Utrecht University

Master Thesis Business Informatics

Dynamic Automated Selection and Deployment of Software Components within a Heterogeneous Multi-Platform Environment

June 10, 2015
Version 1.0

Author
Sander Knape
s.knape@students.uu.nl
Master of Business Informatics
Utrecht University

Supervisors
dr. S. Jansen (Utrecht University)
dr.ir. J.M.E.M. van der Werf (Utrecht University)
dr. S. van Dijk (ORTEC)

Additional Information

Thesis Title	Dynamic Automated Selection and Deployment of Software Components within a Heterogeneous Multi-Platform Environment
Author	S. Knape
Student ID	3220958
First Supervisor	dr. S. Jansen Universiteit Utrecht Department of Information and Computing Sciences Buys Ballot Laboratory, office 584 Princetonplein 5, De Uithof 3584 CC Utrecht
Second Supervisor	dr.ir. J.M.E.M. van der Werf Universiteit Utrecht Department of Information and Computing Sciences Buys Ballot Laboratory, office 584 Princetonplein 5, De Uithof 3584 CC Utrecht
External Supervisor	dr. S. van Dijk Manager Technology & Innovation ORTEC Houtsingel 5 2719 EA Zoetermeer

Declaration of Authorship

I, Sander Knape, declare that this thesis titled, *'Dynamic Automated Selection and Deployment of Software Components within a Heterogeneous Multi-Platform Environment'* and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: June 10, 2015

Acknowledgements

This master thesis report was performed as a graduation project for the Business Informatics master program at the Utrecht University. During the entire research project, ORTEC provided me with a great place to work for which I am very thankful.

My research was guided by many skilled and experienced people who aided me in the successful progression and finalization of this research project. Slinger Jansen and Jan Martijn van der Werf have been invaluable in the creation of an academical-worthy thesis. Even though Slinger was scarcely available due to responsibilities all the way around the world, he always found the time to provide me with some much-needed feedback or answer any question I had.

At ORTEC, I met many new intelligent and skilled people from whom I learned a great deal. Steven van Dijk guided me through the entire process of the design research and often put me on the right track when I lost focus or needed to structure all the information floating around in my head. Vincent taught me a lot about programming and I enjoyed our talks about software architecture, design patterns, and the like. His technical guidance was invaluable for the development of the prototype. Even after he was detached to an off-site project, he always found the time to help me with my project. With Chuck, I had many enjoyable talks about the agile methodology, which is a very interesting topic on its own. Niels and Thijs were always available for a good talk – if they were around – and I could always come to them if I had any questions. Finally, I became good friends with Mick, my fellow intern with whom I shared many challenges related to graduation and especially, writing.

For some much-needed distraction and relaxation, I'd like to thank Frank Underwood, Oliver Queen, Harvey Specter, Mike Ross, Jay Pritchett and family, and last but not least Tyrion Lannister and friends (and foes).

Abstract

Cloud computing is a new paradigm in computing where both services and infrastructure are provided on a pay-as-you-go basis. In particular, Platform as a Service (PaaS) offerings are becoming popular as it requires less maintenance than the more traditional Infrastructure as a Service (IaaS) offerings. The cloud environment however is highly dispersed, subsequently creating a heterogeneous environment. Each cloud provider is different, and using multiple providers or migrating to another cloud is very time consuming and therefore, expensive.

The multi-cloud paradigm optimizes to decrease costs and optimizing quality by leveraging multiple cloud providers simultaneously. With cloud providers not standardizing anytime soon, one solution to decrease migration costs is by developing a multi-cloud broker that is able to deploy an application to multiple cloud providers. In addition, the mere fact that multiple cloud providers are now available for use adds the question of where to deploy a software application.

This research therefore proposes a method that can automatically select, configure, and deploy an application within this highly heterogeneous cloud environment. Modeling both the application and the cloud environment is achieved through combining the modeling language TOSCA and feature models. By adding user-defined constraints such as costs and hardware configurations, it becomes possible to select an optimal cloud provider for each application component. The now generated deployment scenario is then automatically deployed to the selected cloud providers. A prototype implements by combining the design science methodology with experimentation.

The PaaS environment is more dispersed than expected. Many implementation differences exist between each cloud provider and the question arises whether any commercial incentive may exist for developing such a flexible method. In particular, we find that a proper model is needed that describes the PaaS environment, capturing all differences and similarities. As not much research yet exists that tackles this problem specifically for the PaaS environment, our findings can prove to be an initial starting point for such an attempt.

Table of Contents

Chapter 1: **Introduction**

Cloud computing is a new paradigm in computing where both services and infrastructure are provided on a pay-as-you-go basis (Mell & Grance, 2011), with many companies deploying their services and executing their computations in the cloud. Gartner research expects that in 2015 more than 180 billion dollars will be spent on cloud services globally (Flood, 2013). In particular, *Platform as a Service* (PaaS) offerings are becoming popular (DZone, 2014) as it requires less maintenance than the more traditional *Infrastructure as a Service* (IaaS) offerings (Lawton, 2008; Petcu, 2013a). Utilizing PaaS, software vendors can quickly build software while not spending time on management and maintenance of hardware, OS, and network.

Cloud computing has a number of advantages; for example, it involves no up-front investment, it's easily scalable and maintenance costs are reduced (Marston, Li, Bandyopadhyay, Zhang, & Ghalsasi, 2011; Zhang, Cheng, & Boutaba, 2010). However, disadvantages also exist, the most notable one being vendor lock-in (Toosi, Calheiros, & Buyya, 2014). The cloud ecosystem suffers from a lack of standardization and as a result, differences in cloud provider services, subsequently creating a heterogeneous environment (Crago, Dunn, & Eads, 2011; Petcu, 2011). Once a cloud provider has been chosen, the technology gets tied to that specific provider, hindering the migration to a different provider (Louridas, 2010). Suddenly, initial cloud provider selection is extremely important. It is difficult to make flexible changes when a cloud provider changes its terms (Bradshaw, Millard, & Walden, 2011) or when the requirements of the application change in such a way that is not supported by the currently adopted cloud provider. This is especially the case with PaaS as these services usually have their own, custom APIs. Entire virtual machines can typically be copied with IaaS services, which fewer changes to adapt to the new provider.

As a response to the above-mentioned challenges, the multi-cloud deployment paradigm is suggested to bridge the gap between cloud providers (Buyya, Ranjan, & Calheiros, 2010). This approach promises several improvements compared to the single-cloud method, such as optimizing costs and quality of the deployed application, the ability to react to new offerings by cloud providers, comply to new constraints (e.g. physical data location as directed by privacy laws (Wang, Wang, Ren, & Lou, 2010)), and, of course, no dependence on one single cloud provider (Petcu, 2013b). Overall, multi-cloud deployment is either achieved by standardizing a set of cloud providers or by creating an overarching layer that abstracts away the specific implementation details by providing a single interface for multiple providers (Petcu, 2011). Whereas most existing research focuses on variability aspects of IaaS deployments (Lucas-Simarro, Moreno-Vozmediano, Montero, & Llorente, 2013; Tordsson, Montero, Moreno-Vozmediano, & Llorente, 2012), this research will focus on PaaS deployment in combination with local deployment.

Cloud adoption – e.g., the process of moving software application to the cloud - has several use cases, one being partial migration (Andrikopoulos, Binz, Leymann, & Strauch, 2012). With modern software systems typically being component compositions, it is possible to deploy only a subset of specific components in the cloud. One example is to move certain computational heavy processes to the cloud in order to speed up certain calculations. Another example is the case of privacy-sensitive data, where the data is stored locally but presentation aspects are stored within the cloud, closer to the customer (Andrikopoulos et al., 2012). Instead of deployment on multiple cloud providers (e.g. multi-cloud), we therefore desire a more general multi-*platform* setup, where local deployment is included in the solution space. Achieving a flexible method that allows these types of use cases is a challenging endeavor (Wettinger, Andrikopoulos, Strauch, & Leymann, 2013).

When considering a multi-cloud environment including local deployment and PaaS providers, a large variability of options exist as each component can be deployed on a number of locations. Considering all options would be time-consuming and error-prone, especially when one needs to deal with the implementation differences between providers (Binz, Breitenbücher, Kopp, & Leymann, 2014). An automated method is to be preferred, in such a way that all possible deployment configurations are considered and tested. Indeed, the intrinsic changes seen in the cloud environment including the proliferation of cloud services, as well as changes to end-user requirements, calls for a dynamic, automated method (Jula, Sundararajan, & Othman, 2014; Sun, Dong, Hussain, Hussain, & Chang, 2014). After selection of a specific configuration, the application should automatically be configured and deployed while taking care of provider-specific implementations.

1.1 Problem Statement & Objective

Automating the process of selecting, configuring, and deploying a component-based application within a heterogeneous multi-cloud ecosystem comes with its own challenges. In light of this research, we separate this research into two distinctive challenges, with the second challenge building on the first one.

1.1.1 Challenge 1: Modeling the application and environment for automated selection

First, to accomplish automation, the application and the environment will need to be modeled in a standardized, machine-readable format (Binz et al., 2014). Several modeling languages exist that can accomplish this, with *The Topology and Orchestration Specification for Cloud Applications*, or TOSCA, having been designed specifically for the cloud environment (Binz et al., 2014). TOSCA is a recently accepted OASIS standard (OASIS, 2013) that can model components, their relations and their management. For example, it can model an 'Application' (component) that 'requires' (relation) an 'Apache Web Server' (component) and 'connects to' (relation) a ‘MySQL Database’ (component). In addition, TOSCA describes management plans that can invoke a series of operations from different components and relations in a specified order (Binz et al., 2014). Using implementation artifacts and deployment artifacts, actual services or scripts are attached to both components and relations used for installation, configuration, and deployment on specific environments. How TOSCA is able to model the variety of possible deployment locations is especially of interest.

Assuming that both the application and the environment are modeled, a specific *deployment scenario* will need to be generated. In this context deployment scenario is defined as “*a software and systems* *configuration that satisfies all constraints of the decision maker, and thereby enables a system to provide a valuable function for its end-users*" (Jansen, 2014). The modeled environment can be considered a 'solution space', where a number of deployment scenario options may be available (Garcıa-Galán, Rana, Trinidad, & Ruiz-Cortés, 2013). Given constraints such as pricing, performance, and location of data, each possible configuration should be tested so that an acceptable deployment scenario (or preferably, the 'optimal' scenario) is generated. In addition, selecting a specific deployment location for a component may restrict the deployment locations of other components. All in all this leads to a large variability of deployment scenario options. Although different methods exist that support the decision making of cloud providers, little research focuses on simultaneous deployment on both cloud and local locations and current methods require a lot of technical knowledge (Sun et al., 2014). In addition, to achieve a fully integrated model, the extensibility of TOSCA will need to be researched to enable deployment scenario selection.

The main deliverable of this challenge is a deployment scenario presenting one or more viable solutions for deploying the given application's components on selected platforms.

1.1.2 Challenge 2: Automated multi-platform deployment of portable software

Given the deployment scenario as delivered by challenge 1, the next challenge is to deploy the application's components at the provided location(s). A number of issues are related to this challenge. First, automated provisioning and deployment of services such as databases, message queues, and load balancers require different actions for different platforms, also known as the *portability* of an application. A method is therefore required to abstract away these platform-specific actions. Next, each component may require communicating with another component; e.g., a compute component may need to get access to a database. This *interoperability* aspect therefore requires the connecting of components of which the locations are unknown until actual deployment. Last, the written software code will need to be able to run on whatever platform it is deployed. Again, platform-specific implementations will need to abstracted away to allow code to be written once, and deployed on each of the supported platforms.

The main deliverable of this challenge is a method that allows component-based software to be deployed as a multi-platform application. In combination with challenge 1, the combined method allows both automated selection and deployment. The final delivered method will be validated using a component-based application written in .NET. A number of cloud services will be modeled and a set of use cases will test a variety of deployment situations.

1.2 Academic & Societal Contribution

The final deliverable of this research will be a method that allows for automated (simultaneous) multi-platform deployment without being dependent on a middleware that would result in vendor lock-in. This section describes the resulting scientific and societal contributions.

1.2.1 Scientific Relevance

The defined challenges in the previous section have all been researched in one way or another, which is why part of this research is the identification of existing methods for those challenges. A method encompassing all challenges - selection, configuration and deployment - however is not available. A contribution of this research therefore will be a state-of-the-start overview of existing methods, compared and considered for integration within a single, automated method. In addition, with TOSCA serving as the backbone of the final delivered method, both selection and deployment with be integrated into a single, automated method. With TOSCA being suited to facilitate the automated deployment, of interest is how to extend it with automated selection. This is, as far as our knowledge goes, not yet done, though mentioned as an interesting research direction by several authors (Brogi, Soldani, & Wang, 2014; Sun et al., 2014). Finally, investigating issues related to cloud service selection is interesting, as this problem may not satisfactorily be solved by simply extending TOSCA (Brogi et al., 2014). The TOSCA ecosystem may well be extended, as contained within this research is the modeling of existing cloud services.

1.2.2 Societal Relevance

The existing approaches are very theoretical. They are modeled to eliminate many of the practical challenges businesses face when moving their applications to the cloud. In reality, partial and perhaps even phased migration is of interest to the business as this allows them to slowly learn how to adapt their software for the cloud. A recent study shows that migration and integration of legacy systems to the cloud, as well as a lack of internal processes, are considered key roadblocks towards cloud computing adoption (Columbus, 2013). The developed method in research will provide an answer to both roadblocks, with the method serving as an integrated process that allows partial migration of new and existing applications. Codification of developed components and modeled services is also of interest, as this allows software developers to re-use previous work quickly for new and existing applications.

Chapter 2: **Research Approach**

In order to properly structure the proposed research, a main research question and related sub research questions are defined. All questions are linked to one or more of the previously defined challenges. Section 2.1 discusses the research questions. Next, section 2.2 discusses the context wherein the research is conducted. Two research methods are used for proper execution of this research. Section 2.3 explains the utilized design science, after which section 2.4 properly defines the experimentation phase of the research. Section 2.5 provides an overview of the entire research execution, after which we will finalize with some notable challenges and limitations in section 2.6.

2.1 Research Questions

The main research question is as follows:

RQ: "*How can cloud selection, configuration, and deployment be fully automated?"*

In order to answer the research question, the following sub questions are defined:

SQ1: "*How can both the application and the environment be modeled to facilitate automated selection?*"

As stated, TOSCA is used to model both the application and the environment, though of interest is how to facilitate the automated selection. An additional contribution of this research will therefore be the validation of TOSCA within the context of this research.

SQ2: "*Which constraints can be modeled to automatically select a deployment scenario for software components?"*

Of interest is which constraints are available and quantifiable in such a way that a comparison can be made. Also of interest is whether these constraints can be modeled within TOSCA, or that some extension of TOSCA or even a separate configuration method is required to model the constraints.

SQ3: "*What methods exist to select a deployment scenario within the heterogeneous cloud environment including local deployment?*"

Sub questions one and two will provide the modeled data required to select a possible deployment scenario. The third sub question will use that data to acquire an actual deployment scenario.

SQ4: "*How can the components of a software application automatically be configured and deployed?*"

In other words, how are the individual components deployed on the correct locations given a generated deployment scenario? Location-specific information is required and will need to be both stored and executed.

2.2 Cooperation with ORTEC

The research is conducted with the support of ORTEC within the context of an internship. ORTEC develops planning- and optimization-related software and provides related services such as consulting and support. Their software often needs to handle large amounts of data and many computational resources are required to perform optimization algorithms. Currently, the software is deployed either on local machines or on server farms maintained by the client using the software. ORTEC foresees possible advantages of deployment in the cloud, such as lower costs for their clients and increased processing speed. An initiative is therefore currently in progress that looks into methods to deploy new and existing software on the cloud.

ORTEC has shown interest in this research and can provide support in a number of ways. First, ORTEC can provide knowledge concerning popular cloud platforms' architectures, deployment, and configuration. This knowledge can considerably facilitate and boost the practical, technological activities required for performing the research. Second, ORTEC can provide one or multiple existing applications that are fit for (partial) deployment on (multiple) cloud providers. Finally, using both the knowledge and application(s), the developed method is tested and validated using a range of use cases.

2.3 Design Science

This research is categorized as the design-science paradigm, defined as a research method that "*creates and evaluates IT artifacts intended to solve identified organizational problems*" (Hevner, March, Park, & Ram, 2004). The seven guidelines (Hevner et al., 2004) as part of this method are used to answer the above stated research questions. Defining and utilizing the process-oriented guidelines will aid in creating a structured approach for performing the research. Below, each guideline is presented and defined within this research context.

- *Problem relevance.* To summarize the problem statement previously define, the cloud ecosystem is currently fragmented in such a way that migration to another cloud provider is no trivial task. As standardization initiatives are up to now not fruitful, a method that allows easy replacement of a specific (cloud) service is to be preferred. As each software component can be deployed on a number of locations, considering all options would be time-consuming and error-prone. This calls for a flexible, automated method that can automatically generate a deployment scenario and both configure and deploy each software component.

- *Design as an artifact.* The guidelines prescribe that a viable artifact must be created, which can be either a construct (vocabulary and symbols), a model (abstractions and representations), method (algorithms and practices) or an instantiation (an implemented or a prototype system). This research will produce a prototype that is able to perform all of the previously defined activities.

- *Research rigor.* Both the construction and evaluation of the produced design artifact requires rigorous methods. To learn from existing initiatives attempting to solve similar or sub-problems, a literature review is conducted. As stated, the research will be conducted at ORTEC. During development of the prototype, a component-based .NET application will be used for experimentation. Experts from ORTEC will test and review the prototype for evaluation.

- *Design as a search process.* When searching for the most effective design artifact, the artifact must use the available means whilst satisfying the legislation in the problem environment. The guidelines state that design science is inherently iterative, continually designing and testing the created artifact. In addition, the scope of the design problem is expanded during this research. This is achieved through the two previously stated research challenges. Each challenge adds a new dimension to the developed artifact.

- *Design evaluation.* The guidelines state that the utility, quality, and efficiency of the design artifact should be demonstrated through well-executed evaluation methods. As the final deliverable of this research will be a prototype, evaluations should demonstrate that its functionality performs better compared to a manual approach. ‘Better’ in this context is further defined in the next section. Experts at ORTEC will review deployment as well the extensibility of the prototype.

- *Research contributions.* The design-science research must provide a verifiable contribution in the area of the design artifact, design foundation and/or design methodology. This research will produce an automated approach for multi-platform selection, configuration, and deployment. This approach should save time and perform the task with fewer errors than a traditional manual method. In addition, the method will allow for a more flexible deployment scenario, where the location of each component can easily be selected and the ‘wiring’ of these components is performed behind the scenes.

- *Communication of research.* The guidelines state that the research should be communicated both to technically oriented (for reproduction) and to management oriented (for acceptation) audiences. The produced thesis will serve for both audiences. In addition, a guide is created that provides a more in-depth view of the prototype.

These seven guidelines will aid in the proper development of the prototype. In addition, we extend the discussed design evaluation with a solid experimentation phase.

2.4 Experimentation

As is mentioned in the previous section, a .NET application is used both for construction and for evaluation of the integrated method. First, by means of iterative experimentation, the application is used to test the viability of the constructed method. Second, after the method is considered developed enough, a new application will in another programming language will be used to test the extensibility and therefore generalizability of the prototype. Of course, of interest is to what extent the prototype can be used for *any* software application.

With the above in mind, a proven experimentation structure is used to ensure a proper research setup (Wohlin et al., 2012). Experimentation is, amongst others, appropriate for evaluating the accuracy of a model, e.g. to see if they deliver what is expected (Wohlin et al., 2012, p. 17).

The experimentation setup includes scoping, planning, operation, analysis and interpretation, and finally presentation and packaging. It should be noted these phases do not strictly follow a waterfall model; it may be '*necessary to go back and refine a previous activity*' (Wohlin et al., 2012, p. 78). The phases are defined in the following subsections.

