scaling and monitoring of cloud resources are done via the management machine.

In the context of this master's thesis, Cloudify could be interesting as it provides the ability to define application and service recipes, which are used to describe the application in a cloud provider neutral way. The authors of [40] think that middleware systems like Cloudify, will have a strong influence on the further adaption of cloud computing, as they provide generic interfaces and infrastructure abstraction.

6.2 Similar approaches

In this section, a comprehensive overview of similar existing approaches and a clear differentiation to our approach is given. In some cases potential synergy effects are discussed.

6.2.1 Managing the Configuration Complexity of Distributed Applications

The authors from the IBM research department introduce in [23] an approach to help to manage the configuration complexity of distributed applications, which are deployed to data centres. Their approach is based on the principle of separating concerns: Application developers capture the logical structure of an application, as this is what developers are responsible for and know the best. Domain experts on the other side, define model transformation rules that incorporates their domain expertise. Application deployers provide common deployment patterns and cloud providers can provide a model that describes all available resources within their cloud. In the article they present a tool, which takes these four inputs and creates a set of possible deployment topologies, which satisfies the defined requirements.

The authors emphasise that one of their main goals is to share deployment knowledge in form of models. Some of existing methods of provision automatisation are based on scripts, which can be error-prone and not effective. Furthermore, they explain why MDE can have such

[14] `https://github.com/openstack/heat/commit/38de4d2564b75316ef5a61eb0b1a87b22a19731c`

a positive effect in relation to application deployment, as the model can be used and reused for various occasions, such as visualization, provisioning and configuration.

As models can significantly improve efficiency, Eilam et al. in [23] are convinced that MDE will be the base of the next generation of configuration management tools and most technologies used in this domain are already moving towards model-based solutions. With the principle of separating concerns, different types of models are introduced, which rely on each other: (i) logical application structure (LAS), contains all functional requirements an application has and is created by developers, (ii) logical deployment model (LDM), is based on a LAS and enhances it with desired deployment patterns. To speed up creation of LDM, logical topology models (LTM) are used, which can be compared to blueprints, which have been discussed in Chapter 3. LTMs are predefined topology models that consists out of best practise deployment scenarios. (iii) deployment topology generator, takes a LAS or a LDM and tries to create a set of physical deployment topologies. Such a topology consists out of a complete software stack, configuration constraints and network requirements. In other words, it is a description of a ready-to-deploy system. The conversion from a LAS into a physical deployment topology is achieved by applying iteratively model transformation rules, which have been created by domain experts.

Eilam et al. distinguish in [23] between deployment and provisioning. The deployment is described as the phase that starts with a LAS and ends with a physical deployment topology. Provisioning is the step, in which the physical deployment topology is parsed and gets converted into a deployment plan that is executed by the provisioning engine. A cloud provider may have specific rules or constraints, which have to be considered during the creation of the deployment plan. This information can be grouped together into a data centre model (DCM), which is taken into account by the planner.

Some of the main concepts are: (i) every module in a diagram has a container. For example a Java container has to be hosted by an application server, but the application sever at the same time is contained by an operating system, which is contained by a machine. (ii) Model transformation rules define how to manipulate the input model in terms of copying, adding, deleting or altering elements based on preconditions that those elements must fulfil. (iii) Within each iteration of applying transformation rules to the model, a solution tree is created, which can contain leafs that can not be further transformed, but at the same time do not represent a valid deployment topology. Those leafs are called dead end and the tree has to be examined along another branch. The authors call this approach transformation-based search, which can be enhanced by a proper heuristic in order to apply transformation rules in a specific order and not in a random order. A leaf of the solution tree has to reflect a valid deployment topology and has to be in accord with the constraints of a cloud provider specified in its DCM.

The prototype implementation consists out of models, which manly describe elements that can be inserted into the diagram during the execution of transformation rules. Interestingly the do not use ATL for transformation rules, but all of them are defined in a imperative way in form of Java code.

They list some of the advantages of their approach, such as: (i) reduction of complexity of application deployments and reusability of models, (ii) separation of concerns, as everyone contributes those parts to the deployment, where she is an expert in. For example, domain experts can define transformation rules, and application deployer can concentrate on their field of

responsibility), (iii) model transformations replace the requirement of (re)writing and adapting deployment scripts and operate on a higher abstraction level.

This approach has some similarities with the one, which has been described in this master's thesis. LAS and LDM in their approach can be seen as the equivalent to PIM and PSM in our approach. We encapsulate knowledge about specific cloud providers in UML profiles, whereas here DCMs are used. To speed up the modelling process LTMs are used that contain a predefined deployment structure. This is similar to our blueprints and software stacks that contain best practise deployment scenarios.

The principle of a strict separation of concerns only has been considered to a certain extend. We distinguish between model library maintainer and modeler. A more detailed definition could be one potential extension (see 7.3).

One important difference is that the workflow presented in this approach ends with a deployment topology, but the actual provision of the defined cloud resources have to be done manually. Our approach goes one step further and tackles the problem of provisioning as well.

6.2.2 Uni4Cloud

The authors of [47] introduce an approach called Uni4Cloud, which tries to facilitate the modelling, deployment and management of applications in a multi-cloud environment on the IaaS abstraction layer. They identified three main stakeholders: cloud providers, a service providers that rent the infrastructure of cloud providers to deploy their applications and service users that use the service hosted in the cloud. Their approach consists out of three modules:

Service modeler. The service modeler is a graphical user interface, which facilitates the creation of a deployment model. In the background, the information is stored as an OVF compliant appliance (for further details of OVF see Section 6.1.1.1). Furthermore, the service modeler contains predefined OVF templates of virtual machines, load-balancers and web servers that can accelerate the modelling process. The resulting OVF file is never manipulated directly, but every time through the user interface, where also application-specific and cloud-specific properties can be defined. As OVF does not offer this functionality out of the box, Uni4Cloud uses self-defined extensions that can be used in combination with the OVF format.

Service manager. This module is responsible for parsing the OVF description file and for deploying it to the selected cloud provider. The service manager does not contact the cloud provider directly, but communicates through the cloud adapter (see next paragraph), targeting a unified cloud provider agnostic interface called OCCI. After deployment, any changes concerning resource management or dynamic behaviour, such as scaling, are done by this module as well. The authors note that this functionality is still in its early stages and is not explained further.

Cloud adapter. The cloud adapter essentially is an implementation of the OCCI interface specification. Every supported cloud provider has its proper plugin which translates unified

OCCI commands into proprietary calls, which are specific to one cloud provider. With this strategy, it is not important, to which cloud provider the service manager is going to deploy to, as the communication is based on OCCI and any difference between cloud providers are transparent.