2.4.1 Scoping

In this phase, the goals of the experimentation are defined in order to answer *why* this research is conducted. This phase is facilitated by the "goal, question, metric" template (Caldiera & Rombach, 1994);

- *Object of study.* The object of the experimentation is the prototype that will be constructed following the guidelines by Hevner et al. (2004), as defined in the previous section. This prototype will include the selection, configuration, and deployment of a component-based application within a heterogeneous, multi-platform environment.

- *Purpose.* The intention of the experimentation is to validate the constructed prototype, identified through comparison of the results with experts' opinion. Given the relative narrow scope, e.g. the small amount of variables, we consider it possible to validate the results of the prototype through expert opinion.

- *Quality focus.* The prototype should be *correct*, should perform within a *reasonable time* (faster than the 'traditional' approach) and should be *repeatable* (e.g. without breaking anything).

- *Perspective.* The prototype will automate actions normally performed by a system engineer or developer. In this sense, a developers' perspective is certainly of interest considering applicability within a real-life environment. In addition, the author will evaluate the method from a research perspective in order to identify generalizability.

- *Context.* As stated, the research is conducted at ORTEC within the context of an internship. At ORTEC, several experts are available for support and validation of the constructed prototype.

Overall we are interested in the applicability of the prototype within a practical environment, thus we perform experimentation with existing software to test the feasibility.

2.4.2 Planning

After having defined the reasons for conducting the research, the next question is *how* this research will be conducted. The experimentation will be performed with two off-line (not on live, running applications) applications, considering both toy/artificial and real problems that are generalizable, and the study is mainly conducted by a student. The formal hypothesis is stated as follows:

"*The automated approach towards selecting, configuring, and deploying a component-based application is just as accurate as and faster than performing the same activities manually*".

The answer to SQ2 will provide the required variables for the experimentation. SQ2 is defined as:

> **SQ2:** "*Which constraints can be modeled to automatically select a deployment scenario for software components?"*

These constraints will serve as the *independent variables* that will be changed in order to identify changes to the *dependent variables*, which is the selected deployment scenario. An example is that increasing the independent variable *price* of a specific cloud database service, should change the *deployment scenario* so that a different, cheaper cloud database service may be chosen. Selection and manipulation of these variables will be expert-based, with the aim of choosing real-life situations. Randomization is not considered possible as the constraints themselves are constrained. For example, randomization may result in the situation where business logic is to be deployed on a database server. With the defined narrow scope in the previous section in mind, we consider it feasible to set up a set of constraints based on experts' opinion. These can then be validated against the expected outcomes, also determined by the experts. No statistical analysis will be performed.

The two instruments for performing the experimentation are two applications in two different programming languages. Another variable is the location of the deployment; different constraints should deploy an application on another platform. Also of importance in this phase is to consider the question of validity of the results we can expect. Four major classes of validity are to be considered (Cook, Campbell, & Day, 1979), each discussed separately;

- *Conclusion validity.* This is sometimes referred to as statistical conclusion validity, as this class is concerned with a significant relation between what goes in (constraints, independent variables) and what comes out (a deployed application, dependent variable). As is mentioned, neither randomization nor statistics will be performed. Thus, this validity is tackled through expert reviews that should ensure correctness of trustworthiness of the used methods.

- *Internal validity*. We have to make sure the witnesses experiment results are actually *caused* by our method and not by something unexpected or unpredictable we have no control over (correlation is not causation). As the method is technology-centered (programming code), the quality and validity of the code has to be insured. This is done by experts working at ORTEC.

- *Construct validity.* This validation class is concerned with the relation between the theory and practice. Threats to this validity are tackled by a literature study, learning from previous similar research, and again by programming experts who can assure proper quality of the delivered code.

- *External validity.* To what extent is the method generalizable? One of the limitations of this research (see section 2.6) is the fact the delivered method is constructed in .NET, which is less present within the cloud ecosystem compared with languages like NodeJS, PHP, or Ruby. A goal of this research is therefore to extend the constructed prototype with capabilities for another programming language and test the extensibility and generalizability of the prototype.

The delivered prototype will automate a process currently performed manually by software and system engineers. Therefore, we consider it of vital importance to include experts in the field in the validation step.

2.4.3 Execution

After scoping and planning, we execute the experimentation, divided amongst three different phases.

- *Operation.* As the experimentation deals with programmatic code and not human beings, this phase should be straightforward. As preparation for the experimentation, an application will be chosen for initial validation. Executing the experiment equals the utilization of the constructed method with the two applications. Each step of the method (selection, configuration, deployment) will be logged for data validation, in order to ensure that each step is executed as expected considering the input from the previous step.

- *Analysis and interpretation.* In this research setup, data analysis and interpretation is relatively trivial, as the outcomes are compared with expected outcome. This phase, together with the previous one (operation) is the iterative part of this research as these will be repeated until the expected results are seen.

- *Presentation and package.* The results will thoroughly be discussed within the final thesis and proper documentation will be set up for duplication within the company. Replication of the experiment will be facilitated by means of documentation and possibly by making the code open source.

Supplementing the design science approach with experimentation provides us with a clear framework and scope for our literature research and prototype development.

2.5 Research Model

Combining the previously defined challenges, research questions and research methods, a model can be defined listing all aspects of the research. This is shown in Figure 2.1.

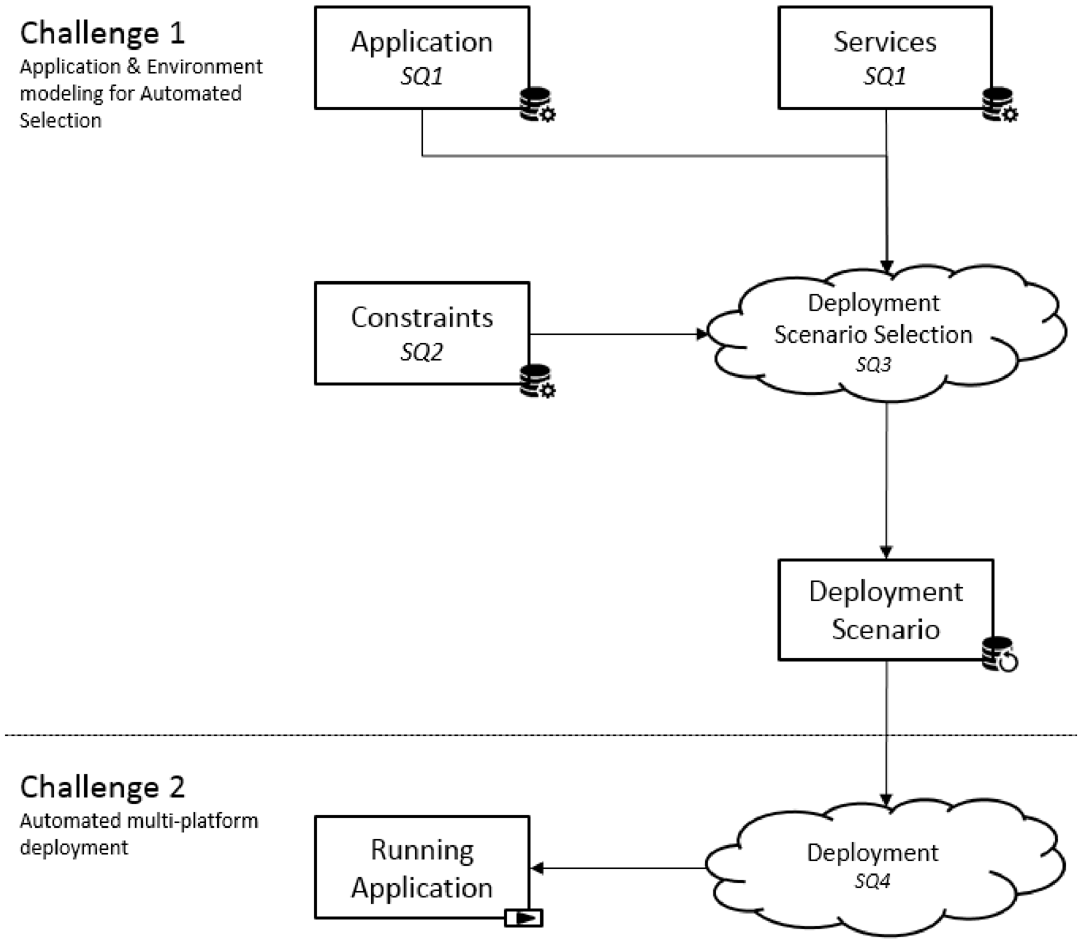

Challenge 1
Application & Environment modeling for Automated Selection
Application
SQ1
Services
SQ1
Constraints
SQ2
Deployment Scenario Selection
SQ3
Deployment Scenario
Challenge 2
Automated multi-platform deployment
Running Application
Deployment
SQ4

Figure 2.1: Research model showing activities and related research questions

Three different modeled aspects serve as the input for selection a deployment scenario: the application, the (cloud) services and the constraints. The first two will be modeled in TOSCA and are the main deliverable of challenge 1. Next, research will show how the constraints can be added to the knowledge base. A selected deployment scenario will need to be converted to an actual running application, which is the main deliverable of challenge 2.

2.6 Challenges & Limitations

Though not considered one of the main objectives or contributions of this research, the final deliverable will require a significant amount of programming in order to validate the produced method. It will be a challenge to write portable software that will be able to operate within the heterogeneous multi-platform environment. Besides being able to be run within a number of environments, the software will require a modular approach that allows agnostic communication between the components. For example, a logic sample code will not know where the database component resides. With the available programming experience available within ORTEC, where the research is conducted, it is believed this challenge can be overcome.

The developed method and application will be developed within the .NET environment. This can be considered a limitation as .NET is supported by a relative small part of the cloud environment. It will be important to properly define the generalizability of the conducted research, and show how other application types can also be used as input for the method.

Chapter 3: **Fundamentals**

This chapter introduces the fundamental background concepts relevant for this thesis. The chapter begins with a general introduction of cloud computing in section 3.1. Portability and interoperability challenges in cloud computing are introduced in section 3.2. Next, the many facets of multi-cloud computing are explained in section 3.3. In section 3.4 we describe TOSCA, used the backbone for the as method delivered by this research, extended with feature models which are explained in section 3.5. The last section, 3.6, summarizes some important statements and conclusions made in this chapter.

3.1 Cloud Computing

The emergence of cloud computing has fundamentally changed the IT landscape. Cloud computing essentially moves management and maintenance activities towards the cloud provider, relieving the customer from buying and maintaining hardware for their IT environment (Marston et al., 2011). Resources such as services, platforms and infrastructure are dynamically provisioned to cloud users, who typically only pay for what they use (Armbrust et al., 2009). The concept can be considered an umbrella term, as it encompasses many use cases such as:

- Storage of data in the cloud (e.g. iCloud, Dropbox or Google Drive);
- Running software in the cloud (e.g. Microsoft Office 365, Google Docs or Facebook);
- Setting up (virtualized) infrastructures in the cloud;
- Niche services such as *cloud gaming* (games are run in the cloud and streamed to local devices).

Cloud computing is not new. The concept itself has been introduced as early as in the 1960's (Parkhill, 1966), at that time described as "*computing as a utility*". The term "cloud computing" became mainstream after Google's CEO Eric Schmidt used the term in 2006 during a search engine conference[1], explaining the new concept as a logical next step succeeding the client-server paradigm. Only two weeks later, Amazon published a press release announcing the "*Amazon Elastic Compute Cloud"* service (Amazon, 2006). At this time, people started using the term "cloud computing", especially for marketing purposes, although no one really knew what exactly it meant.

In this light, a number of researchers and organizations began standardizing the definition of cloud computing. One seminal work compares over 20 definitions (Vaquero & Rodero-Merino, 2008), resulting in the following encompassing definition for cloud computing;

> *"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms, and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of*

[1] The conversation in which the term was used by Eric Schmidt can be found at: http://www.google.com/press/podium/ses2006.html

resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs."

Though broad, the definition misses one characteristic currently considered key to cloud computing. The virtualized resources (such as hardware, development platforms, and/or services) are *shared* across a number of customers. This means that cloud providers effectively need to balance their hardware resources in such a way to optimize their energy consumption and subsequently lower their costs (Beloglazov, Abawajy, & Buyya, 2012). A different definition that is embraced by the research community is the one by the National Institute of Standards and Technology (NIST);

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." (Mell & Grance, 2011)

In addition, NIST defines a list of five essential characteristics of cloud computing, as well as three possible service delivery models and four deployment models. As these provide a broad explanation of cloud computing, they will be explained in the next sections. Finally, some advantages to cloud computing are explained that in large part account for why the paradigm has become so immensely popular.

3.1.1 Essential Characteristics

The following key characteristics are defined by NIST in order to clearly distinguish cloud computing from similar concepts such as grid computing, utility computing and virtualization (Mell & Grance, 2011; Zhang et al., 2010);

1. *On-demand self-service*. The computing capabilities such as computational time and storage can unilaterally and automatically be provisioned to the customer. In other words, secondary support activities such as procurement are no longer required, effectively optimizing the software supply chain (Porter, 2008).

2. *Broad network acces*s. Access to the cloud is available through multiple device types, not limited to laptops and desktops but also handhelds such as mobile phones.

3. *Resource pooling*. The provided resources are pooled to serve multiple, multi-tenant customers at the same time. Different physical and virtual resources are dynamically assigned and re-assigned with changing demand. Consequently, a customer does not know the exact location of the provided resources, although many cloud providers provide the functionalities to control this on a high level, such as country, state, or data center. This is especially important for privacy-sensitive data that, for example, cannot leave a specific country (Pearson, 2009).

4. *Rapid elasticity*. Cloud elasticity is defined as "*the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such*

that at each point in time the available resources match the current demand as closely as possible" (Herbst, Kounev, & Reussner, 2013). In this sense, a cloud provider that automatically provisions and de-provisions resources can be considered 'more elastic' than a cloud provider can where manual activities are required.

5. *Measured service*. In order to provide resource pooling and rapid elasticity, some form of monitoring is required, which provides the necessary data to make decisions. Examples of what is monitored are storage, processing, bandwidth and active user accounts. In addition, the data can transparently be showed to the customer.

In addition, two other key characteristics not explicitly defined by NIST are (Armbrust et al., 2009);

1. The elimination of an up-front commitment by cloud users. In cloud computing, you typically pay for what you use. This is an important characteristic as it allows companies to undertake innovative opportunities without large initial investments in hardware resources.

2. The ability to pay for use of computing resources on a very short-term basis (e.g. by the hour or minute). This characteristic allows for provisioning of resources only when they are actually needed, without extra costs. This motivates cloud users to let go of resources not required, and subsequently allows cloud providers to optimize their resource usage (*pooling*), possibly turning off equipment that is not required leading to reduced costs (Beloglazov et al., 2012). This is an important characteristic as it may lead to significant reductions of computing resource power requirements (Baliga, Ayre, Hinton, & Tucker, 2011).

3.1.2 Service delivery models

Figure 3.1 shows an overview of the available service models, including some examples. A more elaborated set of examples are given in Box 3.1. NIST defines the following three service models (Mell & Grance, 2011);

1. *Software as a Service (Saas)*. This layer provides services or compositions of services to users that usually require no technical knowledge. These services are available through web browsers (e.g. webmail, Google Docs) or are offered as applications (e.g. iTunes, Picasa) that store data within the cloud.

2. *Platform as a Service (PaaS)*. This layer provides a development platform to software vendors where they can design, develop, and test their applications. Users are bound to the programming languages, libraries, services, and tools supported by the cloud provider.

3. Infrastructure as a Service (IaaS). This layer provides a virtualized infrastructure to the user, e.g. storage, network, processing, and other fundamental computing resources. This layer provides the most flexibility as users can deploy their own arbitrary software such as operating systems and applications.

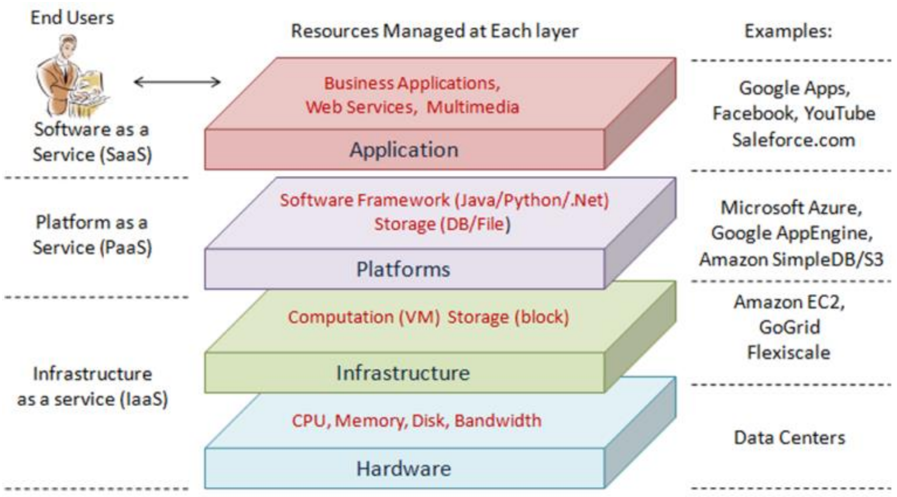

End Users
Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a service (IaaS)
Resources Managed at Each layer
Business Applications, Web Services, Multimedia
Application
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Computation (VM) Storage (block)
Infrastructure
CPU, Memory, Disk, Bandwidth
Hardware
Examples:
Google Apps, Facebook, YouTube Saleforce.com
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Amazon EC2, GoGrid Flexiscale
Data Centers

Figure 3.1: The Cloud Computing Architecture (Zhang et al., 2010)

Some other service model definitions have been proposed, although none are as famous as the above three and are usually contained within one of those. The fourth service model visible in Figure 3.1 is Hardware as a Service (HaaS). This layer is responsible for the management of the actual hardware, such as servers, routers, switches and power (Zhang et al., 2010). This layer is solely of interest to the cloud provider, as the cloud user has no control over this layer except in some very special cases where enterprise users lease an entire data (Rimal, Choi, & Lumb, 2009).

Other service models mentioned in literature are Database/Desktop/Development as a Service (DaaS), Framework as a Service (FaaS), Testing as a Service (TaaS) and even Business as a Service (BaaS) or Organization as a Service (OaaS) (Rimal et al., 2009; Silva & Lucrédio, 2012). Often combined in the overarching 'Everything as a Service' (XaaS), these service models can be contained within one of the three service models as defined by NIST. They do however show the many facets of cloud computing and how it encompasses every aspect of the software lifecycle.

3.1.3 Deployment models

A cloud deployment model can be considered a 'cloud type', and specifies one of the important decisions a company has to make when adopting the cloud. NIST defines the following deployment models (Mell & Grance, 2011);

1. *Private Cloud.* This model gives exclusive rights to a single organization who alone provisions the hardware and infrastructure provided. The hardware may exist on or off premises.

2. *Community Cloud.* Similar to the private cloud, this model gives exclusive rights to a set of organizations who share similar demands and concerns, usually related to privacy issues. One example may be where a set of hospitals use the same cloud provider who adheres to standards for dealing with sensitive patient related data.

3. *Public Cloud.* This is the cloud as it's known in the most common sense. The cloud infrastructure is available to the general public. The actual hardware exists on the premises of the cloud provider.

4. *Hybrid Cloud.* This model is a combination of two or more of the above deployment models. The two combined models remain distinct and unique but complement each other. Some standardization or middleware is required to communicate with both providers. One hybrid cloud example is the issue of *cloud bursting,* where for example a public cloud can be used for increased computational power in the event of a high load (Nair et al., 2010).

Some deployment model definitions closely related to the hybrid cloud model are federated cloud, inter-cloud and multi-cloud (Grozev & Buyya, 2012). Each of these definitions combine at least two public or private clouds, but with small conceptual differences. These concepts will more thoroughly be discussed in section 3.3.

Examples of each service model

To provide some more clarity on the differences between IaaS, PaaS and SaaS, some examples of each service model are provided.

- *Software as a Service*. Using Dropbox, a user can store his or her files in the cloud and access them from any location and device with an internet connection. With Office 365, documents can be both stored and edited in the cloud. The key characteristic is that entire software packages are provided to the user, which can thus be used by the non-technical audience.
- *Platform as a Service*. Microsoft Azure provides a platform on which applications can be deployed that are written in .NET, NodeJS, Java, PHP and others. In addition, services are provided that allow easy use of for example a database, a service bus or for file storage. Google App Engine provides similar functionalities but for example does not support the .NET runtime. Key characteristic is that the environment on which the programming language or the service runs is provided and is maintained by the cloud provider.
- *Infrastructure as a Service*. The biggest and best-known player in the IaaS market is Amazon. Other providers offering similar services are for example RackSpace and GoGrid. Microsoft and Google also offer IaaS services; Amazon also provides PaaS offerings. With IaaS, a computing infrastructure is provided on which cloud users typically install entire virtual machines (VMs). These VMs are then for example provisioned with a programming environment (e.g. Apache + PHP) and other services such as a database (e.g. MySQL). Difference with PaaS is that cloud users have more control over the underlying environment. A new PHP version can be installed whenever the cloud user wants to. This, of course, also means that more time needs to be spend on maintenance of the environment.

Box 3.1: Examples of each cloud service model

3.1.4 Advantages, issues and challenges

Although the cloud offers many revolutionary promises, it is not (yet) ready for everyone. A number of advantages and challenges are associated with cloud computing. It should be no surprise that some of the advantages will be closely related to the previously defined key characteristics of cloud computing. As we will see, a number of the challenges can be tackled within the *multi-cloud* paradigm, which will be discussed in section 3.3. The advantages can be summarized as follows (Marston et al., 2011; Zhang et al., 2010);

- *No up-front investment*. Previously, when someone had a great idea for an application accessible through the internet, large initial costs were required through the acquisition of IT hardware. This impedes innovation, as people may decide the risk is too high. Cloud computing lowers these barriers; you only pay for what you use. Costs only rise if the application indeed becomes popular and requires more resources.

- *Highly scalable.* Another problem associated with the initial hardware equipment acquisition was the *amount* of hardware that was required. Would the developed application become a success, the large amount of users could possibly shut down the application in the situation where not enough hardware had been acquired. Acquiring too much hardware, on the other hand, might prove to be problematic in the situation where the developed application would not become popular quickly enough to earn back the initial costs. Within the cloud, both situations are covered as you only pay for what you use. The cloud is highly scalable in the sense that resources can easily (and even automatically) be up- or downscaled with changing demands.

- *Easy access.* Cloud computing services are typically available through the web, worldwide. Cloud providers easily make it possible to store data on a number of different data centers all around the world. This ensures the data is always close the customer, improving performance.

- *Reducing risk and maintenance expenses.* With the traditional IT hardware setup, companies needed to maintain their own hardware. This requires more people and to reduce risk, a redundant network setup was required resulting in increased hardware costs. Today, cloud providers take care of both redundancy and maintenance, allowing cloud users to worry only about their core business.

- *Take advantage of large operational resources.* Cloud capabilities often appear unlimited to the cloud user (Mell & Grance, 2011). Though of course not unlimited, capabilities are enormous, allowing cloud users to utilize huge amounts of processing power and storage, something deemed almost impossible before the existence of the cloud.

An overview of cloud computing issues and challenges can be found in (Ghanam, Ferreira, & Maurer, 2012). Summarizing, issues and challenges are:

- *Security & privacy*. Most issues deal with the data that organization store in the cloud. Can they upload their privacy-sensitive data to the cloud? How secure is the connection? Who is responsible when data is stolen?

- *Infrastructure*. Mainly aimed towards the cloud providers, issues arise such as how to allocate servers optimally, load balancing and traffic management. These issues are related to sustainability issues, as it is important to minimize the amount of energy required. In addition these issues are also of influence to availability, reliability and scalability.

- *Data management*. Also aimed mainly towards cloud providers, issues related to data are data segmentation and recovery, fragmentation and duplication, retrieval, processing, provenance, anonymization and finally, data placement.

- *Interoperability*. This challenge is related to the ability to deploy software applications on different cloud providers. Together with portability, this issue is thoroughly described in the next section.

- *Legal issues*. Very much related to privacy issues, one main concern is the location of data. Where is the data actually stored, and can cloud providers ensure that data doesn't leave defined geographical constraints?

- *Economic challenges*. Included issues mainly deal with return of investment (ROI) of cloud computing adoption. How much time and money is actually spent on cloud adaptation? Is migration to the cloud cost-beneficial?

- *Service management*. The X-as-a-service paradigm has its own number of challenges. One issue is to automatically combine such services and the ability to deal with service outages and failures. There is a need for a workflow that encompasses the service lifecycle.

- *Quality*. The main concern within this category is the definition of service level agreements (SLAs), where cloud providers define concepts such as guaranteed service uptime. One issue is the tradeoff between complicatedness and expressiveness. Another issue is the lack of standardization, making it hard to compare SLAs between providers. Finally, a last issue in the quality category is that of quality of user experience. For high-demand applications such as video streaming and gaming, the user experience can suffer when there is a lack of bandwidth.

- *Software*. Issues related to software are those of software migration to the cloud and combining agile process with cloud adoption. Also related is standardization, as currently software may have to be written twice for different cloud providers. The communication between and coordination with cloud providers is another issue during every stage of the software development process (i.e. requirement, design, implementation, testing).

- *Trust*. Trust is considered a major obstacle of cloud adoption. Issues are related to trusting that data will not vanish, for example in the event of hardware failures or when a cloud provider is bought or goes bankrupt. Another trust issue is accurate resource usage and therefore fair and honest pricing.

3.2 Cloud Portability and Interoperability: Issues & Challenges

The heterogeneous characteristics of the cloud environment make it hard for applications to be deployed on different environments. Differences can exist between supported programming languages, libraries, services, and others. Due to a lack of standardization, cloud providers bring new, proprietary interfaces to the cloud landscape (Petcu, Macariu, Panica, & Crăciun, 2013). Seamless switching between providers is therefore impossible. The cloud ecosystem is fragmented and the wish to easily deploy applications on different cloud providers will need to be tackled through other means. This section describes to key aspects related to multi-cloud deployment.

Box 3.2 provides a concrete, real-life example that might occur when moving an application from Microsoft Azure to Amazon AWS. The example shows how it may be a challenge to move an application to another cloud provider. In other words: it is a challenge to make an application *portable*. Portability is a software quality attribute that has formally been defined by IEEE as "t*he ease with which a system or component can be transferred from one hardware or software environment to another*" (Standard Coordinating Commitee IEEE, 1990). Thus, software code can be considered portable when it can "*be operated easily and well on computer configurations other than its current one*" (Boehm, Brown, & Lipow, 1976). Within the context of cloud computing, the ability to run an application on a number of different cloud computing platforms is therefore a portability characteristic.

Before code can be run on a cloud platform, it needs to be moved to that cloud platform. This means the cloud provider needs to be accessed and after authentication and authorization, the entire application needs to be provisioned. This issue of cloud access is the most often occurring subject within research related to cloud security (Iankoulova & Daneva, 2012). Cloud security is the main barrier for cloud adoption and the most-written about subject related to cloud issues and challenges (Ghanam et al., 2012; Iankoulova & Daneva, 2012; Kuyoro, Ibikunle, & Awodele, 2011). Access control security issues are related to recognizing "*parties that want to interact with the system, making sure that the parties are who they say they are and giving them access only to the resources they are allowed to access*" (Iankoulova & Daneva, 2012). Overall, not just the internals of the software need to be adapted; the process and tools used for provisioning software is also required to be portable.

Closely related to cloud portability challenges is the matter of cloud interoperability. IEEE defines interoperability as "*the ability of two or more systems or components to exchange information and to use the information that has been exchanged*" (Standard Coordinating Commitee IEEE, 1990). Therefore, within the context of cloud computing, interoperability refers to "*both the link amongst different clouds and the connection between a cloud and an organization's local systems*" (Dillon, Wu, & Chang, 2010). Properly implemented interoperability is key for letting clouds work together or inter-operate. Again using the example of Box 3.2, the described situation is also an interoperability challenge as after moving the application to Amazon, it was no longer possible to communicate with the Azure service bus component.

Example of differences in service bus APIs

A *service bus* is a general software concept that allows communication between different (de-coupled) components. One or more components can send messages to the service bus and one or more other components can 'listen' to the service bus, thereby receiving messages. Both Azure[2] and Amazon AWS[3] have developed their own service bus (respectively named *Azure Service Bus*[4] and *Amazon Simple Query Service (SQS)*[5]) and both are accessed through a different API that exposes a different interface.

Imagine an application running on Azure that uses the *Azure Service Bus, which* we just have migrated to Amazon AWS. The now deployed application on Amazon AWS cannot communicate anymore with the Azure Service Bus, for example because of security settings, or because the implementation assumes a locally available Azure Service Bus. The first challenge is to rewrite the software in order to support the Amazon API. Changing each line of code that uses the Azure API is now required, a daunting and error-prone task.

A second, less obvious task that needs to be investigated is the difference of *queue name restrictions* in both environments. A queue is a required entity within a service bus that needs to be created, and it is with this entity that is used for sending and receiving messages. An interesting difference is that a queue name in Azure must be between 3 and 63 characters long; in Amazon this is between 1 and 80 characters. Another difference is that Amazon allows underscores (_) where Azure does not. Besides the *technical* changes required in order to start using the Amazon SQS, other *functional* changes may thus also be required, something that will need to be investigated before migrating from Azure to Amazon.

Box 3.2: Concrete example of software implementation differences between Microsoft Azure and Amazon AWS

Portability and interoperability are sometimes used interchangeably (Armbrust et al., 2010; Dillon et al., 2010; Lewis, 2012), whereas others do make the proper distinction (Petcu, 2011). In this thesis whenever the term 'portability' is used, 'interoperability' is meant as well as in many cases communication between components (interoperability) will need to be made portable, and can therefore be seen as a subset of portability challenges.

The lack of portability is a main barrier for cloud computing adoption as it may lead to vendor lock-in (Lewis, 2012; Petcu, 2011). Moving software to another cloud platform is costly, as software needs to be rewritten. IEEE defines a number of use cases related to the portability and interoperability of software

[2] http://azure.microsoft.com/
[3] http://aws.amazon.com/
[4] http://azure.microsoft.com/en-us/services/service-bus/
[5] http://aws.amazon.com/sqs/

applications within the cloud (Liu et al., 2011). First, *data portability* is the ability for a cloud user to move their data to another cloud provider. Second, *service interoperability* is the ability to use data and services across different cloud providers. Third and last is *system portability*, which is the ability to move an entire application to another cloud provider. Different use cases are important for different cloud service models. For example, data portability is especially of interest to SaaS cloud users (e.g. moving Gmail e-mails and contacts to an Outlook account). Service interoperability is especially of use to PaaS cloud users, as components offering features such as databases, service buses and file storage are provided as services, and it may be beneficial to use a service from another provider (see Box 3.3).

Portability and interoperability explained

Imagine an application that uses three services: a service bus, a MySQL database and file storage. The application is running on Azure and uses the Azure-specific services (see Figure 3.2, the left part). In the event the application is moved to Amazon AWS, it is preferred to use the same services but now those as provided by Amazon (e.g. the ‘Amazon Simple Query Service’ instead of the ‘Azure Service Bus’). If this is possible without any changes to the code, the application is *portable*. Would the application running on Amazon, still continue to use the Azure-hosted MySQL database, this is an *interoperable* characteristic. Code running on Amazon uses data stored on Azure; the providers thus interoperate. In Figure 3.2, horizontal and diagonal lines are examples of interoperability; the vertical lines are portability.

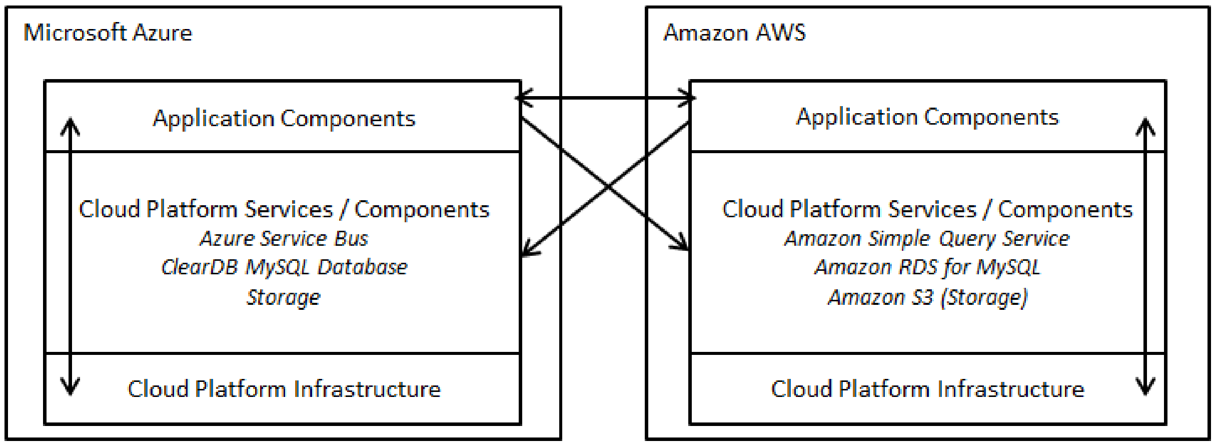

Microsoft Azure
Application Components
Cloud Platform Services / Components
Azure Service Bus
ClearDB MySQL Database
Storage
Cloud Platform Infrastructure
Amazon AWS
Application Components
Cloud Platform Services / Components
Amazon Simple Query Service
Amazon RDS for MySQL
Amazon S3 (Storage)
Cloud Platform Infrastructure

Figure 3.2: Showing interoperability (horizontal and diagonal lines) and portability (vertical lines)

As the figure shows, application components can also connect with each other (the horizontal line). For example, the Amazon AWS instance may be running on a machine capable of performing computational-heavy calculations, and the Azure instance makes use of these capabilities through a web-accessible interface (web-service).

Box 3.3: Concrete example showing the difference between portability and interoperability

Looking back at the challenges defined in this section, we identify three main categories that are of interest which are summarized in Table 3.1. The second category, encompassing cloud access issues related to authentication and authorization is coined ‘meta-portability’. We consider these issues higher-level than the portability issues as they are required before (code) portability can be achieved. Code cannot be deployed on different platforms before those platforms can be accessed. Thus, in order to enable portability, another set of issues is rudimentary required to be solved.

Challenge	Description
Portability	Porting code to another provider whilst using similar services from that provider.
Meta-Portability	Gaining cloud access through authentication and authorization and initializing other requirements for cloud access and use[6].
Interoperability	Accessing a service from wherever; either code running within the same provider or with code running on another provider.

Table 3.1: Main categories of multi-cloud challenges

3.3 Multi-Cloud

As has been shown, vendor lock-in is a main issue towards cloud adoption because of the portability issues. Cloud users desire the flexibility of seamless switching between cloud providers. This allows them to adapt to changes in cloud providers, e.g. in pricing, policy, or availability. In addition, a multi-cloud setup is considered a solution to an important cloud adoption inhibitor; the unavailability of services (Armbrust et al., 2010). Essentially, cloud users are eager for a *multi-cloud* adoption strategy, where they can benefit from unique characteristics available from different cloud providers.

A number of use cases can be defined in which a cloud user (simultaneously) uses multiple cloud providers. The simplest example is where an entire system is migrated to another cloud provider. This may be of interest, for example, when a cloud provider goes out of business, changes its (privacy) terms or because another cloud provider offers better options (Petcu, 2011). Another commonly written about example is the case of *cloud bursting*. We talk about cloud bursting when an external cloud provider is used for scalability for certain time intervals or specific circumstances, such as a high amount of users trying to access the service (Nair et al., 2010). This is usually performed in a public/private cloud combination setup, where the private cloud has limited capabilities and instead of acquiring more hardware that may be costly, a public cloud is used when needed. Last, multiple cloud providers can be used to provide redundancy. Would issues arise at one cloud provider, everything that is required to run the application already exists in the same state at another cloud provider. Traffic is now forwarded to only the cloud provider that experiences no problems, until the other cloud provider is again running normally.

A few similar definitions are closely related to the definition of multi-cloud computing. The first of these a federated cloud. The difference here is that a federated cloud is by definition a voluntary connection

[6] As we will see in Chapter 6, different requirements exist for using an automated cloud solution.

between cloud providers (Grozev & Buyya, 2012). The cloud providers therefore chose to connect their clouds and cooperate through an agreement. Within a multi-cloud setup, this is not the case per se. Although the providers may be aware of and may even support the interconnection, the initiative comes from a cloud user. This user therefore is also responsible for provisioning and management of the multi-cloud setup (Grozev & Buyya, 2012). The encompassing definition of both multi-cloud and a cloud federation is *inter-cloud*. This is the generic definition for cloud-connected setups, and both a federated cloud and a multi-cloud can be considered a type of inter-cloud. Last, when the interconnection occurs on the same delivery model level (SaaS, PaaS, and IaaS) this is called a *horizontal* connection. Would a connection for example be made between an IaaS and PaaS provider, a *vertical* connection is created (Toosi et al., 2014).

Many other definitions have been proposed that depict the various forms of using multiple clouds such as cloud-of-clouds, sky computing, aggregated clouds, multi-tier clouds, and more. These definitions are not significantly different from the already-described definitions above (Petcu, 2013b) and will thus not be described further. For further information we point to reader to one of the attempts to taxonomizing the many existing multi-cloud definitions (Grozev & Buyya, 2012; Moscato et al., 2011; Toosi et al., 2014; Vaquero & Rodero-Merino, 2008). We consider the research in this thesis part of the multi-cloud paradigm.

Within the multi-cloud paradigm a distinction is made between services and libraries (Grozev & Buyya, 2012). The main difference is that a service is typically externally hosted and is often a 'black box' in the perspective of the user. The service is a kind of middleware that is used by the user to access multiple cloud providers. A library is hosted and often also developed and maintained in-house. This library is part of the software system that connects to multiple clouds instead of being external to the system. One big advantage of a library versus a service is that when using a library, all authentication information stays in-house whereas with a service, authentication information will need to be transferred to the external service (Thatmann, Slawik, Zickau, & Axel, 2012).

In general, two methods exist to facilitate the connection of multiple clouds, namely standardization and brokering (Toosi et al., 2014). First, standardization attempts to connect clouds by defining general interfaces that can be used to access a cloud. Standardization can occur on a number of levels, including network architecture, data format, metering and billing, quality of service, resource provision and security and identity management (Rong, Nguyen, & Jaatun, 2013). To give a concrete example, within an IaaS setup standards can be used for virtual machines (Andrikopoulos et al., 2012) so that they can easily be migrated to another provider. On the PaaS level standards are required for services, e.g. a service bus. Box 3.2 in the previous section showed the example of two different service buses, one provided by Microsoft Azure and the other by Amazon AWS. Even though both provided very similar functionalities, the interfaces to connect with and utilize these services differ so that software has to be rewritten for it to use another service bus. This wouldn't have been the case of both providers had used the same standard to work with their service bus.