Uni4Cloud uses open standards such as OVF (described in 6.1.1.1), but OVF does not seem to be practicable for describing cloud resources, due to the lack of support of this format. The proposed service modeler seems to be in an early stage and there is space for improvement. To the best of our knowledge, we could not find subsequent publications based on Uni4Cloud. As already mentioned, the integration of elastic scaling, has been considered but is still not implemented.

The reason why OVF is used as a format for the service modeler is unclear, as it gets parsed by the service manager in the second step and OVF definitions are translated into OCCI specific commands such as *create instance* or *create network*.

Instead of introducing a new modelling language, our approach relies on the well established UML and uses native modelling elements, which are enhances by profile applications. Furthermore, we also address the ability to define scaling rules.

6.2.3 CloudMIG

The focus of cloudMIG [25], is set on the controlled migration of an application and its sub-parts into a adapted version of it, in order to leverage cloud specific advantages, such as scalability and resource efficiency.

By means of an experiment, the authors pointed out that efficient application migration is more than just creating virtual instances and installing the application without any further modification. To proof this assumption, they installed the open source ERP system Apache OFBiz[15] on a couple of virtual instances and measured response time and CPU utilisation among a certain time frame. Their results show that some instance could no handle the increase of requests efficiently, as the request response time went above an acceptable threshold. On larger instances, CPU resources were over-provisioned, which resulted into more expenses. Although the application was runnable on virtual instances, thus cloud compatible, it could not exploit the advantages what a cloud offers. Furthermore, the authors are convinced that the advantage of cloud computing in comparison to a self-owned computing infrastructure, of being more elastic in terms of allocated virtual instances, is not exploitable by simply installing the unmodified application in the cloud, but more sophisticated application re-engineering has to be done. In other words the problem of under-provisioning and over-provisioning can also occur in the cloud.

As already mentioned, CloudMIG aims for an easy re-engineering process to make existing applications ready for migration to the cloud and takes out some complexity, when adapting it to target a specific cloud provider. As there will be always heterogeneity among cloud providers, CloudMIG strives for a generic application migration approach.

The authors introduce a Cloud Suitability and Alignment (CSA) hierarchy that can be used to classify existing software systems for their suitability of being run in one specific cloud. There exists five statuses: Incompatible, Compatible, Ready, Aligned, Optimised. They can be

[15] `http://ofbiz.apache.org/`

ordered in form of a pyramid, which means if an application in relation to one cloud provider has as a status *aligned*, it is *ready* and *compatible* as well. The CSA hierarchy only takes technical hurdles into account and no organisational restrictions or security policies.

Apart from CSA, Cloud Environment Constraints (CEC) are used to describe limitations of a specific cloud provider. On a PaaS abstraction level, there may be some restrictions in terms of a limited available feature set of the programming language execution environment. There are three different types of CEC violations: Warning, Critical and Breaking. Those types are in direct connection to the CSA status of an application. If the current status of the application causes at least one CEC violation with the status *breaking*, the CSA would be *incompatible*.

The CEC definitions are included by a Cloud Environment Model (CEM), which has to be created for every cloud provider CloudMIG should support. Furthermore, it includes transformation rules and cloud provider specific properties.

According to the authors, all simplistic migration approaches have at least one of the following shortcomings: (i) applicability, as only a few cloud providers are supported (ii) level of automation: most of the re-engineering work is done manually as it is quite complex and cloud environment constraints are not checked at design time (iii) resource efficiency, as software may not be designed for the cloud environment's elasticity and (iv) scalability, as there does not exist automated support for evaluating the saleability of the target architecture. After having discussed the terminology of CloudMIG, an overview of the migration process is given. It consists out of six phases and tries to address the previously mentioned shortcomings:

Extraction. As CloudMIG is a model based approach, the current architecture has to be represented as a model. Sometimes the internal structure is not known completely or incomplete. CloudMIG introduces a software architecture reconstruction methodology that analyses the program code and generates a model, which conforms to OMG's Knowledge Discovery Meta-Model [42]. In order to consider the last two shortcomings *Resource efficiency* and *Scalability*, fundamental knowledge about the application's statistical properties, such as invocation rates or average request size in bytes have to be collected to make proper decisions about how the target architecture should look like. Data can be retrieved by processing log files or the usage of monitoring tools and are stored in form of metrics in a model that conforms to the Structured Metrics Meta-Model (SMM) [42], which also has been published by OMG.

Selection. After a model of the current application has been generated and utilization metrics have been derived, a cloud provider and its associated CEM are selected. It contains a set of CECs and transformation rules that are used in the next phase.

Generation. The generation phase has three outputs: Target architecture, mapping model and constraint violation model and furthermore is subdivided into three sub processes. Firstly, the model transformation phase uses the model transformation rules defined in the CEM and applies them to the elements of the current architecture. In other words, each element is assigned to one possible cloud resource, whereas it is about an initial assignment, which can be changed by one of the subsequent steps. The next step is manually done by the developer and consists of the configuration of rules and assertions. Some rules may be changed, such as altering numerical

values or adjusting the execution order of them. The last step improves the mapping created in the first one by considering resource-efficiency and rules that were adjusted by the developer. Not all resource adjustments in favour off efficiency will result into an overall increase of efficiency, as there may be some side-effects. For instance, splitting the application into components and deploying them on different virtual machines to get optimise CPU usage, will increase the network traffic due to the necessary communication between those components. In order to get the best solution a heuristic rule-based search algorithm is used to consider all possible solutions and select the best one. There may be some parts of the applications that have to be adapted manually, as they violate one of the CEC, which is done in the last phase *transformation*.

Adaption. If the automated generation process has not considered some of the case-specific requirements, the target architecture can be adapted manually. This phase can consists out of multiple iterations.

Evaluation. Before the application gets transformed into the target architecture, it has to be evaluated in terms of metrics. Apart from static analyses such as LCOM, the model can be simulated with CloudSim[16], to get a better insight of possible consequences and behaviour of the application. The results may cause the developer to adjust some parts of the deployment, which means going back to the adaption phase.

Transformation. The last phase is about realising the plan. There does not exist any support from CloudMIG and is completely left to the developer.

CloudMIG extensively addresses the problem of adapting an application in a way, so that it can be run on cloud resources and can exploit the advantages cloud computing offers. Our approach mainly focuses on the underlying infrastructure. It is conceivable to combine both approaches in order to benefit from each other.