Many standardization initiatives exist, a list of which is compiled by the Cloud Standards Customer Council, an end user advocacy group that is dedicated to "*accelerate cloud's successful adoption and drill down into standards, security and interoperability issues*" (OMG, 2011). Their Wiki currently shows a list of fourteen cloud standard initiatives (Cloud Standards Customer Council, 2014); the most well-known initiatives are summarized in Box 3.4. The list contains both overlapping and non-overlapping standards on all levels and unfortunately, these standardization attempts are conducted in isolation. The large amount of standards is considered a barrier to cloud providers adopting a single standard (Fogarty, 2011; Lewis, 2012). On the one hand, cloud providers are not interested in the implementation of standards as they believe the current vendor lock-in is beneficial for them (Machado, Hausheer, & Burkhard, 2009; Petcu, 2011). However, others feel they may benefit from adhering to standard and thus allowing easier migration or multi-cloud setups as it will attract more customers to the cloud (Marston et al., 2011). Another barrier for standardization is the fact that each cloud provider offers differentiated services (e.g. to offer (seemingly) unique functionalities to their users) which are hard to fit in a single standard (Petcu, 2011). To quote Joe Skorupa, vice president of Gartner:

> *"Even if an open cloud standard should come to pass, every provider would still continue to implement its own proprietary enhancements to differentiate its wares from the competition. […] Vendors do not want clouds to become commodity products because they do not want to compete on price alone."* (Claybrook, 2011)

Overall, standardization initiatives are largely focused on the IaaS layer (Cunha, Neves, & Sousa, 2014). One such exception is CAMP, an initiative started in August 2012 by large players such as Oracle, Rackspace, Redhat, Cloudbees and Huawei (Carlson et al., 2012). CAMP has been accepted as an OASIS standard, however to date no concrete, practical results have yet been achieved. None of the relevant vendors have yet adopted the standard. At this point in time it does not appear that the larger cloud providers will start implementing a single standard anytime soon.

The lack of standardization led to the development of another approach towards bringing clouds together, which is through a brokering mechanism (Tordsson et al., 2012). In a traditional sense, a broker is an independent party (e.g. a person or an organization) that brings together sellers and buyers (Spiro, Stanton, & Rich., 2003). An example is a real estate agent. Through a website (service), buyers and sellers of homes can find each other in one single location. In addition the real estate agent can also provide tours of houses (another service), further lowering the amount of time required by both the buyers and the sellers to get a deal together. Making use of the services offered by a broker is therefore typically done to save time and therefore, money.

Within the context of the cloud environment, a cloud broker service (CSB) is "*an entity that manages the use, performance and delivery of cloud services, and negotiates relationships between Cloud Providers and Cloud Consumers*" (Liu et al., 2011). A report by Gartner defines the following three roles of a CSB (Sampson, 2012);

- *Aggregation.* The bundling of different services allows for providing a unified service to the cloud broker user. A billing service, for example, can send one instead of several invoices to the user.
- *Integration.* The aggregation of services allows for (technical) integration between different cloud providers. Through one seamless interface, the cloud broker user can migrate data between providers or integrate software with multiple cloud providers.
- *Customization.* A CSB can provide a number of options that span multiple clouds. For example through aggregation of the data center locations, it can provide options for where to store the data. The cloud broker user does not need to know which cloud provider is actually used.

In essence, a cloud brokering mechanism is an added layer – a *middleware* - between a cloud user and cloud provider that takes care of the communication between the cloud user and (multiple) cloud provider(s). Thus instead of talking directly with a cloud provider, the cloud user talks with the CSB, who forwards the call to a cloud provider and returns the answer, possibly altering the answer to a common format as well. These 'calls' can for example be the use of a service (e.g. querying a database) or the provisioning of a virtual machine. Box 3.5 shows an example of an application using a CSB.

Cloud standardization initiatives

A number of cloud initiatives exist, ranging from standardization of cloud access management, virtual machines and defining standardizes interfaces to services such as authentication, files, queues, hash tables and tasks (Lewis, 2012). The most well-known initiatives are shortly presented here.

- *OVF (Open Virtualization Format)*[7]. A standard that describes a pre-configured virtual machine image that can be deployed across heterogeneous platforms. An OVF "package" usually contains multiple images. A descriptive meta-data file contains properties such as the name and hardware requirements of each image. Depending on the underlying architecture, the correct image can be chosen.
- *CDMI (Cloud Data Management Interface)*[8]. As the name suggests, this standard defines the interface for creating, retrieving and updating "cloud storage elements". Cloud storage can both be a database and the storage of files on the file system.
- *CIMI (Cloud Infrastructure Management Interface)*[9]. This standard defines the logical model for the management of resources within the IaaS domain. Resources such as machines, volumes, networks and monitoring are defined.
- *TOSCA (Topology and Orchestration Specification for Cloud Applications)*. TOSCA is a modeling standard that describes how to define the components and their relations of an application as a topology graph. Through management plans, these components can be deployed, configured and managed on an actual cloud provider. TOSCA is thoroughly explained further in section 3.4.

[7] http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf
[8] http://www.snia.org/cdmi
[9] http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.1.0.pdf

- *Cloud Manifesto.* This standard isn't known primarily for its actual specification, but is still very important because it's one of the first initiatives towards a cloud standard. Though initiated by none other than IBM, the manifesto became controversial after Microsoft publicly spoke out against it, being disappointed by the "*lack of openness*" (Martin, 2009). Eventually, the initiative got "*laughed out of the market*" (Fogarty, 2011). Though failed, it did show how *not* to attempt a cloud standardization initiative.

Box 3.4: Examples of the most important cloud standardization initiatives

In addition to deliver service interface abstractions for implementation purposes, a CSB can also provide a uniform management interface (Tordsson et al., 2012). This interface can be used for deployment on multiple cloud platforms, for example to initiate, pause, resume, and terminate virtual machines or for monitoring purposes. Through aggregation the user can see data that is of interest from different cloud providers within a single interface and easily compare or customize the multi-cloud setup.

Using a Cloud Service Broker (CSB)

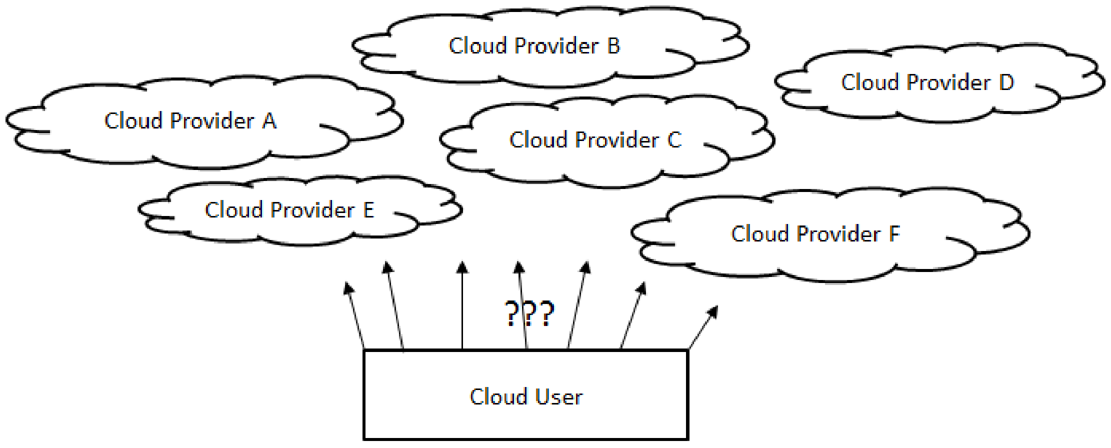
Imagine an application that wants to seamlessly make use of both Microsoft Azure and Amazon AWS. A selected CSB – named CSBX – supports both these platforms. Looking back at the previous section, the three identified challenges are tackled by using CSBX;

- *Meta-portability*. The application developer already registered for two accounts on both Microsoft Azure and Amazon AWS. The application forwards both credentials (both entirely different formats) to CSBX on initial connection (this should only be done using a secure connection). The broker can now communicate with the accounts of both cloud providers, and the application can now use CSBX to transfer its software code to any of the cloud providers.
- *Interoperability*. The application uses a *service bus*, a feature offered by both cloud providers but with notable differences (see Box 3.2). Instead of requiring an implementation for both providers, the application communicates with the general *service bus* API provided by CSBX. Depending on the configuration, CSBX forwards this call to one or perhaps even both cloud providers. In addition, any answers are returned to the application in a general, specified format. The application can now seamlessly communicate (*interoperate*) with both services; it doesn't matter if the code and service bus operate within the same cloud provider.
- *Portability.* Even though we are now using a CSB, portability is still the biggest challenge as it requires writing code that adheres to what CSBX can actually work with. For example, instead of writing files directly to the system, a CSBX 'file storage' API should be used that encapsulates any differences of writing to the file system. When doing this correctly and in the situation where the CSB moves the code to another provider for whatever reason, the code will continue working as the provider-specific functionalities are now utilized. As explained earlier, portability and interoperability are closely related: would CSBX forward any API calls to another cloud provider, this is both an interoperability and portability characteristic. One problem that is found here: instead of adhering to a cloud provider API, the application now

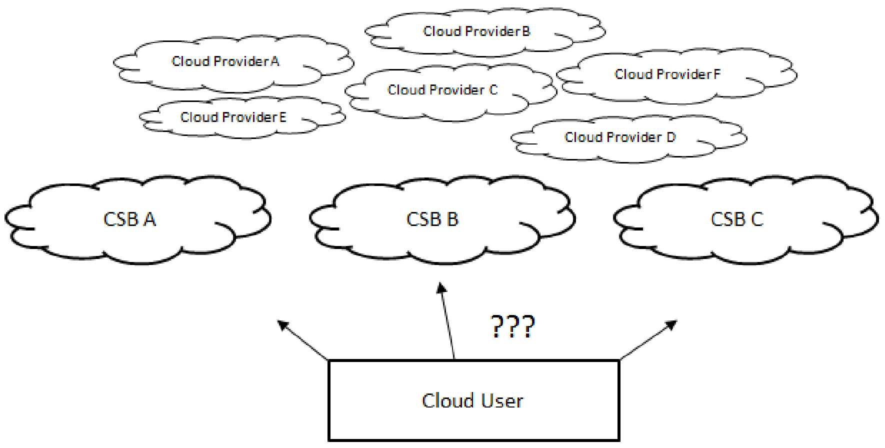
has to adhere to the CSBX API, creating a strong dependency on a different level.

Box 3.5: An example showing a practical use of a cloud service broker

The CSB sector is rapidly growing; Gartner expects $100 Billion of revenue by the end of 2014 (Bova & Petri, 2013). CSBs have filled the gap of how to handle a multi-cloud setup, including smooth cloud adoption, management, migration and maintenance (Wadhwa, Jaitly, & Suri, 2014). This market however is becoming fragmented, similar to how the cloud environment is already fragmented. Whereas before it was a challenge to select a cloud *provider*, it now becomes a challenge to select a cloud *broker*. The problem has therefore shifted to another level, although essentially is still the same. In fact, a 'CSB registry' has already been proposed that should facilitate in the selection of such a broker (Wadhwa, Jaitly, Hasija, & Suri, 2015). Figure 3.3 shows how these situations are very much alike.



a) the former situation where a cloud user has to choose between different cloud providers



b) the new situation where a cloud user can choose between a number of CSBs

Figure 3.3: The increasing number of cloud service brokers (CSB) creates a similar situation to before the existence of cloud service brokers

Figure 3.4 and Figure 3.5 graphically explain respectively standardization and brokering. Both figures show an application including two interfaces. For example, the green interface may be a database whereas the blue interface is a service bus. The former figure shows how the puzzle pieces fit with each cloud provider – they are standardized, and thus offer the same interfaces. The latter figure shows how each cloud provider exposes a different interface for a similar service. Thus, a cloud broker is used that can communicate with each provider, and handles requests received from the application. In this case the broker is outside the application and can thus be categorized as a *service*.

Both standardization and brokering take a technical perspective towards a multi-cloud setup and typically do not support business-oriented decisions. Before a connection between clouds is made the decision first has to be made *which* clouds to connect with. The discovery, selection and configuration of a multi-cloud setup is a whole research area on its own (Sun et al., 2014). Because of the lack of standardization within the cloud environment, selecting a (range of) cloud provider(s) can be a challenge because of the differences in names, characteristics and functionalities of similar services. A great analogy is sketched in (Silva, Rose, & Calinescu, 2014). Both Starbucks and Caffè Nero sell coffee in three common sizes: small, medium, and large. However, both use different names for these sizes. For example a *Grande* sold by Starbucks can be considered a 'medium' coffee, whereas at Caffè Nero one would receive a 'large' coffee when ordering a *Grande* coffee. Besides differences in naming, a Starbucks 'medium' latte contains 156% more calories compared with the Nero version. Thus, besides the technical portability and interoperability challenges as described in the previous section, these *functional* differences as well hinder the smooth usage of multiple clouds (Sheth & Ranabahu, 2010).

A method to address this issue is through the use of a domain specific language (DSL), also known as an application-oriented, special purpose, specialized or task-specific language (Mernik, Heering, & Sloane, 2005), and is suited for a specific domain or context (Brambilla, Cabot, & Wimmer, 2012). The cornerstone of a DSL is the meta-model, which consists of an abstract (construct) and concrete syntax (representation of construct). The meta-model essentially is an abstraction of a specific domain (in our situation, the 'cloud environment' domain), which maps concrete instances of types to more general concepts that can subsequently be used to improve the understanding of communication about the domain. Following the metaphor of comparing apples with oranges: a DSL tells you which are the apples and which are the oranges, so that you know which you can actually compare, and how.

The previous section described the two instances of the Azure and Amazon 'service bus' (a construct), respectively named 'Azure Service Bus' and 'Amazon Simple Query Service' – two representations of that construct. When using a broker, its management user interface would probably group the service buses from each provider within a single page or section. This aggregation requires a common language and a mapping from the concrete instances to the common construct. The broker's maintainer would need to map the construct representations to the construct once. Then, both the user and automated methods and scripts can use this knowledge to make decisions.

Besides specifying specific entities within the specific domain, relations between those entities are also modeled. For example, a cloud service is 'provided by' a cloud provider. A cloud service can thus not exist without a provider. Another example is the requirement of specific properties; e.g. a cloud service may require a connection URL through which the service can be accessed. These 'rules' are as well stored in the meta-model in a machine-readable form and can thus be automatically processed, for example to validate the modeling of a specific cloud provider including its services. This is an important step towards cloud automation and should aid in the selection, configuration and deployment of applications within a multi-cloud setup.

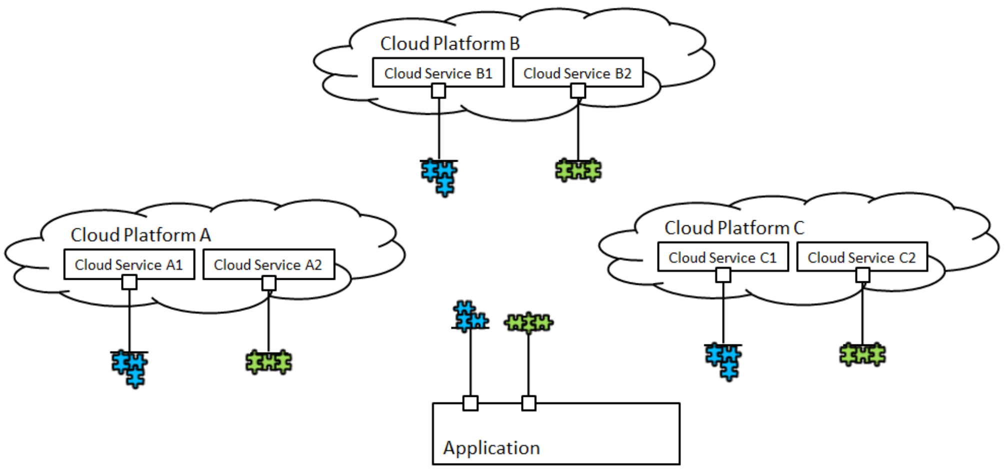


Figure 3.4: Seamless communication between clouds through standardization: each cloud has the same interface and the application can thus communicate with each of them

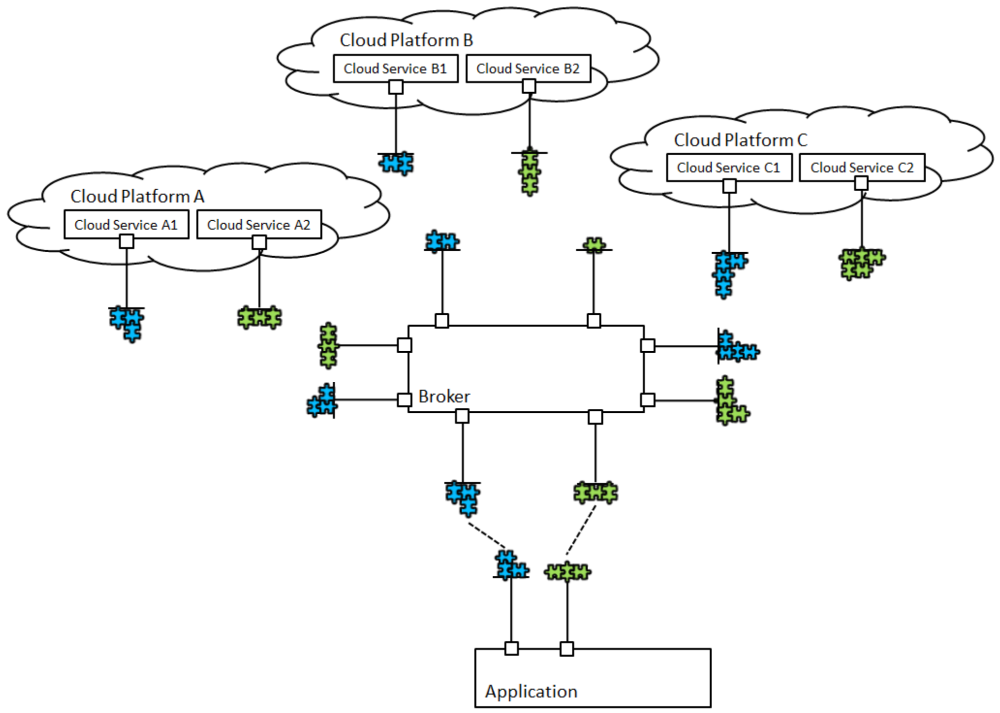


Figure 3.5: Seamless communication between clouds through brokering: instead of directly communicating with each cloud platform, a broker is used as middleware that knows the protocols for each specific cloud provider

3.3.1 Towards Multi-Platform Deployment

Up till now this section has only considered deployment scenarios where every component is deployed within the cloud, hence the term *multi-cloud*. In this sub-section we introduce the more general term *multi-platform* deployment, hereby defined as a multi-cloud deployment with the addition of *local* deployment, i.e. on a laptop, desktop computer or even a handheld such as a mobile phone. Multi-cloud is therefore a subset of multi-platform.

The bridge towards multi-platform deployment is theoretically relatively small. After all, the cloud environment consists out of computers essentially no different than those running within the offices. If we allow an application to be deployed on different cloud providers, meaning, different computing platforms, why not include local deployment as simply another ‘platform’? If a local computer is connected to the internet and is configured with proper access settings, a cloud orchestration tool can push application components to this laptop similar to how it pushes them to the cloud. When the orchestrator itself is running on the laptop, deploying components to that same laptop should especially be a breeze.

Some use cases for which we consider local deployment as added value are;

- *Demonstration*. Demonstrating the software, for example to a potential customer, may include not showing only the software itself, but also the deployment of that software. Multi-platform deployment may prove to be difficult when within an unknown network. At this point it may be beneficial to host the deployment software on the same laptop from which the software is deployed.
- *Migrating to the cloud*. As existing software will typically need to be rewritten for it to be deployed to the cloud, moving an entire application to the cloud may be a timely and expensive undertaking. Therefore a phased approach is of interest where individual components are increasingly deployed to the cloud, whereas the rest remains on a local system.
- *Keeping privacy-sensitive data local*. It is not uncommon that data needs to remain on local systems due to privacy regulations, or because organizations simply are not comfortable with storing their data in the cloud (Kumar, Garg, & Quan, 2012). Applications using local data may still (partially) be deployed within the cloud, during which the database components is to be deployed locally.

3.4 The Topology and Orchestration Specification for Cloud Applications

One of the subjects of interest within this research is TOSCA – the Topology and Orchestration Specification for Cloud Applications (Binz et al., 2014). TOSCA is a method that aids in the automation of deploying and managing (cloud) services. It is an OASIS (Organization for the Advancement of Structured Information Standards) standard since January 16, 2014. The standard can be used for describing components, their relations, and processes that manage them. Before diving any further into the standard, we first give some background into the development of TOSCA.

TOSCA was initially developed from the believe that automation in IT is a critical factor for coping with the increasing degree of complexity (Binz et al., 2014). With the advent of cloud computing, these complexities can be dealt with through outsourcing of IT maintenance and management. Entire applications can be hosted within the cloud where management and maintenance operations are automatically taken care of by the cloud provider. However, IT applications are increasingly developed as compositions of individual components, together aggregated into a single system providing synergetic functionality. Approaches for describing composite applications have been defined, however a meta-level orchestration method which manages those components was still missing (Andrikopoulos et al., 2012). Each component needs to be managed, configured and deployed individually, as well as making sure the components are able to communicate with each other (interoperability). These operations are often performed manually, hindering automation, repeatability and self-service (Binz et al., 2014). Achieving automation requires the modeling of the applications' components, their relations and management in a machine-readable format. This is exactly what TOSCA does.

Figure 3.6 presents an overview of the TOSCA architecture. On the right we can see node types and relationship types, both which have properties and node types which also have interfaces (executable scripts). The topology template on the left contains a number of nodes and relations. A node template inherits a node type, whereas a relationship template inherits a relationship type. An example of a node may be a MySQL server. Each MySQL server typically contains the same or similar properties. Even lifecycle operations for creation and destruction of such a node may be the same for different servers. Such information is therefore stored in the node type. Through inheritance, reusability is increased as node types typically only need to be modeled once, and can then easily be re-used.

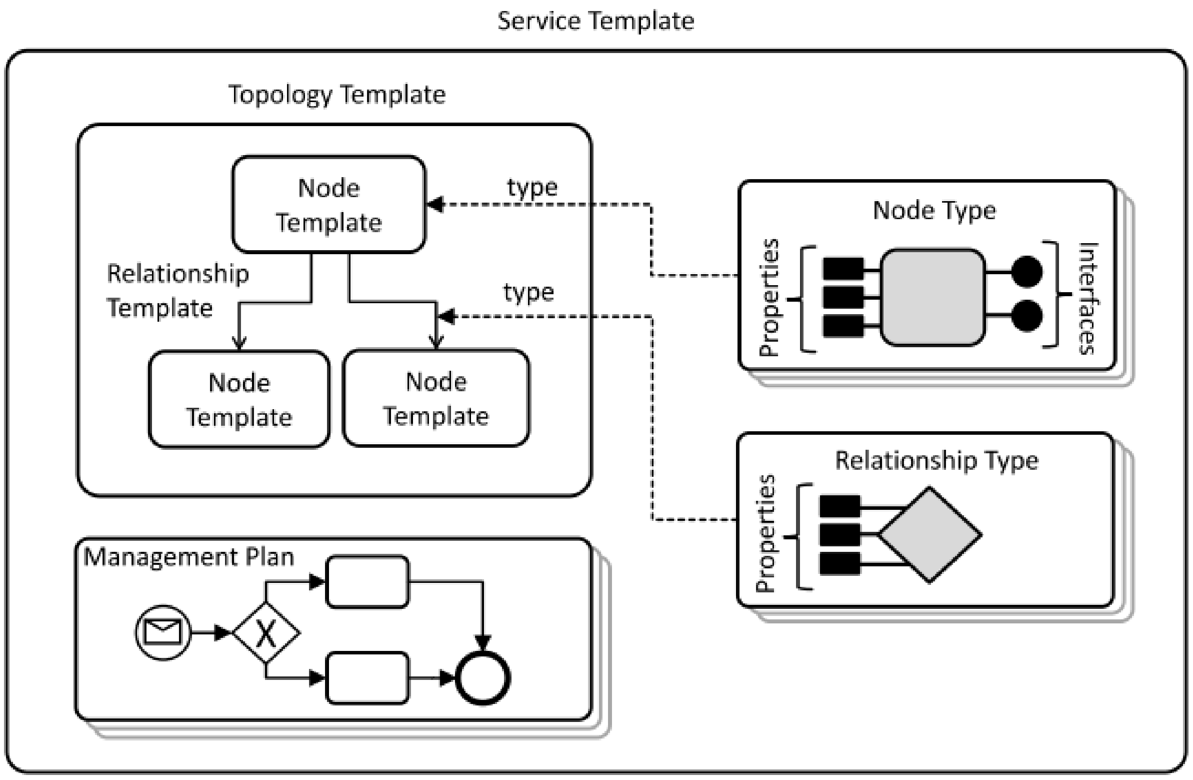


Figure 3.6: An overview of the TOSCA architecture. From (Binz et al., 2014).

The last entity displayed within Figure 3.6 is the management plan. These plans can manage multiple nodes within an entire application. Deploying an application will require a number of steps (such as deploying each node and executing scripts), which is defined in a management plan.

Different scripts are associated with each node. A node may contain scripts for deployment, re-deployment, starting a service, stopping, and destroying the entire node. Relationships are especially powerful. An example of such a relationship is *hosted-on*. A piece of software code is hosted on a PaaS or IaaS container. Both the software code and the container will be a node within TOSCA. Each of these will contain scripts for the proper installation. A relationship will then contain scripts is well, such as providing the software code script the correct credentials for gaining access to the container. This may be only known *run-time*, e.g. after the container has been successfully provisioned.

Important to note is that the TOSCA *specification* is different from a TOSCA *interpreter*. The TOSCA specification only specifies an XML or YAML file where information about types (node and relationship) and templates (application compositions) is described. Such a file will contain all information required to execute lifecycle operations on an application within a specific environment. Such an XML or YAML file will need to be parsed and interpreted however. Scripts need to be executed and node properties need to be injected into such scripts. In the next chapter, we will see an example of such an interpreter, Cloudify.

3.5 Feature models

The previous section described the TOSCA standard, which aids in the deployment and management of (multi-)cloud services. Before deploying the components of an application, one first needs to know *where* to deploy the components. TOSCA assumes this is already known; the management plans that are executed already contain the information of where to deploy each component. In this research we seek to automate the process of cloud selection as well. For this we turn to feature models.

A feature model (Kang, Cohen, Hess, Novak, & Peterson, 1990) is often used to model the variability existing within the software product line (SPL) branch. The term was coined in 1990 and has since been one of the main research topics within SPL (Benavides, Segura, & Ruiz-Cortés, 2010; Kang et al., 1990). An SPL is a software engineering and development paradigm that focuses on the re-use and optimization of a software product and its components (Clements & Northrop, 2002). It includes a set of similar products that share common code. An example is the Microsoft Office 'family' with application such as Microsoft Word, Excel and PowerPoint. SPL by itself is a promising paradigm that has been shown to increase the return on investment, shorten the time-to-market and improve the software quality (Thüm, Apel, Kästner, Schaefer, & Saake, 2014).

The two key concepts of feature models are *commonality* and *variability*. Commonality means that different products share common features. For example, each webshop shares some functionality such as a product list, contact methods, payment options and perhaps a search feature. The products within the Microsoft Office family all share features such as the ribbon menu, save & load functionalities and methods to review the contents. Variability, on the other hand, deals with the fact that each feature knows different kinds of concrete implementations. For example, the payment method for a webshop may be VISA, MasterCard or Bitcoins. Contact methods may be a webform or a phone number. A number of different design stylesheets may be available, with the primary color being red, green or blue. A search feature may or may not exist. Feature models are suited to create a graphical, easy-to-read overview of the *configuration options* (the valid combinations of features (Benavides et al., 2010)) and it shows the hierarchy between those features. Figure 3.7 shows an example of a feature model of a webshop.

One may have noticed that the design stylesheet example is not a feature. This is correct, and it shows that feature models are not limited to features but can encompass many things that include some sort of variability. If a pre-defined set of options exist, they are likely to be added to the feature model. We consider a feature to be a "*prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*" (Kang et al., 1990). Feature models can impose constraints on the possible configurations through the different types of relationships between the features, of which the following types exist (Quinton, Haderer, Rouvoy, & Duchien, 2013a);

- *Mandatory*. This feature is required. Each webshop has a list of products and a payment method.
- *Optional*. This feature is not required. A webshop may or may not have search functionalities.
- *Or*. At least one of the features must be selected. A webshop must have one or more payment methods. Only VISA payments and all three options are both valid.
- *Alternative*. Exactly one of the features must be selected. Only one design stylesheet can be selected.

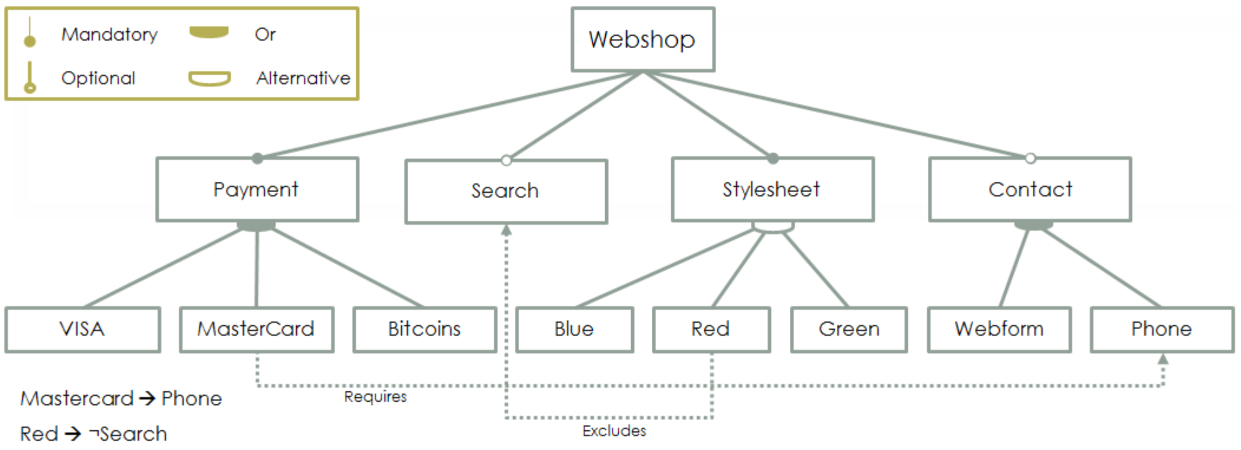


Figure 3.7: An example of a feature model. The legend shows the types of relations (mandatory, optional, alternative, or). Dependencies between nodes are shown as arrows (requires / excludes) and through logical formula's.

In addition, dependencies between features may be added. Figure 3.7 shows that in case MasterCard is selected as a payment method, a phone number must be listed on the website. Also, in case the selected stylesheet is 'red', no search functionality may be available (the rationale for this is not included in the model, and is in this example arbitrary). The combination of the possible types of relationships and dependencies creates a powerful method for the modeling of domains that have a high amount of variability.

A second type of extension to the feature model is the *attribute*. One paper describes feature models do not allow the modeling of quality attributes (Benavides, Trinidad, & Ruiz-cort, 2005). These quality attributes (also known as non-functional requirements) differ from functional attributes as they describe "*how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise*" (Stellman & Greene, 2005). Coined *extended* feature models (EFM) (Benavides et al., 2005), these data structures allow for an even larger amount of possibilities with the same amount of items. The attributes allows including more information about such an item. Other names sometimes used in literature are *advanced* FMs or *attributed* FMs (Benavides et al., 2010).

One prerequisite of a quality attribute that is to be included in an EFM is that this attribute is measurable. Otherwise it would be impossible to make comparisons based on these attributes. Relations between attributes are also possible. For example, a *price* attribute of an item can be the sum of the prices of its sub-items. Complex constraints such as "IF price of feature A is lower than X, THEN feature B is excluded" are also allowed (Benavides et al., 2010, 2005). Figure 3.8 shows a subset of Figure 3.7 with the addition of some example attributes.

As was mentioned in the beginning of this section, we desire a method that automatically selects a deployment scenario based on the possibilities modeled within the feature model. The field of *Automated Analysis of Feature Models* (AAFM) defines a catalog of operations that can be used to extract information from feature models (Garcıa-Galán et al., 2013). Feature models can be analyzed through different logic paradigms such as propositional logic, constraint programming and description logic (Benavides et al., 2010). For example, within the paradigm of constraint programming, constraint satisfaction problems have been used to model EFMs so that a possible configuration can be found, amongst other operations (Benavides et al., 2005). The operations that are of interest to this research are the following (Benavides et al., 2010):

- *Void feature model*. This operation determines whether a FM has at least one valid configuration. This is useful for debugging a FM before actual usage. Figure 3.9 (a) shows an example of a FM with no solution.
- *Anomalies detection*. Also of interest for debugging, this operation checks for certain anomalies within FMs, such as dead features (features that can never be included) or redundancies (e.g. dependencies that are already implied through other dependencies). Figure 3.9 (b) shows a FM with a redundant dependency.
- *Filter*. This operation takes a set of constraints (a configuration) and returns a set of possible FM configurations. For example, looking at Figure 3.7, we may want to know which possible configurations exist that have at least the VISA payment option and a red stylesheet.
- *Optimization*. This operation takes an "objective function" and returns the optimal configuration given that function. This is especially useful when using EFMs with attributes. For example, we

may inject the function that minimizes cost into the FM depicted in Figure 3.8, which would return a configuration with only the ‘MasterCard’ payment option selected.

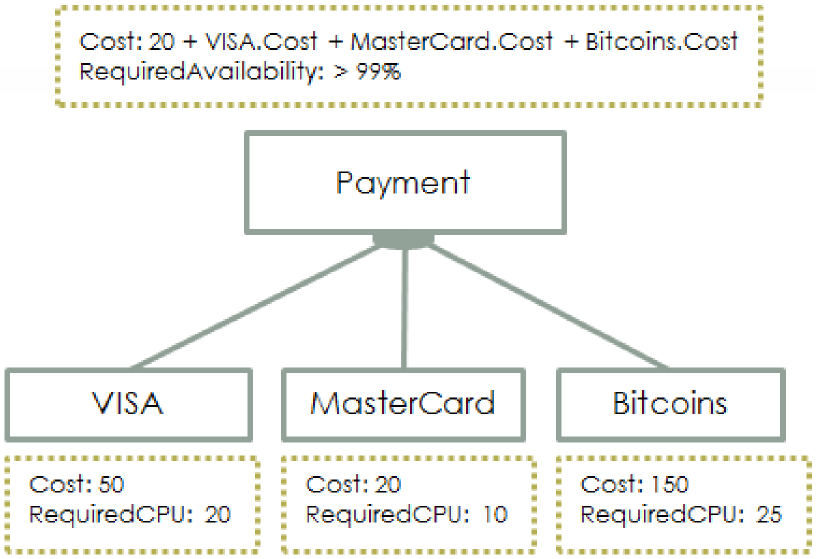
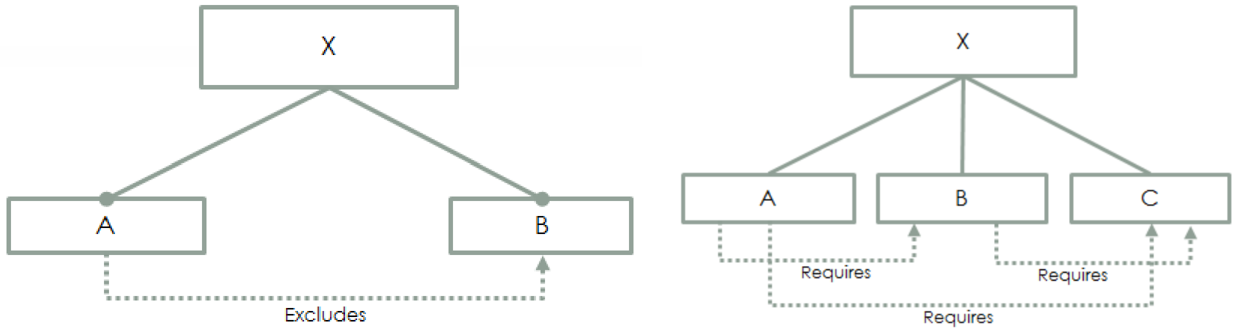


Figure 3.8: An example of an extended feature model that includes attributes



a) both A and B are mandatory, however A excludes B, and thus no valid configuration exists

b) because A requires B and B requires C, it is redundant to add the dependency A requires C

Figure 3.9: Two examples of FMs, a) has no valid configuration and b) has a redundant dependency

All examples shown in this section were small. Indeed, for each example it was easy to see which configurations were possible, what kind of configuration was the ‘cheapest’ or which error or redundancy exists in a FM. A FM however can become enormous, at which point it is practically impossible to make any sense out of them. When a FM exists out of dozens or even hundreds of features with all sorts of dependencies and attributes, it’s not feasible anymore to manually search for inconsistencies or to find an optimal or even a valid configuration. The graphical display of a feature model at this point is also questionable, and shouldn’t be of much use besides showing the complexity of the modeled domain. When dealing with large and complex feature models, using an AAFM is therefore the only method to extract any useful information out of them.

The large number of possible configurations in feature models

With each added item to the feature model, it grows exponentially. However, by default, we humans appear to be unable to grasp the idea of exponential growth. One famous quote related to this understanding is "*The greatest shortcoming of the human race is our inability to understand the exponential function*" (Bartlett, 2012). To fully understand the power and practical use of (extended) feature models, it is therefore of interest to dive a little deeper in how they can comprise such a large amount of options. For this purpose, let us look back at Figure 3.7 and determine how many options exist within this feature model. For simplification, the dependencies between features (the arrows) are ignored. First, let's see how many options each sub-item has;

- *Payment.* At least one option has to be selected, or perhaps all options, and anything in-between. This accounts for a total of 2^3 - 1 = 7 options. (The "minus 1" is added because "no options" is not possible).
- *Search.* With no sub-items, this feature is either included or is not. Thus, two options.
- *Stylesheet.* Exactly one has to be selected, and three sub-items exist. Thus, three options.
- *Contact.* Similar to payment, but now with only two sub-items and zero options is also possible. Thus, 2^2 = 4 options.

Next, following basic probability calculations, the number of allowed configurations for the entire feature model is 7 * 2 * 3 * 4 = 168. This is a large amount for such a small tree!

Finally, let us add another item to the feature model on the level of payment, search, stylesheet, and contact. Let us assume this item is similar to *payment*; it has three 'alternative' sub-items, and therefore seven options. The entire feature model will now have 168 * 7 = 1176 possible configurations. A *huge* increase and a concrete example of the enormous growth with adding more features.

Box 3.6: An example that shows how feature models grow exponentially with each added item

3.6 Observations

In this chapter, we gave a general overview of the cloud and multi-cloud landscape. Here we quickly summarize the most important observations.

- Cloud vendors are very much heterogeneous, which means that writing software especially for one such vendors can easily create a vendor lock-in. This makes it hard and therefore costly to (partially) switch to another cloud provider.

- Three technical challenges related to a multi-cloud setup are meta-portability, portability, and interoperability.

- Though different standardization initiatives exists within the cloud environment, no consensus has yet been reached. In addition, enough evidence exists that cloud providers are not interested in a single standard. A different solution is therefore of interest in the form of a cloud service broker (CSB).

- The rise of CSBs leads to a choice problem on a different layer. Whereas before, a user had to choose between different cloud providers, now a choice needs to be made between CSBs. A lock-in can now occur on this layer.

- Using multiple clouds adds the challenge of *which* cloud to use.

Overall, this chapter gave an overview of both the problems and possible solutions within the heterogeneous cloud environment. Especially of interest are the two categories of solutions: standardization and brokering. Therefore, in the next chapter we will analyze the current 'state of the art' solutions to get an idea on the feasibility of such solutions.