6.2.4 CloudMF

CloudMF [24] introduces two levels of abstraction: Cloud Provider-Independent Model (CPIM), which is cloud-agnostic and can be used as a template and Cloud Provider-Specific Model (CPSM), which contains cloud provider specific concepts and is used as an input for the deployment and provisioning engine. The described workflow can be separated into two parts: Modelling environment and Models@run-time environment:

Modelling environment. The first step is done through an editor, which enables the user to create a CPIM of the application in the domain specific modelling language (DSML). They distinguish between type and instance definitions. Type definitions contain information of how an artifact is retrieved, deployed and started. A instance definition, which is derived from a type, are concrete representations and are used in the next step as an input for the refinement

[16] `http://www.cloudbus.org/cloudsim/`

engine. The refinement engine converts a CPIM into a CPSM by adding cloud specific features, which is done through the provisioning and deployment engine that in turn communicates with the respective cloud provider. Once a CPSM is created, the provisioning can be done directly in an imperative way by contacting directly the provisioning and deployment engine, or in an declarative way through the models@run-time environment.

Models@run-time. The aim of the models@run-time environment is to reflect a change in the running system in the model and the other way around, which means if the CPSM is changed the running system should be updated as well. In order to do that some prerequisites are necessary, such as sensors to detect changes in the running system or actuators, which are capable to the propagate model changes to the running system.

CloudMF introduces two different types of models: CPIM and CPSM, which are related to PIMs and PSMs of our process, but instead of using UML as the modelling language, a proprietary DSL is used.

6.2.5 Aeolus

Aeolus[16, 39] address the problem of component deployment in the cloud. Components are used to describe resources that can provided or require functionalities through ports. Each component is expressed as a state machine, whereas a transition from one state to another can be prevented if required functionalities are not satisfied. The focus lies on software packages that can have dependencies on other software components. Moreover, they distinguish between strong and weak requirements. In [16] they introduce the Aeolus flat model and describe eight formal definitions, by means of the set theory notation. The complexity an Aeolus model tries to illustrate in a declarative way is a "universe of possible components" [39] and a target state in which individual components have to be in an active state in order to execute deployment. The complexity is resolved by breaking it down into a "sequence of low-level deployment actions" [39], which consist out of the creation and deletion of a resources, the binding and unbinding of a port of a component and the state change of a component. Understandably there can exist more than one sequence of low-level deployment actions to achieve a given configuration of components. The authors of [39] elaborate on a novel planning technique to create a deployment plan that is based on Aeolus models. It is subdivided into three parts: Reachability analysis, abstract planning and plan generation.

Reachability analysis. The first step includes the calculation of all possible states each component can obtain. As soon as the target state is reached, the creation of an abstract plan can be initiated. For simplification during the reachability analysis, the low-level deployment actions *bind* and *delete* are omitted. To find a way to reach the target state, a certain amount of state sets are generated. The creation starts with all components being in their initial states. With each iteration, a new set of states is created, by changing the state of a component having in mind any functional dependencies. As soon as another iteration does not generate a new set of states, there are two possibilities: the target state is among the the generated sets or there does not exist

a solution for the goal. All sets together form the so-called reachability graph, which is used in the next step.

Abstract planning. If the previous step identifies one or more possible solutions, an abstract plan is computed. The abstract plan is retrieved by traversing the reachability graph in reverse order by starting with the final state. Based on a heuristic it is determined, which node should be collected and should be part of the abstract plan. For instance, between two nodes, the node that is able to satisfy the maximum number (not already satisfied) requirements should be preferred.

Plan generation. The last step consists out of deriving a concrete plan based on the abstract plan from the previous step. The strategy is to visit nodes from the abstract plan and execute the action in order to get there, until the target state is obtained. Sometimes it is necessary to adapt the plan, if a component duplication is required - this would be the case if the same component in different states should be deployed simultaneously.

Aeolus uses its own defined syntax to define the desired target system in the cloud, which is parsed by *Zyphyrus* and forwarded to *Armonic*, a deployment tool based on state machines that can translate deployment definitions into API invocations [18].

This master's thesis introduces a UML centric approach and application modelers do not have to learn additional meta languages, as it would be the case for Aeolus.

CHAPTER 7

Conclusion

7.1 Critical Reflection

The aim of this section is to summarise the master's thesis results in a critical way and identify current limitations and possible improvements.

7.1.1 All-Embracing Solution

Our proposed process should simplify the deployment and provisioning of an application to the cloud and furthermore should be intuitive. At some stages the prototypical implementation lacks of flexibility and is bound to some constraints, as it is hard to develop an all-embracing solution. The implementation of a solution that addresses all different deployment scenarios and application specific singularities can be complex or not feasible. Only by applying the process to real world examples, disadvantages and weaknesses can be identified that furthermore could be used as an input for modifications, adaptions or improvements of the process.

This master's thesis follows a model-based approach, which proofed to be feasible and practicable. The EMF provides well maintained tools to manipulate models and align them with the deployment process and use them to support the provisioning process. The presented CMM is vendor independent and provides a unified way to describe cloud deployments on the IaaS abstraction layer.

7.1.2 Automation

The goal of the process is a (semi-)automatical deployment and provisioning of an application to the cloud. Especially when it comes to application specific environments, model to model and model to text transformations are quite static and outcomes have to be manually altered (as already discussed in Section 5). This includes manual insertion of additional installation and configuration scripts into the template to install specific application dependencies and configure them. This strongly depends on the application which is run on each server and can not be

defined in a generic way as too many variables would influence the way an installation script arranged.

7.2 Current Limitations and Possible Improvements

In this section, the presented process is analysed under the aspect of existing limitations. Some of them can be circumvented by implementing extensions, which are discussed in Section 7.3.

Modelling of Networks. When creating virtual appliances in the cloud, not all of them should be reachable from the internet. This is achieved through security groups, which can be seen as firewall rules that determine which ports and protocols are going to be forwarded to a virtual machine. At the same time, parts of the network should be private as well, as the communication between nodes should not take place through the internet. This can be achieved by placing virtual machines into private networks within the cloud. The ability of creating private subnetworks is still under development. Our plan is to distinguish between different type of links (communication paths) between nodes and to make it possible to assign private IP addresses to cloud resource.

Provisioning of Application. The evaluation concluded that more complex applications can not be run on a virtual instance within the cloud without being further configured and "cloudified". The approach that is presented in this master's thesis focuses on the deployment and provisioning of the underlying infrastructure that the application needs to be run in the cloud. Formats, such as CloudFormation, offer the possibility of embedding a simple application-provisioning workflow within the template. The functionality is limited though and consists of console scripts that get executed during creating a new virtual machine. The application archive has to be publicly available in order to be copied automatically into the web directory. As an application in most of the cases relies on a data persistence layer, the IP address of a database has to be known in advance. Under certain circumstances, the creation of the database may consume more time than launching a new virtual instance, which means that the IP address is still not known. For this reason, we leave the automated application provisioning to software solutions such as *Chef Opscode* or *Docker*, which encapsulates an application and its dependencies in a Docker container. Both are discussed in Section 7.3.