Chapter 4: **State of the Art**

This chapter shows the existing approaches for tackling the challenge of automating the selection and deployment of applications within the multi-cloud environment. First, section 4.1 describes the current research, methods, and challenges for automated cloud selection. Next, in section 4.2 we describe solutions on different levels considering portability and interoperability challenges. Within this section, the issue of automated deployment is considered from a cloud orchestration viewpoint. Section 4.3 discusses several methods for modeling the cloud. Finally, section 4.4 compiles a list of observations that are of interest to the setup of the automated method that is described in the next chapter.

4.1 Automated Cloud Selection

A suitable cloud deployment scenario is acquired by matching two distinct entities: a software application and the cloud environment. A software application contains specific components, such as code, database, and message queues, whereas the cloud environment provides services that can manage those components. A valid deployment scenario considers the proper alignment of these two entities, including external, explicit constraints that are defined by the software developer or a software user. The proliferation of cloud services as well as continuous changes to both the cloud environment and changing software requirements calls for a dynamic, automated method for cloud selection (Jula et al., 2014; Sun et al., 2014; Wang et al., 2010). One example of such changes has been reported recently, showing that Amazon has changed its cloud prices 44 times since 2006, that Microsoft cut its compute and storage prices respectively by 45% and 65%, and a cumulative decrease of 38% for Google App Engine prices during 2014 (Cloud Spectator, 2015).

Several challenges are related to automated cloud selection, summarized as follows (Jula et al., 2014; Sun et al., 2014):

- An easy to use mechanism for updating data about service providers (e.g. changing prices) should be available;
- Of interest is how to deal with incomplete or out of date information;
- How to describe non-quality attributes, such as availability and reliability;
- How to describe dependencies and conflicts between components;
- How to ensure security when dealing with automated management of services amongst different cloud providers;
- How to model different cloud target groups (e.g. IaaS vs. PaaS vs. SaaS).

Overall, the challenges can be divided in two categories (Sun et al., 2014). First, as automation requires a standardized form of domain modeling, the question is how to store the information in such a way that it can be both retrieved and processed. Indeed, a recently performed literature review identifies a trend favoring semantic modeling (Sun et al., 2014), as this form is machine-readable and implicitly enforces standardization (Dastjerdi & Buyya, 2011). Second, with the data available, the question is how to interpret it and what kind of algorithms to use in order to make decisions. Both challenges will be discussed including existing solutions.

4.1.1 Modeling

Ideally, standardized, machine-readable data about different cloud providers would already be widely available on the internet. Unfortunately, no single consensus has yet been reached about how to provide such information. One attempt is the Cloud Service Market[10], where many different providers (at the time of writing, 168) can be found, ranging from both PaaS and IaaS services but with a strong emphasis on IaaS. Some large cloud providers are (partially) listed such as Amazon AWS and Google App Engine, however other large providers such as Microsoft Azure, Heroku and OpenShift are absent. In addition there is a clear lack of meta-information that can be used to make informed decisions about the different cloud platforms. Finally, with the latest site news message dated in October 2013, the information seems both incomplete and possibly out of date.

An attempt at modeling cloud provider pricing is cloudorado.com[11]. Here many cloud providers' prices can be queried through many different filter criteria. The obvious criteria such as amount of RAM, Storage, and CPU are available. In addition many other filters exists, such as being able to vertically scale without a reboot, whether a free IP is included or if the storage is encrypted. In total more than 130 criteria are included. PaaS providers and services are not included, though the website mentions that they are currently working on this.

One of the main contenders for filling the gap of cloud provider representation is TOSCA, with its rich capabilities for modeling both hierarchies, attributes and dependencies (Sun et al., 2014). One advantage we consider important is that TOSCA cannot only model the cloud environment, but the application's components as well. More importantly, dependencies between these can be model through 'requires' and 'provides' attributes. Its static nature and lack of support for automated selection has already been noted as an interesting and important research direction (Brogi et al., 2014). Work that provides such capabilities using feature models is described in several papers.

One research group has modeled the provided features offered by Amazon AWS within a feature model (Garcıa-Galán et al., 2013). Included in the model are CPU instances, database options, storage facilities and physical deployment locations; their research therefore focuses in the IaaS layer. They use an existing AAFM with a self-written extension for finding an optimal deployment, and validate the performance of this prototype through a case study. With the main goal of this study being the performance of this prototype, they conclude that finding an optimal configuration for a multi-instance setup (e.g. multiple databases for redundancy) quickly becomes too slow. Of interest for further study therefore is to optimize the used algorithms. In addition, they note that a (preferably already existing) Cloud-DSL would be required when dealing with multiple cloud providers. Actual deployment of the found configuration is not part of the study.

[10] http://www.cloudservicemarket.info
[11] https://www.cloudorado.com/cloud_server_comparison.jsp

A multi-cloud selection method using feature models is described in (Quinton et al., 2013a). Four different EFM's are created – each for a specific cloud provider – and a 'cloud ontology' is used to map similar services offered by these provider to a single entity. The addition of this abstraction layer allows an application to, for example; simply ask for a 'MySQL database' provided by this layer, which exposes several options provided by the feature models. A simultaneous multi-cloud configuration was successfully generated, with code and data both residing on a different cloud provider. They do note however that actually deploying the application is not without difficulty, as "*some modifications can be required before the application upload, e.g., setting a correct database connection URL*" (Quinton et al., 2013a).

A similar approach is described in (Wittern, Kuhlenkamp, & Menzel, 2012). A separate feature model coined a 'domain model' is used to describe abstract features, similar to the cloud ontology in the previous approach. A 'service model' is then created for each cloud provider. The mapping between the two is realized by using the same names for nodes in tree, where the previous method specifically linked an ontology node to a service node.

A fourth multi-cloud selection approach is described in (Cavalcante et al., 2012). An interesting difference with the previous two approaches is that this time, all cloud providers are represented within a single feature model. Thus, instead of requiring a separate cloud ontology for the aggregation of similar services, this single EFM achieves this abstraction by placing the similar services under the same node. The method is able to show the costs for different simultaneous multi-cloud deployment configurations, of which a user can manually select one. The authors explain how the software is able to run on different systems using the *factory* design pattern. Unfortunately, the method for analyzing the feature model is not explained, nor do they mention if they actually deployed the application and whether they ran into any problems.

Instead of using feature models to model the cloud environment, they can also be used for modeling the possible components of an application (Paraiso & Haderer, 2012). Variability in this context exists for example of the type of programming language that is chosen. One component may be written in Java, Scala, or Ruby and depending on the chosen cloud technology, the correct component is to be deployed. This will add another method for optimizing cloud performance: instead of just testing different cloud providers, it is also possible to test different components even in other languages. This however will require an elaborate orchestration tool that is able to wire these dynamic components together. For many use cases, this added level of complexity is undesirable. In addition, the feature model is not used for automated selection, making this publication less related to our research.

Another approach for automated selection that does not use a feature model is CloudGenius (Menzel & Ranjan, 2012). This approach supports taking into account multiple criteria, however only supports single-tier applications, e.g. applications consisting out of only a single component. This is of interest when an application exists within a single appliance for which an applicable virtual machine needs to be selected. This requires the various components within the application to be already configured in order to be able to communicate with each other. In other words, this research places its focus on the IaaS layer.

Overall, only limited research is available concerning the usage of FMs within the cloud domain. The existing attempts are promising however and the cloud domain is suited to fit in the FM. The difference between using a single FM and using multiple FMs for each cloud provider is an interesting one, and the best fit for integration with TOSCA will need to be selected. None of the projects included the addition of actually deploying the created deployment scenario, partly because some projects return not a single but multiple scenarios. A summary of the reviewed papers is presented in Table 4.1.

Publication	FM strategy	Number of providers	Service level	Automated Deployment	Used parameters
(Garcıa-Galán et al., 2013)	Multiple	1	IaaS	No	Cost, Capacity
(Quinton et al., 2013a)	Multiple	4	PaaS	No	Capacity
(Wittern et al., 2012)	Multiple	3	SaaS	No	Cost, Capacity
(Cavalcante et al., 2012)	Single	2	PaaS	No	Cost

Table 4.1: Overview of literature that discusses feature models in the context of automated cloud selection. The "FM Strategy" is either with a FM for each cloud provider (multiple), or one FM that encompasses all providers (single).

Also included in Table 4.1 are the parameters used on which to base the selection. This is discussed next.

4.1.2 Decision making

Given a set of data including a software application, constraints and the cloud environment, several challenges are reported in the literature that still await related to the automated selection using this data (Sun et al., 2014). First, the quantification of qualitative parameters provides to be a challenge. Data related to the reputation of a cloud provider, e.g. trust, is hard to quantify as this is very subjective (Bedi, Kaur, & Gupta, 2012).

Next, even if data is quantified, a challenge is how to weigh the different parameters. Three approaches are discussed for deciding the weights of the different parameters (Sun et al., 2014). First, user weighting lets the user decide and provides the possibility to manually make alterations and compare different distributions. Second, through expert weighting the parameter weight distributions are given and constant, and decided through an earlier technique. Only one paper described a pure expert-weighting method, and concludes that the method delivers "*reasonable cloud adoption decisions*" (Saripalli & Pingali, 2011). Third, evenly distributed weights have been used, though such research also recommends the addition of user-input for deciding on the parameter weights (Zeng, Zhao, & Zeng, 2009). A general consensus exists that it is beneficial to have the user decide the weight of the different parameters (Sun et al., 2014).

A majority of the automated selection research deliver deployment scenarios for only a single component (Sun et al., 2014). Thus, such research only attempts to optimize the deployment of a single virtual machine (IaaS) or only a single database (PaaS). When taking into account multiple, possibly dependent components, the complexity quickly rises (Sun et al., 2014). Such research has been performed using different data structures for storing the decision information, such as vectors, matrixes, trees, and feature models. None of these options appears to be significantly more or less effective. However, some data structures are less suited for a dynamic setup. Another distinction found in the literature is the difference between static a dynamic methods. A static methods follows a 'solve once and for all' mentality, where it is hard if not impossible to change the underlying data model at a later point in time (Sun et al., 2014). As has been argued before, we desire a dynamic method as we acknowledge the occurring changes within both the cloud environment and the software application.

A last consideration of interest is *which* parameters to use. This has been defined as one of the research questions within this thesis as follows:

> **SQ2:** "*Which constraints can be modeled to automatically select a deployment scenario for software components?*"

Combing the results from two recently performed literature reviews (Jula et al., 2014; Sun et al., 2014), the parameters that have been used the most are cost and performance. Capability is often used interchangeably with performance, where users can provide a minimum of, for example, RAM requirement (Jula et al., 2014). Other parameters related to performance are response time, availability, and reliability. Less used within studies is reputation. This can measured using number of users and how these users rate the service. Parameters that are scarcely used are durability, usability (e.g. the user friendliness of the user-interface), scalability, and accessibility. This data simply is hardly available (Sun et al., 2014) or in the case of usability, very subjective.

For most of these parameters, a second question is what interval to look at. For example, are we interested in the actual downtime (availability) of a cloud provider in the last hour? Alternatively, do we look at the last day, week, month, or even year? No clear answer to this question can be found but it may explain why most studies look at cost and capability parameters: these are only of interest at the current state. Still, both these parameters have their issues. Cost is typically flexible and pay-per-use, with prices depending on the amount of usage (Jula et al., 2014). Cloud providers may provide a 'free tier', where usage is free for a specified amount of time or storage. In addition, prices typically go down when more resources are used.

Capability, though listed by cloud providers as a constant value, may not at all be constant. Because hardware is typically shared amongst multiple clients (multi-tenancy), performance may fluctuate as a result of 'neighbors' using the same hardware (Dillon et al., 2010). Several projects provide performance information, such as CloudSpectator[12] and CloudHarmony[13]. Whereas the first project contains information for only a small amount of providers, CloudHarmony contains information for many providers and also provides an API for automated performance lookups. CloudHarmony has barely been used by the scientific community for inclusion in cloud selection criteria, though one paper presents positive results based on using CloudHarmony (Hsu, 2014). CloudHarmony did receive some critique in

[12] http://cloudspectator.com/
[13] https://cloudharmony.com/

that they use artificial data for the benchmark instead of real data, which may result in unrealistic performance numbers (Ferrarons et al., 2014).

Summarizing, existing literature provides different interesting points that are of interest to our research. First, a large focus exists on multi-cloud IaaS environments compared with a PaaS approach. Second, though feature models are shown to be a promising method for modeling the cloud environment, only little research exists using these. In addition, of interest is the fact that none of this research actually deploys generated deployment scenarios automatically. Thus, as a third observation, we believe adding TOSCA to this approach will provide a much needed bridge between automated selection and automated deployment. Fourth, a focus on single-service selection suggests more research is needed that automates multi-service automated selection. Again, TOSCA will provide to be a valuable asset, as it provides a flexible structure that includes all data required to both select and deploy a multi-component software application.

4.2 Interoperability & Portability

This section considers the current approaches towards gaining interoperability and portability within the heterogeneous cloud environment. Whatever the solution, some form of standardization is required; the question is *where* to standardize. Portability requires at least some form of commonality. A database cannot be provisioned on a message queue service, which requires some high-level modeling language to distinguish the two. Deploying a message queue on two different providers requires either for the providers to expose the same interfaces, or for the application code to take into account the differences. This section therefore discusses different standardization attempts at different levels. Admittedly, a solution without standardization exists but proper coding principles will quickly turn again to a form of standardization. This is explored in Box 4.1.

To properly and understandably structure these approaches, a layered segmentation with inspiration from both (Ferry, Rossini, Chauvel, Morin, & Solberg, 2013) and (Petcu, 2011) is used. These layers are presented in Figure 4.1. A stack can be considered a specific type of cloud provider. As we will see, cloud PaaS stacks exists that can be installed on private environments, mimicking the functionalities of a public PaaS cloud provider. When this stack is also available as a public cloud provider, both the private and public clouds can be used in a combination, therefore creating a hybrid cloud environment. Because the stacks are the same, automated management can be performed through a single interface.

Using a single stack limits the cloud user in several ways, for example in that interesting functionalities provided by other cloud providers (with different stacks) are out of reach. Therefore, we can seek standardization on a higher level by using a broker API. This API can manage different stacks, again accessed through a single interface. The user now is limited by the cloud stacks supported by the API, thus an active community and ecosystem is to be preferred keeping the API up to date.

The highest-level standardization approach is the use of a cloud orchestrator. This orchestrator will hide the underlying method for communicating with multiple clouds to the user. Standardization is typically gained through an API, with the orchestrator providing additional features such as automated selection, deployment, and such to its user. The orchestrator may also support the use of custom scripts, possibly allowing the user to use other existing APIs or stacks that are not inherently supported by the orchestrator. The orchestrator is therefore the most powerful option, but potentially also the most complicated one.

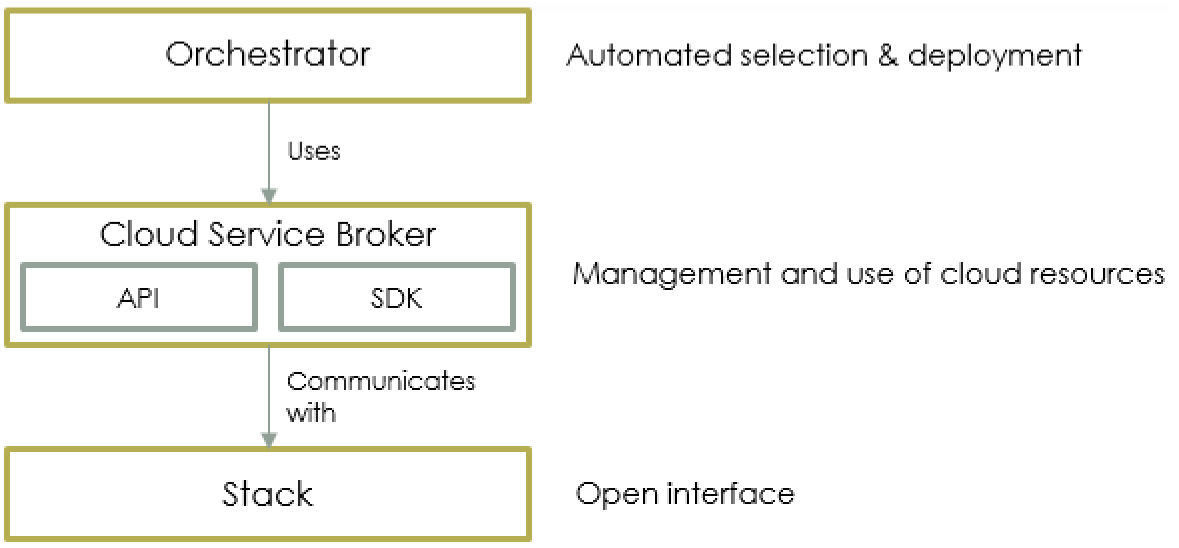


Figure 4.1: Shows the relationships between the different identified layers in the cloud environment

Next, we will further elaborate on the possible approaches to gaining standardization within the multi-cloud environment. In addition, we will show the current methods existing for each of the discussed layers.

Approaching multi-cloud interoperability and portability without standardization

Let us assume for a moment that no form of standardization is applied on any level within the application's code and infrastructure. This will boil down all the way to the code, leading to an un-organized mess with many *if-else* statements. Let's take a look at an example (in pseudocode);

```
function getMovies() {
    // we need to access a database, so first check where it is hosted
    var database_provider = Config.getDatabaseProvider();
    var movies;

    if(database_provider == 'Azure') {
        // use (if available) the Azure SDK
        // specific code that fills the movies variable using the Azure database
    } elseif(database_provider == 'Amazon') {
        // use (if available) the Amazon SDK
        // specific code that fills the movies variable using the Amazon database
    } elseif(database_provider == 'Heroku') {
        // use (if available) the Heroku SDK
        // specific code that fills the movies variable using the Heroku database
    }

    return movies;
}
```

To keep the example small no provider-specific code is added, however one can imagine that a lot of custom code is required for each different provider. Even worse, these code snippets will include a lot of duplicate code amongst different functions. Overall, the code will not adhere to any generally accepted code principles such as SOLID[14]. A logical approach to improve this code is by using design patterns such as the factory method pattern[15]. This new method may look like this;

```
function getMovies() {
    // the DatabaseFactory checks some configuration file where the database
    // is currently hosted, and returns a correct instance
    IDatabase database = DatabaseFactory.getDatabase();

    return database.getMovies();
}
```

What happens here is that the provider-specific code is hidden in specific classes that can easily be re-used within the application. Extending or editing this code is done in one distinctive place, allowing for the above method to remain unedited. Thus, when a new provider is added, neither the above method using that provider, nor the already implemented providers need to be changed. This is a powerful and safe method for extending an application with minimal impact to existing implementations.

Important though is what exactly happened. Through use of the factory method pattern, we created *standardization* at the code level by abstracting the database into a single interface (one common method to communicate with different databases). What this example shows is that it is highly unlikely that much software code will exist that uses no form of standardization to work with different cloud providers. Moreover, if there is, the code will be hard to reuse and extend.

Box 4.1: A code example that shows the repercussions of applying no standardization to a multi-cloud setup

4.2.1 Cloud Stacks

The use of a common cloud stack is the lowest level method for standardizing different cloud environments. A cloud stack is considered as the base infrastructure (IaaS) or platform (PaaS) on which can be build. Therefore, each cloud provider is a stack on its own, and they all differ. When using a common stack for different providers, one interface can be used to communicate with this stack and thus more easily setup a multi-cloud environment. Different providers of course will need to support the

[14] SOLID is an acronym that stands for: **S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation, and **D**ependency inversion. These five principles together should lead to code that is easy to maintain and extend. Further elaboration is considered out of scope for this thesis; information on these principles can be found at http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[15] The factory method pattern is a common code principle that allows software to request a type of code (such as a database), without requiring to know which specific instance of that code is used (e.g. a database code for Amazon or Azure). More information can be found at http://www.oodesign.com/factory-method-pattern.html

same stack, which for example can be done through a cloud federation (see section 3.3). A more common method for using stacks however is by combining the public and private cloud within a hybrid cloud setup. This means that these stacks are installed within a private cloud environment and because they have the same interface as one or more cloud providers, the private cloud environment can easily be used together with a public cloud account.