Modelling of Software Stacks. In our approach we tried to model common software stacks and included them into our model library. The problem is that there is no general software stack, which is valid for all applications. For instance, a JBoss server may require special configurations and parameters that are hard to express in form of a model. In other words, there are so many different cases of how a JBoss application server can be configured that it would not make sense to reflect the configuration in form of models. There exist solutions, which target exactly the problem of supervised simultaneous multi-node configurations, such as Chef Opscode, which is discussed in the next section.

7.3 Potential Extensions

In this section potential extensions that could add additional functionality to our approach are discussed.

7.3.1 Scaling rules for PaaS

Apart from analysing scaling configurations from Amazon and OpenStack, we also tried to consider scaling rules on the PaaS abstraction layer. It turned out that scaling rules on a PaaS level are application centric. This means, they are targeting the application context such as pending latency or concurrent requests and can not be combined with scaling rules targeting IaaS. For that reason we introduced a new section in out Xtext grammar, called *application*. As shown in Figure 7.1, a rule can define scaling behaviour for a whole instance, or only for a specific application. Examples for scaling criteria for applications are *PendingLatency* or *ConcurrentRequests*.

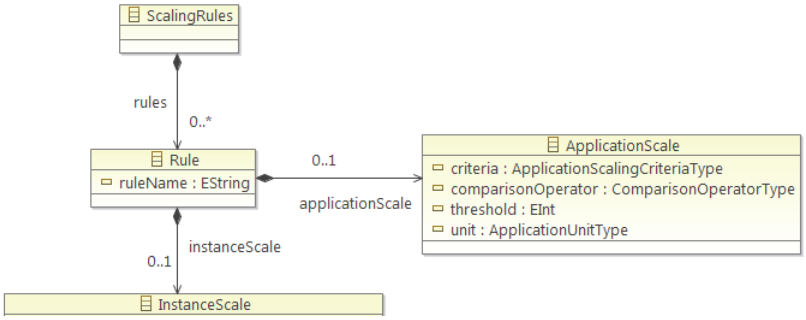

Figure 7.1: Extending the Scaling Metamodel to Define Application Scaling Behaviour

7.3.2 In-place Transformation for Requirements Matching

Although refinement operations are done manually, we experimentally used in-place transformations to do basic requirements matching. For instance, the rules can determine if hardware constraints of a node are compatible with one of the instance types and applied the respective stereotype to the node element. In Listing 7.1, an excerpt of the in-place transformation is given that shows how the most economic instance type is determined.

7.3.3 TOSCA Integration

The already introduced open standard TOSCA [44], which has been published in November, 2013, defines a uniformed way of describing cloud resources and their orchestration in a textual format. Amazon AWS with CloudFormation was the trend setter in the field of cloud resources orchestration, which was adapted by the orchestration service Heat of the open source cloud OpenStack, as lots of the concepts can be found in it. Other than CloudFormation, which uses JSON for the template syntax, Heat and also TOSCA use YML as a format to describe templates.

Listing 7.1: In-Place Transformation: Requirements Matching

```
1  thisModule.economicPerformanceRequierementsMatching(
2          targetStereotype.getStereotypeApplication(appliedStereotype),
3          possibleStereotype.getStereotypeApplication(appliedStereotype)
4  )
5
6  helper def : economicPerformanceRequierementsMatching(current : OclAny, new : OclAny) : Boolean =
7          current.virtualCores > new.virtualCores
8          and current.memory > new.memory
9          and current.localDisk > new.localDisk
10 ;
```

As this master's thesis already supports CloudFormation and Heat, it is easy to add support for new formats such as TOSCA.

7.3.4 Chef Opscode integration

As already mentioned, one of the limitations of our approach is the ability to model the necessary software stack in an all-embracing way. There exist software solutions, which try to solve exactly this problem. One of them is Chef from Opscode[1]. Chef is capable of configuring virtual appliances once the have been provisioned. This means, as soon as a virtual instance is created, Chef comes into play.

It is possible to describe the necessary infrastructure as code snippets, which can be checked in into a version control system and later on can be distributed to the respective nodes that have to be configured. Chef focuses on the software layers that are above the operating system. The configuration itself can be seen executable documentation of the infrastructure, which is needed to execute the applications [14].

Chef comes with its own domain specific language, based on Ruby, which provides a way of describing the desired state of a resource (so-called policies). Comparable to orchestration templates such as CloudFormation or Heat, it is described what kind of resources are required, but not how this should be achieved. Depending on the operating system the way of how the desired state of a node is achieved can be different. Chef chooses, depending on the underlying software, a certain provider, that abstracts away how new software gets installed on the node or how the system gets configured. As the description is platform independent, it can be used for private server systems as well as for cloud infrastructure. The basic setup of a Chef environment consists out of a administrator workstation, which is capable of creating new configurations for parts of the application infrastructure, a Chef server, which stores all configurations and keeps a record of all manageable nodes, and nodes (virtual instances) that have to be configured. To assure that Chef Server knows about available nodes it has to manage, they have to get bootstrapped once in the beginning. Chef defines some concepts, which are explained shortly [14]:

Resources. A resource is a small building block of a configuration template. For instance, this could be a packet that should be installed, a file that has to be modified or a service, which

[1] `http://www.getchef.com/`

should be started during startup. They can depend on each other and inter-communicate through notifications, such as a resource could trigger another resource to be called.

Recipes and Cookbooks. A recipe contains an arbitrary amount of resources and segments of code that define the policies, whereas a Cookbook contains various recipes and other file templates that are typically needed for a specific software or functionality. To retain flexibility and cover most of the operating systems, recipes which are describing the same software are grouped together into one cookbook, which mostly is community maintained through a version control system such as Git. For instance, the Nginx cookbook can be found under `https://github.com/opscode-cookbooks/nginx` and can be used for various Linux distributions. Each recipe is written in Ruby and can depend on other recipes. The goal is to achieve an unattended installation process, which automatically does most of the setup workload.

Run list. During the configuration of a machine, a chef client gets executed that retrieves a run list, which contains a definition of cookbooks to be executed. For instance, a run list could contain a JBoss and an Apache cookbook. In other words the run list is a collection of policies, which have to be configured on that specific node. The chef client follows a check and repair strategy, which means if the state of a resource meets its definition, no action is taken, whereas any deviation from the resource definition is tried to be accommodated.

Roles. In an orchestration of cloud resources the individual virtual instance is not addressed any more, but roles are assigned to a group of resources such as web server, database server. This means a node can have one or more roles that it fulfils in the cloud appliance constellation. Because of that with Chef it is possible to define roles and related cookbooks.