Of interest to our research are stacks that allow for a multi-cloud setup. Thus, a large PaaS cloud provider such as Heroku[16] will not be discussed, as this is a public-cloud only provider.

Different stacks currently exist. Eucalyptus[17] is an Amazon AWS compatible stack with a focus on the IaaS layer. Because the API is compatible with the Amazon AWS API, this stack can be used concurrently with an AWS account. This stack is therefore a good option for companies who already have experience with the AWS API and want to standardize an (existing) private cloud environment. Another stack with focus on the IaaS layer is Apache CloudStack[18]. This stack includes optional compatibility with Amazon EC2 (computing instance) and Amazon S3 (storage). One public cloud provider, GreenQloud[19], has built its own stack on top of Apache CloudStack. This adds the possibility of creating a hybrid cloud environment with GreenQloud as the public cloud and Apache CloudStack installed on the private environment. A big advantage for both stacks is their compatibility with a public cloud provider, making it possible to setup a hybrid multi-cloud environment on a single interface.

OpenNebula[20] is another IaaS stack with support for Amazon EC2, Azure, and SoftLayer. Within one interface, virtual machines can be started that are hosted on those public cloud providers and VMs within a private cloud environment. Anything other than starting or stopping VMs on the public providers is however not possible, so knowledge about those providers in order to effectively use them will still be required. Recently, OpenNebula released a new stack called vOneCloud[21], an open-source alternative to VMWare's vCloud[22]. These stacks are built specifically for data centers that are managed using VMWare's vCenter Server[23] software. When using the vOneCloud version, it can easily be combined with the default OpenNebula stack.

The IaaS stack with the most adopters is OpenStack[24]. Public clouds amongst the adopters are RackSpace and HP, including some ten others. This stack definitely provides the most options amongst both public and private cloud providers. A Google Trend shows that OpenStack has the most searches compared to some of its competitors[25]. Though it has gained a large user and company adoption, it also

[16] https://www.heroku.com
[17] https://www.eucalyptus.com/
[18] http://cloudstack.apache.org/
[19] https://www.greenqloud.com/
[20] http://opennebula.org/
[21] http://vonecloud.today/
[22] http://www.vmware.com/nl/products/vcloud-suite
[23] http://www.vmware.com/products/vcenter-server/
[24] http://www.openstack.org/
[25] http://bit.ly/12V0uwQ

receives criticism. One of the most common is its modular setup which complicates the installation (Lando, 2014).

Summarizing, these IaaS stacks are mainly of interest to large organizations who want either to become a new public cloud provider, or to setup their own private cloud, possibly combined with a public cloud. With enough adopters for a single stack, it may be of interest to use different public cloud providers simultaneously to provide redundancy. Of course, more of interest to our research is a PaaS stack.

The first PaaS stack, OpenShift[26], comes in three different flavors: online (public cloud), enterprise (servers & private cloud), and origin (all previous plus laptop and pc). The use of both the online and enterprise edition allow an organization to use OpenShift as a hybrid PaaS multi-cloud environment. With the origin version allowing installation on local computers and laptops, developers can easily test applications and help extend OpenShift by writing custom plugins or fixing bugs. PaaS stacks typically come with additional add-ons that can be used to add new services to an application. OpenShift currently provides 26 add-ons, ranging from (no-)SQL databases, monitoring, messaging, search and more.

Cloud Foundry[27] is another PaaS stack that, similarly to OpenShift, comes in three different editions, allowing for a hybrid cloud environment. Less add-ons are available (17), although an exclusive big data suite can easily be added as a service to the application, making this PaaS especially of interest to a specific big-data niche. OpenShift, in comparison, includes auto-scaling which means it can automatically increase or decrease the number of computing instances based on, for example, the amount of traffic (Heller, 2014). Both Cloud Foundry and OpenShift support a similar set of programming languages. Through Iron Foundry[28] the .NET runtime is supported within Cloud Foundry environments.

A PaaS stack built on Cloud Foundry is Stackato[29]. Stackato only has a private cloud version, however as it is compatible with OpenStack which also contains a public cloud version, a multi-cloud setup may still be possible. This would add another layer though, as both OpenStack and Stackato will need to be maintained. Compared with Cloud Foundry, Stackato mainly adds enterprise support such as trainings, resellers, and more possibilities for e.g. phone, e-mail or personal support. Technological differences with Cloud Foundry include a more advanced management dashboard, IDE integration, a persistent file system, and many more features.

Two relatively new PaaS stacks are Deis[30] and Octohost[31]. These stacks are built on top of Docker[32], a new virtualization technology that promises better performance than existing technologies and currently has a lot of traction in the technological community. Their use of Docker allowed them to

[26] https://www.openshift.com/
[27] http://www.pivotal.io/platform-as-a-service/pivotal-cf
[28] http://www.ironfoundry.org/
[29] http://www.stackato.com/
[30] http://deis.io/
[31] http://www.octohost.io/
[32] https://www.docker.com/

quickly support many cloud providers (meaning, deploy the PaaS stack on an IaaS provider) and programming languages. With the current popularity and quick rise of Docker, stacks like these are worth keeping an eye on. However, both stacks are not available as hosted, public clouds. In addition, as they are still relatively new their stability and future directions are unclear.

Overall, these stacks can be used to set up a common platform – either on the infrastructure or on the service level – on multiple private and in some cases public clouds. Especially the combination of private and public, hybrid, is of interest as these are the most flexible setups (Zhang et al., 2010). Privacy-sensitive data can be kept private, whereas other data or applications can be hosted and processed in the often-cheaper public cloud. Within the context of this research, these stacks can certainly be put to use as they easily allow applications to be deployed on multiple clouds. However, especially keeping public clouds in mind, these stacks are limited, and only have a given set of services and functionalities. For example, unique functionalities offered by other cloud providers outside of these stacks are out of reach. We will get back to this subject in subsection 4.2.3. Table 4.1 shows an overview of the stacks discussed in this section.

	Vendor	Service level	Deployment models	License
Eucalyptus	Eucalyptus Systems	IaaS	Private	GPL v3
CloudStack	Apache	IaaS	Private	Apache 2.0
OpenNebula	OpenNebula Systems	IaaS	Private	Apache 2.0
vCloud	VMWare	Iaas	Private	Commercial
OpenStack	OpenStack Foundation	IaaS	Public (through supporting vendors) & Private	Apache 2.0
OpenShift	Red Hat	PaaS	Public & Private	Apache 2.0
Cloud Foundry	Pivotal	PaaS	Public & Private	Apache 2.0
Deis	OpDemand	PaaS	Private	Apache 2.0
Octohost	Two individuals	PaaS	Private	Unknown
Stackato	ActiveState	PaaS	Private	Custom

Table 4.1: Overview of cloud stacks

4.2.2 Cloud Service Brokers

A CSB is an entry point to communicate with a backing service or software component. Within the PaaS cloud computing context, of interest are CSBs that can provision, update, list and remove such components, for example computing services, databases, message queues, and more. In addition, CSBs can be used to communicate with provisioned services, such as requesting the queued messages on a message queue, or insert data into a database.

CSBs come in two flavors. The first is as an SDK (Software Development Kit), a set of functionalities written within and for a specific programming language. The second is an API (Application Programming Interface), which is typically a web service that can receive requests through a REST protocol. An API is language-independent, and can be used by each programming language that can send and receive requests through a REST interface. Every public cloud provider will provide an API that developers can use to connect with the services, and in addition provide several SDKs for different programming languages that can ease the development for that specific language. For example, Microsoft Azure provides an API and SDKs for .NET, Java, Node.js, PHP, Python, and Ruby.

The previous section explored solutions for placing a single cloud stack on multiple providers or servers. This single stack would then be accessed through a single CSB, which would allow one CSB to communicate with services amongst different providers. In this section, we are especially interested in CSBs that are able to communicate with *multiple* stacks. Remember that a single cloud provider is also considered a stack, thus such CSBs could possibly communicate with multiple providers.

	Vendor	Type	Language	Number of supported stacks	License	Latest Release
libCloud	Apache	SDK	Python	+- 40	Apache 2.0	Februari 18, 2015
Deltacloud	Apache	API	-	+- 15	Apache 2.0	April 17, 2013
jclouds	Apache	SDK	Java	+- 15	Apache 2.0	March 29, 2015
fog	N/A	SDK	Ruby	+- 40	MIT	May 7, 2015
mOSAIC	N/A	API	-	+- 6	Apache 2.0	June 18, 2013
PaaS Manager	N/A	API	-	4	-	-

Table 4.2: Overview of multi-cloud cloud service brokers. The latest release is recorded on May 11, 2015

An overview of the five existing CSBs is provided in Table 4.2. It shows that an SDK is available for three different programming languages (Python, Java, and Ruby). Two APIs that are more generally available, although both latest releases are more than 1.5 years ago. Each CSB supports a large number of cloud providers, each of which at least supports the largest cloud providers partially (Amazon, Azure, and Google Cloud Engine). An overview of which stacks exactly are supported is provided in Table 4.3; this table list all stacks discussed in the previous section. Unfortunately, the biggest conclusion that can be made looking at the CSBs is that PaaS support is very much limited. Granted, each SDK and API is able to provision a database, but other than that, the focus is very much on IaaS.

The only exception to this is the PaaS Manager API (Cunha et al., 2014). This manager supports four different PaaS platforms, of which one was included in the discussion in the previous section: Cloud Foundry. In addition, an extension to Cloud Foundry called Iron Foundry[33] is supported, which adds .NET

[33] http://www.ironfoundry.org/

capabilities to Cloud Foundry. The two other supported PaaS stacks are Heroku[34] and CloudBees[35]. These are public-cloud only PaaS providers and thus haven't been discussed previously. Though the authors criticize the lack of practical results from other, often-academical attempts, their work is not released publicly. This way no validation or extension of their method is possible. Their success is motivating however, and shows that developing a PaaS broker CSB is in fact possible.

Looking at Table 4.3, it is clear that the most-supported stack is OpenStack. With our focus on the PaaS environment, this is of little interest of us. Would our method be extended towards the IaaS environment, OpenStack may be the first contender for being added.

Layer	Stack	libCloud	Deltacloud	jclouds	fog	mOSAIC	PaaS Manager
IaaS	Eucalyptus	X	X			X	
	CloudStack	X		X	X		
	OpenNebula	X	X			X	
	vCloud	X			X		
	OpenStack	X	X	X	X	X	
PaaS	OpenShift						
	Cloud Foundry						X (and Iron Foundry)
	Deis						
	Octohost						
	Stackato						
	Heroku						X
	CloudBees						X

Table 4.3: Shows which cloud stacks are supported by which libraries

4.2.3 Cloud Orchestrators

Whereas an API or SDK as discussed in the previous section is able to communicate with multiple cloud stacks, an orchestrator can do so as well but with some additional services. Still a buzzword in the industry, cloud orchestration is defined differently by some authors with no single one being generally accepted or used (Bousselmi, Brahmi, & Gammoudi, 2014). Here we consider cloud orchestration as the automation of application management spanning multiple cloud providers. This includes selection, configuration, deployment, monitoring and logging, a user-interface for management, dependency management, as well as re-deployment of applications possibly towards other providers (e.g. migration).

The first cloud orchestrator discussed is Cloudify[36], one that presents their own definition for cloud orchestration: '*automation on steroids*'[37]. An application is described using TOSCA-based blueprints (the

[34] https://www.heroku.com/
[35] https://www.cloudbees.com/
[36] http://getcloudify.org/
[37] http://getcloudify.org/FAQ_cloud_devops_automation.html

'topology' in TOSCA). The blueprint is used for automated management of that application. An installation is required in order to use Cloudify – a public web interface does not exist. Cloudify is native to OpenStack, which means that deploying and managing an application on an OpenStack public or private cloud is easy. This is because support for the OpenStack API is built in, thus enabling Cloudify to provision and further manage OpenStack nodes such as compute or database servers. The plugin system is powerful as it allows other clouds and tools to be supported by Cloudify. Other stacks that are supported through plugins are Apache CloudStack, SoftLayer and VMWare. In addition, a plugin exists for the libCloud library discussed in the previous section. Finally, automation tools such as Puppet, SaltStack, and Chef are supported as well. Cloudify is able to execute scripts that make use of these libraries or tools, indirectly giving it support for many providers that can be used combined.

Though must plugins are currently 'official' (having been made by the Cloudify vendors), some other plugins are starting to emerge that are made by other developers. Would someone have written a plugin for Windows Azure, it could be easily added to a Cloudify installation allowing for easy management of Azure cloud resources. It will be interesting to see if Cloudify is able to give rise to a large backing user base that develops plugins and keeps them up to date.

Another cloud orchestrator is Scalr[38], dubbed by its own creators as a 'cloud management platform'. Different from Cloudify, Scalr is available as a web service through a user-friendly interface. An enterprise edition also exists that allows Scalr to be installed on a local premises. Also different from Cloudify is the lack of a plugin environment that allows for the easy extension of the software. Scalr provides support for the largest public and private cloud providers, including Amazon EC2, Google Compute Engine, OpenStack and Eucalyptus. In the web interface, credentials for these cloud platforms can be provided after which Scalr can easily provision servers with custom or predefined images.

Both Cloudify and Scalr are very much focused on the IaaS layer. Scalr is a good choice for a multi-cloud setup in this context, though when one wants to use a cloud provider that is not supported by Scalr, this will become problematic. Cloudify however is very much focused on reusability and extensibility, possibly executing scripts and leveraging existing tools and API's to manage PaaS as well as IaaS services.

Another service-oriented orchestrator is soCloud, described in (Paraiso, Merle, & Seinturier, 2014). This orchestrator places focus on the challenges of portability, provisioning, elasticity and fail-over. With portability and provisioning tackled, it becomes possible to use multiple clouds to increase the performance of software applications. Several PaaS environments are supported through an underlying middleware platform called FraSCAti (Merle, Rouvoy, & Seinturier, 2011). Unfortunately, soCloud is highly dependent on Java and does not support any component or service that is not related to Java.

Also originated from the academical community, MODAClouds provides a method for using multiple IaaS and PaaS cloud environments interchangeably (Ardagna et al., 2012). Their noted challenges are mainly from a business perspective: to avoid vendor lock-in, manage risks and assure delivered quality. MODAClouds uses mOSAIC as the underlying middleware platform (Petcu et al., 2013). This API is very

[38] http://www.scalr.com/

much scattered and not standardized, making it hard to use in a practical environment (Paraiso et al., 2014). MODAClouds is still very much a concept and no practical results have been released yet.

Both soCloud and MODAClouds make a clear distinction between their middleware layer that communicates with different cloud platforms (respectively FraSCAti and mOSAIC) and their orchestration features. Scalr, on the other hand, is more closed and shares little about their middleware setup. The mere existence of the middleware layer is of interest, as this is functionally the same as the library layer discussed in the previous sub-section. Cloudify, for example, has a plugin that supports the libCloud library. An orchestrator with a flexible setup is therefore able to use any existing library, greatly enhancing its capabilities. Whereas Cloudify supports OpenStack out of the box, through the libCloud plugin it also (partly) supports Eucalyptus, CloudStack, OpenNebula and vCloud.

This raises the question what exactly is missing to allow for portability and interoperability between different PaaS providers. As we have seen, no library yet exists that supports different PaaS providers and that is openly available. Multiple orchestrators do exists, and their flexibility allows them to use a PaaS library if desired. Clearly the gap exists in the middle layer, thus further research should focus on middleware that is able to manage resources for multiple PaaS providers.

4.3 Observations

As in the previous chapter, this chapter brought some things to light that are of interest to the context of our research. Looking at the existing approaches in this chapter, we note the following observations.

- There is a strong focus on IaaS research, both related to automated selection and to interoperability and portability. This is supported by a literature review concerning vendor lock-in (Silva, Rose, & Calinescu, 2013), and further motivates us to put our focus on the PaaS domain.

- Feature models can and have been used in two different ways for the cloud domain: each cloud provider in a single FM, or all providers in a single FM. Which is ‘best’ is especially of interest. Because we have chosen TOSCA to model our software applications, we seek a method that allows for integration between TOSCA and feature models. Extending TOSCA with automated selection capabilities has already been suggested as an interesting research direction (Brogi et al., 2014).

- No attempt has yet been made to automatically deploy multi-cloud configuration that have been generated with a feature model. This will therefore be the first research that attempts this.

- Most research concerning automated selection considers only a single component to deploy. Selecting a deployment location for multiple components at once is therefore another interesting research approach.

- There is an interesting hierarchy in the cloud environment domain between orchestrators, libraries, and stacks. Because orchestrators use libraries to manage stack instances, these orchestrators provide the highest level of flexibility.

- The largest gap concerning PaaS portability & interoperability solutions is on the CBS layer. Orchestrators are typically flexible enough to support both PaaS and IaaS, though multiple solutions exist for this layer.

The reported findings in this chapter show the existing approaches for automated cloud selection and deployment. The unlikelihood of cloud standardization has been strengthened and a strong focus on IaaS research is evident. These conclusions together with some other noted observations provide a clear research gap and therefore a scope for our own research, which will be discussed in the next section.

Chapter 5: **A Method for Automated Cloud Selection & Deployment**

The previous chapters presented an overview of the current state of multi-cloud computing. This overview is key in deciding which approach to take within the context of this research. This chapter presents an automated method for multi-platform selection, configuration, and deployment. First, we will discuss the rationale for the scope we have decided on within this research. Next, we define some requirements for the method. After presenting some scenario's we envision for the method, the method itself is presented. Finally, we discuss the implementation of the method.

5.1 Rationale

First and foremost, we envision a method that automates multi-platform selection, configuration, and deployment. In previous chapters, we have discussed the heterogeneity of the cloud environment and its ever-increasing popularity, combined with the issue of vendor lock-in. Some solutions have already been proposed, some of which followed by a concrete, working product. Our goal is to develop a prototype based on a new solution from a new perspective on the problem.

This chapter discusses the solution – the method – for the stated problem. The next chapter provides more detail on the developed prototype that implements this solution.

Next, we provide some different perspectives on the solution and present the rationale for the decision we took.

5.1.1 Standardization vs. brokering

Analyzing the existing literature has shown us that two different approaches can be taken to tackle the problem. Standardizing different cloud environments is nowhere in sight, and perhaps will never even happen. A multitude of standards already exists even with support from some big names in the technology sector. Taking this route appears to be a futile attempt.

A brokering service will therefore be developed that is able to deploy an application in the cloud. A number of cloud stacks will be supported. Our demands for automated selection and deployment call for a *Cloud Orchestrator* (see Figure 4.1). As we have seen, given a flexible enough orchestrator, it will be possible to utilize existing *Cloud Service Brokers*. A number of CSB's already exist (see Table 4.2), meaning we may be able to use some to fasten development speed.

Of course, a number of cloud orchestrators also already exist. However, a gap exists that will allow us to build something new.

5.1.2 IaaS vs. PaaS

One of the noted observations in the previous chapter is the focus on IaaS for existing solutions. A number of different cloud service brokers exist that are able to communicate with multiple IaaS cloud platforms. In addition, existing cloud orchestrators also focus exclusively on the IaaS layer. Clearly, a gap exists within the PaaS layer. Therefore, the focus of our research will be primarily on the PaaS layer. With no other attempts found that discuss the development of a multi-cloud PaaS broker/orchestrator; this should be a very interesting research approach.

5.1.3 Scope

Of course, with every research it is import to decide on a narrow scope. Our method will focus on the successful selection of an environment and deployment within that environment of a software application. Box 4.1 shows how software code may be structured given a multi-cloud environment. This, however, is outside our scope. Any software application we will use to test and validate our method has been prepared so that it will be able to operate in the environments we support.

In addition, our method will focus on *initial* deployment. Of interest of course is how to update (as in, *re-deploy*) an application in the cloud. However, for this research we will focus solely on the initial deployment of the application. Automatically re-deploying an application – even outside the cloud context – is a whole research topic by itself. Adding such functionality will not teach us much more about the (meta-)portability and interoperability issues within the cloud environment.