After having explained the basic functionality of Chef Opscode, the question is how it can be integrated into our process. As already said our process lacks simplicity of defining a software stack. At the same time the information of how cloud resources are arranges and how they depend on each other is present in the UML models that have been created during the deployment process. This data could be used to automatically generate run lists and define roles, to support the integration with Chef. There already exist lots of cookbooks maintained by the Chef Opscode community, which means an automatic search for cookbooks that meet specific requirements should be possible as well.

On the other hand it would be also possible to extend our self-implemented Provisioning Client. As it is based on jClouds, which is compatible with Chef Opscode, it would be possible to bootstrap virtual nodes with the Chef client programmatically and use cookbooks on them.

7.3.5 Docker

Docker[2] was founded by Solomon Hykes and offers a functionality of executing applications in isolated containers, which communicate with the underlying hardware through the Docker

[2] `https://www.docker.com`

Engine.
Figure 7.2 depicts the difference between a conventional virtual machine on the left hand side, which includes beside application code and other dependencies its own operating system and a Docker container, which is a lightweight application container that can be deployed to any arbitrary physical hardware, as it gets executed by means of the Docker Engine in an isolated userspace process.

In other words this means that Docker provides all advantages a virtual machine has and additionally is highly portable and efficient, due to its small container size. The container is in a way standardised which means application developers care about the container contents, such as the application itself and its dependencies, whereas the system administrators maintain the appliances the Docket Engine runs on. With the approach that has been presented in this master's thesis, both tasks can be accomplished by the developer.

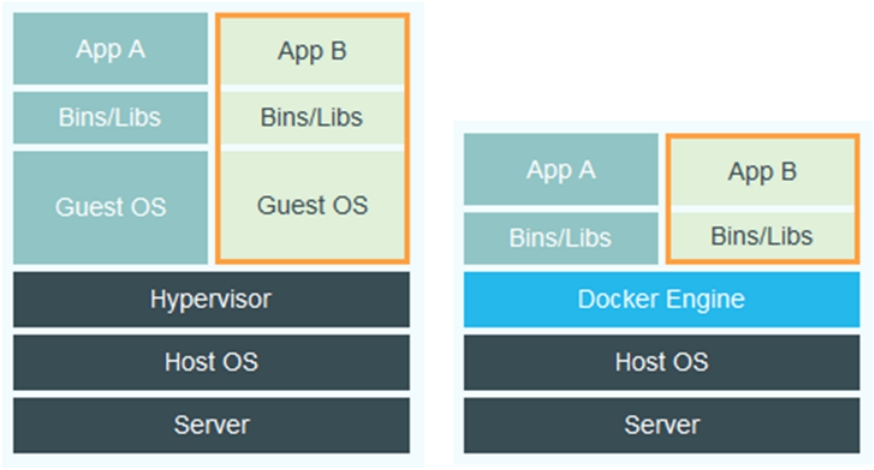

Figure 7.2: Difference Between Conventional Virtual Machines and Docker, Source: `https://www.docker.com/whatisdocker/`

Once a Docker container for an application has been created, it can be scaled easily as launching additional containers is a matter of seconds. In combination with our approach, scaling would happen on two layers. Additional Docker containers on one side, additional virtual instances on the other side. As already mentioned, Docker container are lightweight, so it is possible to run multiple containers on the same virtual instance, which increases efficiency and may lower the overall running costs.

A container gets launched from a Docker image, which can be seen as a read-only template. Most of the images are maintained by the Docker community and lots of pre-configured software stacks and execution environments such as JBoss or Glassfish are available at Docker Hub. Docker also follows a layer approach (see Section 2.4 for more details), but for example if the application gets updated by a newer version, it is not necessary to recreate the whole image from scratch, but an existing layer gets updated or an additional layer is added. In order to propagate image changes, it is not necessary to send the whole image via the network, but just the update.

This can be compared to a version control system that also only transmits changes.

To evaluate if Docker would be a suitable extension for our approach, we installed Docker on a virtual machine and tried to run a JBoss Docker container from a pre-configured image available at Docker Hub. The necessary command to launch a Docker container and start the JBoss web server is illustrated in code listing 7.2.

After the container has been build, it is a question of seconds to launch the application server. Compared to a virtual instance, which has to start all services after having booted, a huge benefit. As Docker uses the virtualisation interface of the kernel of the host, the overhead of running a container can be reduced to a minimum [22].

Listing 7.2: Command to Launch a JBoss Docker Container

```
1 docker run -d -p 8080:8080 -p 9990:9990 tutum/jboss
```

With MasteroNg[3] the orchestration of Docker containers can be described in a textual format. It can be seen as a Heat template for Docker purposes. Beside the definition of working memory and amount of CPU cores, the inter-dependencies among containers can be defined. A text-to-model transformation into a MasteroNg compliant format would be conceivable.

7.3.6 CloudSim

CloudSim [13], is a "Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services". The aim is to simulate virtual appliances and services in form of models before the provision process of cloud resources takes place.

CloudSim offers a Java library that contains cloud concepts and models such as data centre or virtual instance. A concept has properties, which have to be configured before a simulation can be initiated. The result of a simulation can include cost calculations, network flow and network utilisation and electricity consumption. The simulation scenario is described in form of a Java program.

The output of the solution presented in this master's thesis, can be combined with CloudSim in a way of created model-to-text transformations, which generate Java code that represent the modelled cloud resources. Before the provisioning of cloud resources is initiated, CloudSim could help to identify suboptimal configurations or network performance bottlenecks, which could be fixed proactively.

7.4 Lessons Learned

In the last section, a couple of lessons learned are discussed, which were as well an important outcome of this master's thesis.

[3] `https://github.com/signalfuse/maestro-ng`, accessed 11-09-2014

7.4.1 Different Strategies of Virtual Image Configuration

In the evaluation Section 5, we evaluated different image configuration strategies, which are explained in 2.4 in more details. Especially in combination with elastic scaling, we could identify some major disadvantages for some of the strategies. For example, if a light weight machine image is used that only contains the operating system, the whole execution stack and software packages have to be installed when the machine has been booted for the first time. This can include the download and installation of an application server such as JBoss, which can consume more time than the virtual machine would have been needed. Another negative aspect would be, if the virtual machine after a couple of minutes is still not available (due to time-consuming installation processes) and the creation of another virtual instance is initiated.
Which strategy should be used, strongly depends on the environment and the nature of the application being run on the virtual machine. If it is not a problem if the creation of an additional virtual machine takes a couple of minutes, the Raw-image strategy could be good enough.

7.4.2 Getting Familiar with Offerings and Available Technologies

It is really important to get familiar with offered technologies and services that are offered by the cloud providers beforehand. In the case of Amazon AWS, creating different resources by hand and experimenting with different APIs was important to gain a deep insight into the provided functionalities. As we configured Openstack as a private cloud, we could get even more detailed knowledge about what is going on behind the scenes. So, rather than making unrealistic assumptions, it was better to go the way backwards: Starting with orchestration solutions provided by the cloud providers and using them manually, helps to get an idea what is possible and what not. With this knowledge, model to text transformations could be implemented and further on, model to model transformations. In other words the process that is presented in this master's thesis widely has been implemented backwards.

APPENDIX A

Cloud Metamodel

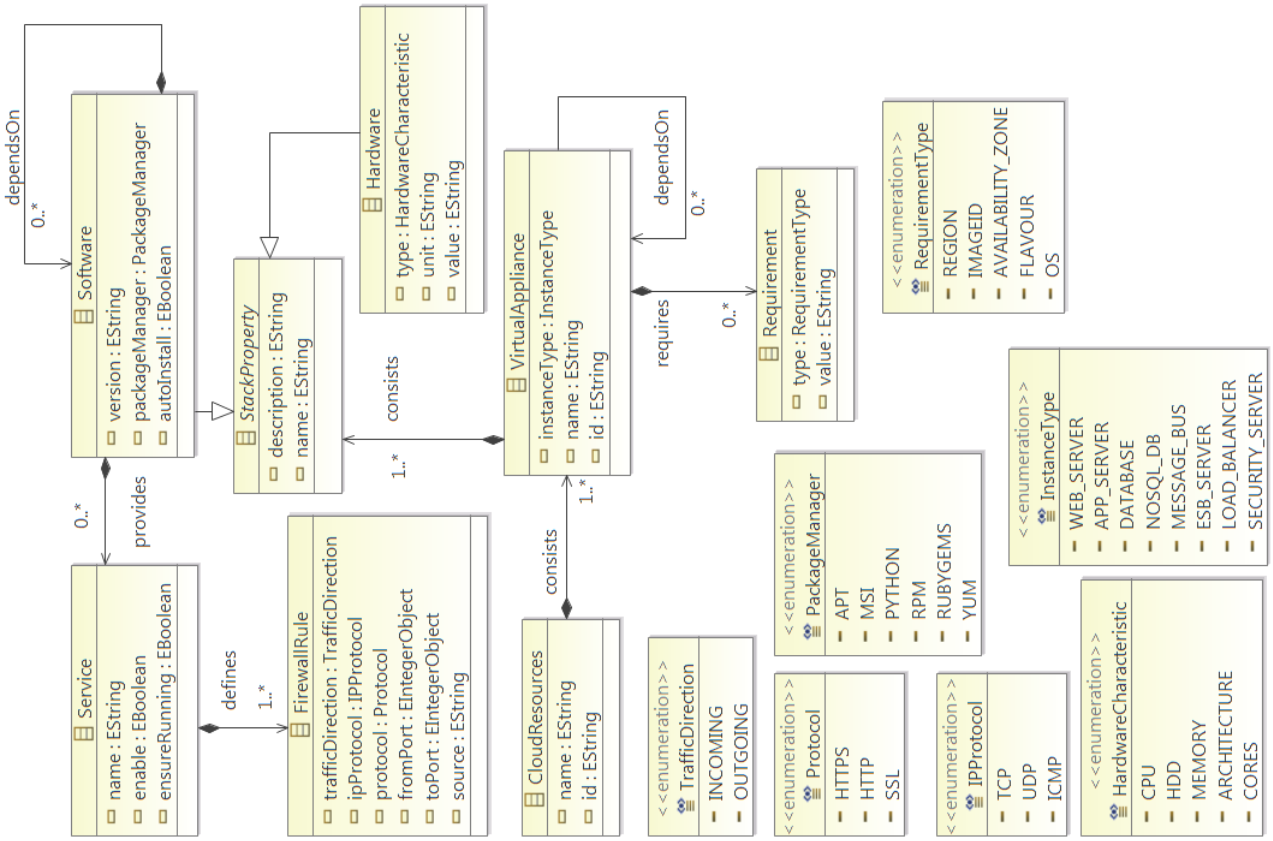

Figure A.1: Cloud Metamodel designed in Ecore

APPENDIX B

Scaling Metamodel

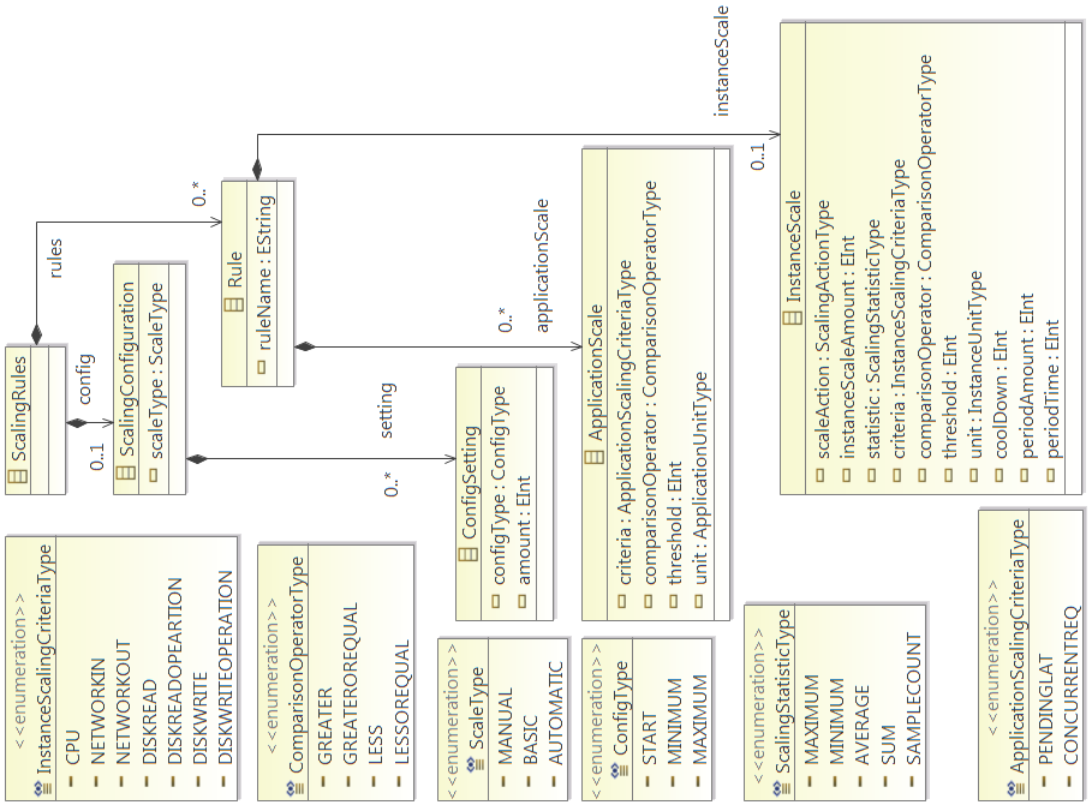

Figure B.1: Scaling Metamodel designed in Ecore

List of Figures

1.1 Context of master's thesis within ARTIST [10] 2
1.2 Deployment Requirements of Sample Application 4

2.1 Cloud Computing Abstraction Layers, Source: Author with ideas from [28] 11
2.2 Virtual Image Configuration Options. Source: [2] 19

3.1 Deployment and Provisioning Process 24
3.2 Relation between a Cloud Resource and Virtual Appliances 25
3.3 Requirements of a Virtual Instance 26
3.4 Definition of Stack Properties 27
3.5 Software and Firewall Rules 27
3.6 Composition of Scaling Rule 28
3.7 Blueprint for a Common Web Application Deployment 29
3.8 Blueprint for a Tomcat Software Stack 30
3.9 Transition to a Model that uses Model Library Stereotypes and Elements. 31
3.10 First Model Refinement Iteration. 32
3.11 Second Model Refinement Iteration. 32
3.12 Third Model Refinement Iteration. 33
3.13 UML Profile Enrichment 35

4.1 Deployment and Provisioning Process 38
4.2 OCL constraint used for Model Validation 39
4.3 Scaling Rule Definition 40
4.4 Model to Model Transformation 41
4.5 Model to Text Transformation with CloudFormation as the Target Format 42
4.6 Deployed RDS Database with Deployment Information 44
4.7 UML Profile Enrichment 44
4.8 Extracted Availability Zones and Instance Type from Amazon AWS 46

5.1 Network Topology for the Private OpenStack Cloud 50
5.2 General Deployment Requirements for Sample Application #1 51
5.3 AWS Specific Deployment Requirements for Sample Application #1 51
5.4 Operating Environment Requirements for Sample Application #1 52
5.5 Provisioned Cloud Resources for Sample Application #1 53

5.6 General Deployment Requirements for Sample Application #2 54
5.7 Scaling Rules For the Application Server 55
5.8 Deployment Requirements of Required Cloud Resources for the Ticket Monster Application 59

7.1 Extending the Scaling Metamodel to Define Application Scaling Behaviour 77
7.2 Difference Between Conventional Virtual Machines and Docker 80

A.1 Cloud Metamodel designed in Ecore 83

B.1 Scaling Metamodel designed in Ecore 85

Listings

2.1 Sample Template of a HOT Template in YAML Syntax 18
4.1 Excerpt of the Scaling DSL Grammar 40
4.2 Sample Output of Textual Model Used by Own Deployer 43
4.3 Sample Output of the Information Extraction Script: Regions and Prices 45
4.4 Sample Output of the Information Extraction Script: Flavours 46
5.1 Configuration of JBoss Server within CloudFormation 56
5.2 Installation of the Package cfn-init 57
5.3 Adding a New Image to the Glance Image Repository 57
5.4 Configuration Scripts for *ComputeNode* 61
7.1 In-Place Transformation: Requirements Matching 78
7.2 Command to Launch a JBoss Docker Container 81

Bibliography

[1] D. J. Abadi. “Data Management in the Cloud: Limitations and Opportunities”. In: *IEEE Data Eng. Bull.* 32.1 (2009), pp. 3–12.

[2] Amazon Web Services. *AWS Webinar Using AWS OpsWorks and Amazon VPC*. `http://www.slideshare.net/AmazonWebServices/ops-works-vpc-130912`. [Online; accessed 24-November-2014]. 2013.

[3] E. Anderson et al. *Forecast Overview: Public Cloud Services, Worldwide, 2011-2016, 4Q12 Update*. Gartner, 2013.

[4] M. Armbrust et al. “A View of Cloud Computing”. In: *Commun. ACM* 53.4 (2010), pp. 50–58.

[5] M. Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, 2009.

[6] M. L. Badger, T. Grance, R. Patt-Corner, and J. M. Voas. *Cloud Computing Synopsis and Recommendations*. Tech. rep. National Institute of Standards and Technology, 2012.

[7] L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. “A Partial Evaluator, and its Use as a Programming Tool”. In: *Artificial Intelligence* 7.4 (1976), pp. 319–357.

[8] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel. “UML-based Cloud Application Modeling with Libraries, Profiles, and Templates”. In: *CloudMDE@MoDELS 2014*. 2014, pp. 56–65.

[9] A. Bergmayr, M. Wimmer, G. Kappel, and M. Grossniklaus. “Cloud Modeling Languages by Example”. In: *Proceedings of International Conference on Service Oriented Computing and Applications (SOCA)*. IEEE Computer Society, 2014.

[10] A. Bergmayr et al. “Migrating Legacy Software to the Cloud with ARTIST”. In: *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2013, pp. 465–468.

[11] J. Bézivin. “On the Unification Power of Models”. In: *Software and System Modeling* 4.2 (2005), pp. 171–188.

[12] M. Brambilla, J. Cabot, and M. Wimmer. “Model-Driven Software Engineering in Practice”. In: *Synthesis Lectures on Software Engineering* 1.1 (2012), pp. 1–182.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. “CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms”. In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50.

[14] Chef Software Inc. *Chef Documentation*. http://learn.getchef.com/. [Online; accessed 25-November-2014]. 2014.

[15] D. Cliff. “Remotely Hosted Services and ’Cloud Computing’”. In: British Educational Communications and Technology Agency (BECTA), 2010.

[16] R. D. Cosmo, S. Zacchiroli, and G. Zavattaro. “Towards a Formal Component Model for the Cloud”. In: *Software Engineering and Formal Methods (SEFM)*. 2012, pp. 156–171.

[17] Dan Farber. *Oracleś Ellison nails Cloud Computing*. http://news.cnet.com/8301-13953_3-10052188-80.html. [Online; accessed 24-November-2014]. 2008.

[18] R. Di Cosmo et al. “Automated Synthesis and Deployment of Cloud Applications”. In: *Proceedings of the ACM/IEEE international conference on Automated Software Engineering*. ACM. 2014, pp. 211–222.

[19] Distributed Management Task Force Inc. *Cloud Infrastructure Management Interface (CIMI)*. http://dmtf.org/standards/cloud. [Online; accessed 24-November-2014]. 2013.

[20] Distributed Management Task Force Inc. *DMTF Accepts New Format for Portable Virtual Machines from Virtualization Leaders*. http://www.dmtf.org/news/pr/2007/9/dmtf-accepts-new-format-portable-virtual-machines-virtualization-leaders. [Online; accessed 24-November-2014]. Sept. 2007.

[21] Distributed Management Task Force Inc. *Open Virtualization Format (OVF)*. http://dmtf.org/standards/ovf. [Online; accessed 24-November-2014]. 2013.

[22] Docker. *Docker Documentation*. http://docs.docker.com/introduction/understanding-docker/. [Online; accessed 24-November-2014]. 2014.

[23] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. “Managing the Configuration Complexity of Distributed Applications in Internet Data Centers”. In: *Communications Magazine, IEEE* 44.3 (2006), pp. 166–177.

[24] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg. “Managing Multi-Cloud Systems with CloudMF”. In: *NordiCloud 2013*. Vol. 826. ACM International Conference Proceeding Series. ACM, 2013, pp. 38–45.

[25] S. Frey and W. Hasselbring. “Model-Based Migration of Legacy Software Systems into the Cloud: The CloudMIG Approach”. In: *Softwaretechnik-Trends* 30.2 (2010).

[26] S. Gilbert and N. A. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2 (2002), pp. 51–59.

[27] M. Harman and B. F. Jones. “Search-Based Software Engineering”. In: *Information & Software Technology* 43.14 (2001), pp. 833–839.

[28] M. Harman, K. Lakhotia, J. Singer, D. R. White, and S. Yoo. “Cloud engineering is Search Based Software Engineering too”. In: *Journal of Systems and Software* 86.9 (2013), pp. 2225–2241.

[29] T. J. Harmer, P. Wright, C. Cunningham, J. Hawkins, and R. H. Perrott. “An Application-Centric Model for Cloud Management”. In: IEEE Computer Society, 2010, pp. 439–446.

[30] T. J. Harmer, P. Wright, C. Cunningham, and R. H. Perrott. “Provider-Independent Use of the Cloud”. In: vol. 5704. Lecture Notes in Computer Science. Springer, 2009, pp. 454–465.

[31] L Haynes, D Leong, B Toombs, G Gill, and T Petri. *Magic Quadrant for Cloud Infrastructure as a Service*. Gartner, 2012.

[32] L. Herbert, C. F. Ross, and M. Grannan. “SaaS Adoption 2010: Buyers See More Options But Must Balance TCO, Security, And Integration”. In: Forrester Research, 2010.

[33] A. R. Hevner, S. T. March, J. Park, and S. Ram. “Design Science in Information Systems Research”. In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[34] R. M. Hierons, M. Harman, and S. Danicic. “Using Program Slicing to Assist in the Detection of Equivalent Mutants”. In: *Software Testing, Verifaction Reliability* 9.4 (1999), pp. 233–262.

[35] J. Hutchinson, M. Rouncefield, and J. Whittle. “Model-Driven Engineering Practices in Industry”. In: *ICSE*. 2011, pp. 633–642.

[36] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. “Empirical Assessment of MDE in Industry.” In: *ICSE*. 2011, pp. 471–480.

[37] S. Kent. “Model Driven Engineering”. In: *Integrated Formal Methods (IFM)*. Vol. 2335. Lecture Notes in Computer Science. Springer, 2002, pp. 286–298.

[38] M. Klems, J. Nimis, and S. Tai. “Do Clouds Compute? A Framework for Estimating the Value of Cloud Computing”. In: vol. 22. Lecture Notes in Business Information Processing. Springer, 2008, pp. 110–123.

[39] T. A. Lascu, J. Mauro, and G. Zavattaro. “A Planning Tool Supporting the Deployment of Cloud Applications”. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*. IEEE, 2013, pp. 213–220.

[40] J. A. Marpaung, M. Sain, and H.-J. Lee. “Survey on Middleware Systems in Cloud Computing Integration”. In: *Advanced Communication Technology (ICACT)*. IEEE Computer Society, pp. 709–712.

[41] J. Miller and J. Mukerji. “Model Driven Architecture (MDA)”. In: *Object Management Group, Draft Specification* (2001).

[42] Object Management Group, Inc., Architecture-Driven Modernization (ADM): Knowledge Discovery Metamodel (KDM), V. 1.3. *http://www.omg.org/spec/KDM/*. [Online; accessed 24-November-2014]. 2013.

[43] OCCI Open Grid Forum. *Open Cloud Computing Interface (OCCI)*. `http://ogf.org/documents/GFD.183.pdf`. [Online; accessed 24-November-2014]. 2011.

[44] D. Palma and M. Rutkowski. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. [Online; accessed 12-September-2014]. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html.

[45] D. Palma, M. Rutkowski, and T. Spatzier. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Simple Profile in YAML Version 1.0*. [Online; accessed 24-November-2014]. 2014. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html.

[46] A. Ranabahu, E. M. Maximilien, A. P. Sheth, and K. Thirunarayan. "A domain specific language for enterprise grade cloud-mobile hybrid applications". In: ACM, 2011, pp. 77–84.

[47] A. Sampaio and N. Mendonça. "Uni4Cloud: An Approach Based on Open Standards for Deployment and Management of Multi-Cloud Applications". In: *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. SECLOUD '11. ACM, 2011, pp. 15–21.

[48] B. Selic. "A Systematic Approach to Domain-Specific Language Design Using UML". In: *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*. IEEE Computer Society, 2007, pp. 2–9.

[49] B. Selic. "MDA Manifestations". In: *The European Journal for the Informatics Professional, IX (2)* (2008), pp. 12–16.

[50] J. Silva and O. Chitil. "Combining Algorithmic Debugging and Program Slicing". In: ACM, 2006, pp. 157–166.

[51] R. Soley. "Model Driven Architecture". In: *OMG white paper* (2000).

[52] Y. L. Sun, T. J. Harmer, A. Stewart, and P. Wright. "Mapping Application Requirements to Cloud Resources". In: vol. 7155. Lecture Notes in Computer Science. Springer, 2011, pp. 104–112.

[53] Y. Sun, J. Zhang, Y. Xiong, and G. Zhu. "Data Security and Privacy in Cloud Computing". In: *IJDSN* (2014).

[54] The Guardian. *The NSA Files*. http://www.theguardian.com/world/the-nsa-files. [Online; accessed 24-November-2014]. 2013.

[55] V. Tran, J. Keung, A. Liu, and A. Fekete. "Application Migration to the Cloud: A Taxonomy of Critical Factors". In: *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. SECLOUD '11. ACM, 2011, pp. 22–28.

[56] P. Wright, T. J. Harmer, J. Hawkins, and Y. L. Sun. "A Commodity-Focused Multi-cloud Marketplace Exemplar Application". In: IEEE Computer Society, 2011, pp. 590–597.