5.2 Requirements

Given that we now have a set scope, we define the following set of requirements in order to give more focus to the method that is to be constructed.

- *Focus on (meta-)portability & interoperability.* Because not much research yet exists within the PaaS domain, construction alone of this method should present us with many lessons about (meta-)portability and interoperability issues within the cloud environment.

- *Flexibility*. The cloud environment is a versatile, ever-changing environment. Cloud services appear, disappear, and each continuously updates their offerings to further increase their competitive advantage. The method should therefore be easily *maintained* and *extended*. Both cloud providers and cloud services should easily be added, updated, or removed.

- *Local deployment*. Though one might say that the cloud environment is already heterogeneous enough, also of interest is how to add local deployment to this mix. This may be of interest for demo purposes or when migrating an application to the cloud in a phased approach, where not all components are yet deployed on the cloud.

- *TOSCA*. With TOSCA being the current leading standard concerning cloud portability, this issue should be tackled using this standard. TOSCA provides a format for defining application’s components and the environment thus should provide to be a bridge between our application and the multi-cloud environment.

- *Environment modeling*. As mentioned, given the ability to deploy an application on multiple cloud providers, the question arises *where* to deploy the application. Ideally, this should be no concern with the method automatically deciding this based on a set of constraints. Before deploying an application, this method should therefore first generate a deployment scenario.

Fulfilling these requirements will provide us with a deep insight into the challenges that were discussed in earlier chapters.

5.3 Scenarios

Based on the previous requirements, and considering a clear distinction between selection and deployment, we consider the following two scenario's – or 'entry points' – for this method.

1. *I do not care where my application is deployed, take this set of requirements, and deploy it in the best location.*
   In this scenario, we first need to decide where to deploy the application based on the set of requirements that was given. Therefore, we first need our information about the cloud environment, and map the requirements against it to search for the 'best' deployment scenario. What this 'best' scenario is will be discussed later.

2. *I know exactly where I want to deploy my application and with which parameters.*
   In this scenario, we can skip the automated selection and immediately move towards automated deployment. This requires a slightly different input to be given to the method.

5.4 Specification

In general, the method is divided into three parts. First, a prerequisite for the method to be used is to have a model of the platform environment. Without any knowledge about the cloud and local environments, no decisions about deployment locations can be made. Second, given an application description and requirements, these models are automatically analyzed to generate a viable deployment scenario. Third, the generated deployment scenario is automatically deployed. Figure 5.1 shows a diagram of the complete method.

5.4.1 Modeling

We consider the domain knowledge to consist out of three distinctive parts. First is the cloud and platform environment, which is where the different application components can be deployed. Second is the component-based software application including dependencies between these components. In addition, the application exposes several implicit constraints, as for example the used programming language requires a specific compute environment. Third and last are explicit constraints that are of interest to the user, such as costs, geographical deployment location, and performance. First, we discuss the modeling of the cloud environment and the explicit constraints. Next, we discuss the modeling of the software application and its implicit constraints.

Cloud environment and explicit constraints. The previous chapter has shown how feature models were used to model the cloud environment. Two different strategies were noted, one with using a single feature model to encompass the entire environment. The second strategy is through using a different feature model for each platform. This latter strategy requires the use of an extra layer often called an ‘ontology’ (Quinton, Haderer, Rouvoy, & Duchien, 2013b), to map identical concepts (e.g. an ‘SQL Server’ type) to each other. An example is shown in Figure 5.2.

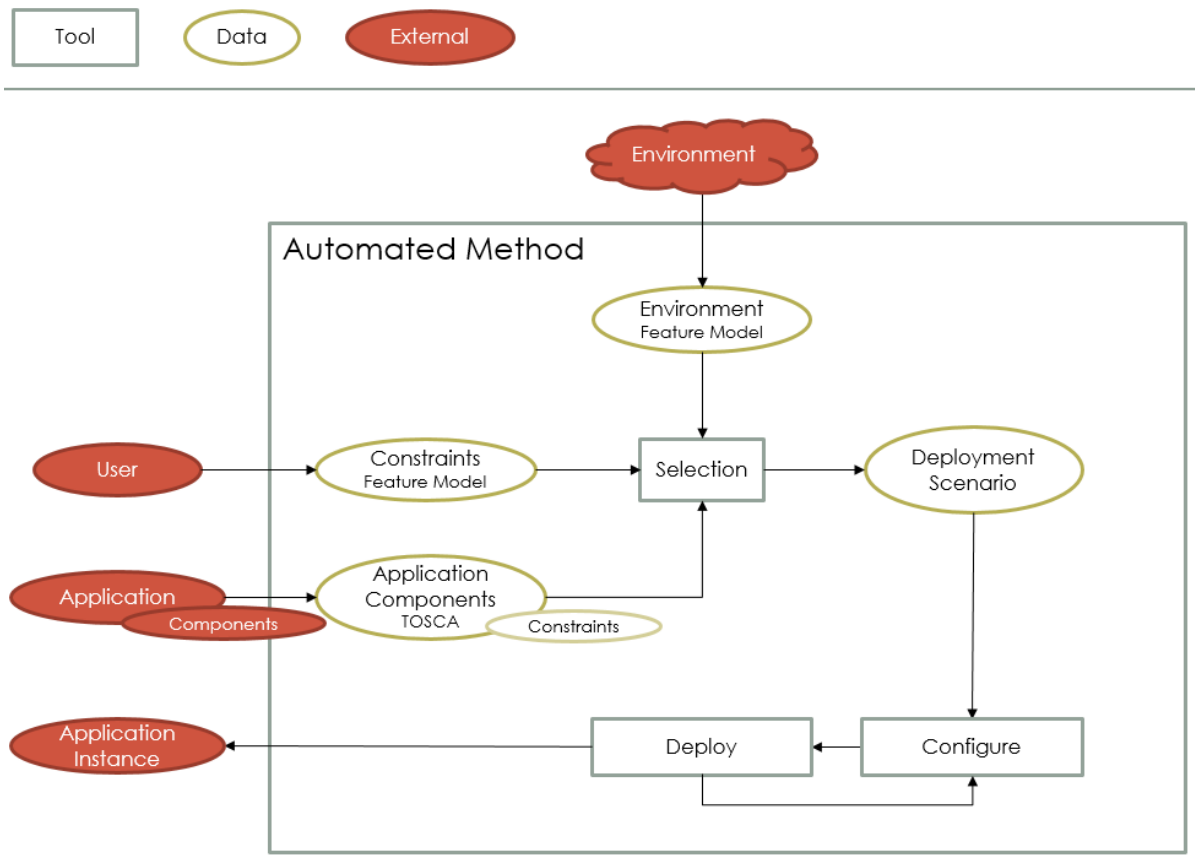


Figure 5.1: Overview of the model.

Placing a cloud provider at the root of a feature model may seem as a logical and obvious choice intuitively. After all, a cloud provider encompasses everything else within the feature tree. Unfortunately, the authors agree, as they do not motivate their design decision. Another approach to take however is too use a *cloud service* (e.g. a *software component*) as the root of the feature model. This approach is not new and has already been tested within the field of context-aware and adaptive software systems (Trinidad, Cortés, Peña, & Benavides, 2007). It’s especially useful when substituting components on the fly (Cetina, Fons, & Pelechano, 2008), though re-deployment of course is not within the scope of our research.

A critical note in our research is that we are deploying multi-component applications. Thus, from the standpoint of each distinctive component within this application, we require a specific cloud service where we can deploy this component. In that sense, the component type is a first-class citizen. It is therefore easier to start searching from a single cloud service root, instead of querying a number of roots within different cloud provider feature models – as is happening in Figure 5.2. Extra information is required to know *where* in the feature model one needs to look.

A procedure for building component-based feature models is followed to define our feature models. First, two assumptions are made when designing component-based feature models (Trinidad et al., 2007):

- *A feature can be mapped onto a component in the component model*; to follow this assumption, we define a feature to be equal to a cloud service. A database or web role and such is therefore a feature or component of the application.
- *The feature model is built thinking about component structure*; this assumption plays a key role in our integration of TOSCA with feature models. The component structure we use within the feature models follows the component structure as defined by TOSCA. Each supported component within TOSCA (web role, database, etc.) is a component, each of which has its own feature model.

Next, four steps are proposed for building the set of feature models (Trinidad et al., 2007):

1. *Defining the core architecture*. The core set of features are the cloud services that are supported by our prototype. Right below the roots of our feature models, we will place each cloud provider that is supported by our prototype.
2. *Defining the dynamic architecture*. The dynamic architecture is essentially every node placed under the feature model's root. An example is the geographical location of a deployed component. Each supported location is a node within the tree, thus an option within the feature model.
3. *Adding the configurator*. The configurator is the component that provides the dynamicity of the feature model. Amongst its responsibilities are the following (Trinidad et al., 2007):
   - It knows the feature model
   - It centralizes the feature de/activation requests.
   - It decides which features must be de/activated because of another feature de/activation.

   The functioning of our Configurator is further explained in the next sub-section about automated selection.
4. *Defining the initial product*. Within the context of our research, the initial product is the initial deployment of our software application. As we do not support re-deployment, this step is trivial.

A next consideration to take is whether to build dynamic feature models based on the provided application model. Take, for example, an application that contains a compute node, a worker node, and a MongoDB node. A feature model may be dynamically constructed that grabs these three existing feature models and puts them in a single model. This approach is depicted in Figure 5.3. The three colored nodes represent the cloud services and an arbitrary root is added on top.

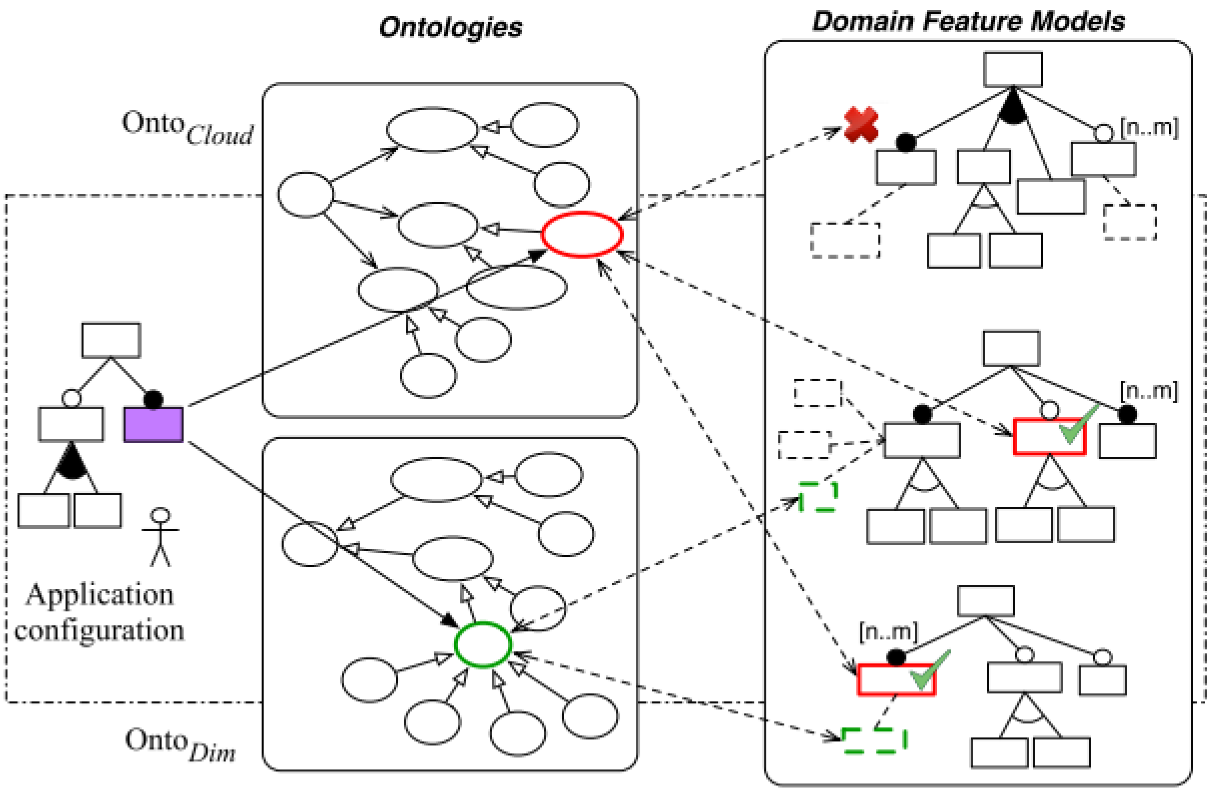


Figure 5.2: An example of using multiple feature models to model the cloud environment (Quinton et al., 2013b). In the middle, we see a 'cloud ontology' (OntoCloud) that is used to map related cloud services. The attributes for these services (such as the specific hardware configuration for a compute instance) are modeled in a different ontology (OntoDim). The application configuration references this ontology, which in turns references the correct concepts within each feature model on the right. The 'Domain Feature Models' are different cloud providers.

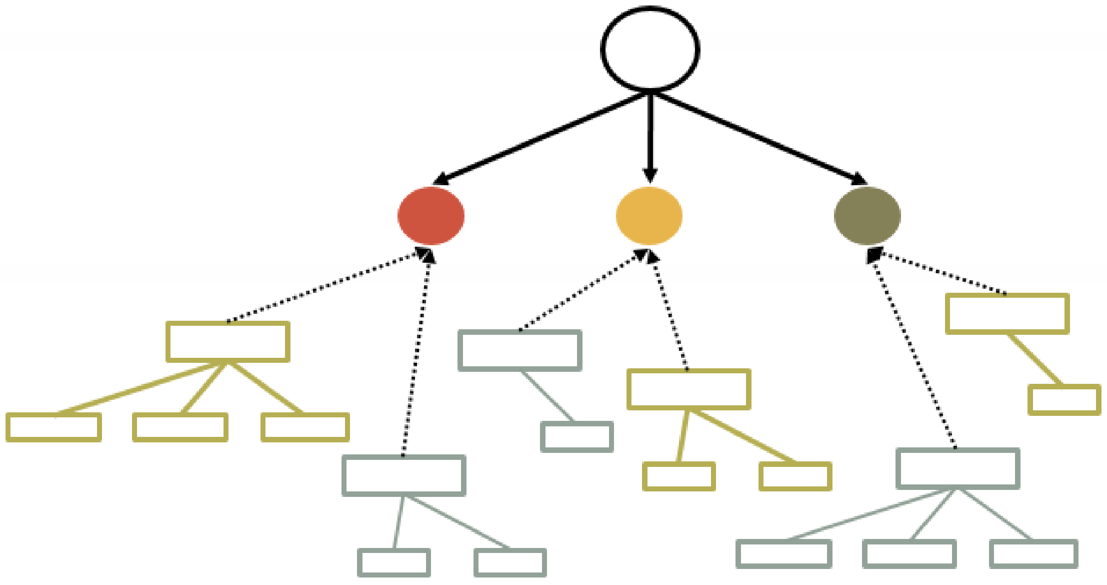


Figure 5.3: An example of a feature model. We propose that the top root is not required. The three first-level nodes are three different cloud services. The differently colored trees below are different cloud providers.

Figure 5.3 shows another important characteristic of the modeled cloud environment: not the different cloud services, nor the different cloud providers show any inter-dependencies. Indeed, no dependencies have been found that exclude the use of a certain cloud service based on a decision made concerning a different cloud service – nor with cloud providers. This isn't unexpected: other research that used multiple feature models to model the cloud environment also didn't include any dependencies between cloud services or providers (Quinton et al., 2013a). Another approach using a single feature model for every cloud service and provider neither showed any such dependencies (Cavalcante et al., 2012).

In fact, generating a single feature model to process is problematic. Given that each cloud service has 100 options – which is conservative – the combined feature model will have 100 * 100 * 100 = 1000000 (one million) options. This is because all 100 options of the compute service will be combined with every 100 options of the worker service, and with all 100 options of the MongoDB service. A global optimum is searched for, which is pointless, as the cloud services do not contain any inter-dependencies. We can therefore safely keep the three feature models separate, and analyze 100 different options for each feature model – in total 300.

Analyzing feature models has been achieved by using FaMa (Garcıa-Galán et al., 2013), a Java-based tool that can analyze feature models. As this tool requires a specific format for describing feature models, we were forced to use this format for describing feature models. These text files quickly become quite large, thus only a partial file is shown in **Code Listing 5.1:** Code Listing 5.1.

```
%Relationships
Mongo: Configuration Provider_Heroku;

Provider_Heroku: Heroku_Hardware Heroku_Deployment Heroku_Location;

Heroku_Deployment: [1,1]{Heroku_Deployment_SingleAZ Heroku_Deployment_MultiAZ};

Heroku_Location: [1,1]{Heroku_Location_EU Heroku_Location_US};

Heroku_Hardware: [1,1]{Heroku_Hardware_Sandbox Heroku_Hardware_SM ...
Heroku_Hardware_M6};

%Attributes
Configuration.Storage: Integer[1 to 1000000], 0, 0;
Configuration.Location: [1,2], 1, 1;
Configuration.MultiAZ: [0,1], 0, 0;
Configuration.HourlyCosts: Integer[0 to 100000], 0, 0;

%Constraints
Heroku_Deployment_MultiAZ EXCLUDES Heroku_Hardware_Sandbox;
Heroku_Deployment_MultiAZ EXCLUDES Heroku_Hardware_SM;

Heroku_Deployment_SingleAZ IMPLIES Configuration.MultiAZ == 0;
Heroku_Deployment_MultiAZ IMPLIES Configuration.MultiAZ == 1;

Heroku_Location_EU IMPLIES Configuration.Location == 1;
Heroku_Location_US IMPLIES Configuration.Location == 2;

Heroku_Hardware_Sandbox IMPLIES Configuration.Storage == 496;
Heroku_Hardware_SM IMPLIES Configuration.Storage == 2000;
Heroku_Hardware_M6 IMPLIES Configuration.Storage == 700000;
```

```
Heroku_Hardware_Sandbox IMPLIES Configuration.HourlyCosts == 0;
Heroku_Hardware_SM IMPLIES Configuration.HourlyCosts == 20;
Heroku_Hardware_M6 IMPLIES Configuration.HourlyCosts == 65833;

Configuration {
    HourlyCosts < 50;
    Storage > 1000;
    Location == 1;
}
```

Code Listing 5.1: The partial MongoDB feature model. Left out are 8 hardware configurations (note the '…'). Everything else is shown.

This example shows a part of the MongoDB feature model. Within our scope, only Heroku is supported for MongoDB deployments. In the example, only three hardware configurations are shown (note the '…'; in total there are eleven options). The top part of the file (starting with `%Relationships`) shows the nodes and relationships of the feature model. The notation is set-based: the top `Mongo` node contains two mandatory sub-nodes. The `Heroku_Deployment` node contains two possible sub-nodes, of which exactly one needs to be selected, as depicted by the cardinality constraint `[1, 1]` (at least and at most one). A better readable version of this is shown in Figure 5.4. Next, starting at the `%Attributes` line, default values, null values and ranges for node attributes are shown. For example, the `MultiAZ` attribute can be either 0 or 1. Both its default value and null value (the value is when this node is not chosen) is 0.

Starting at the `%Constraints` line, two constraints are shown. When a `MultiAZ` environment is considered, these lines will restrict the analyzer from choosing a `Sandbox` or `SM` hardware configuration. Next, (also within the `%Constraints` section) we see a number of implications. For example, when we select the `Heroku_Hardware_Sandbox` hardware option, we set the chosen `Storage` size to 496MB and the `HourlyCosts` to 0 (this is a free configuration). Finally, we can see some more constraints. These constraints are actually received from the user, whereas the earlier constraints are static. This is where the user will inject some specific requirements, such as a required storage of at least 1000MB and no hourly costs of more than 50 cents.

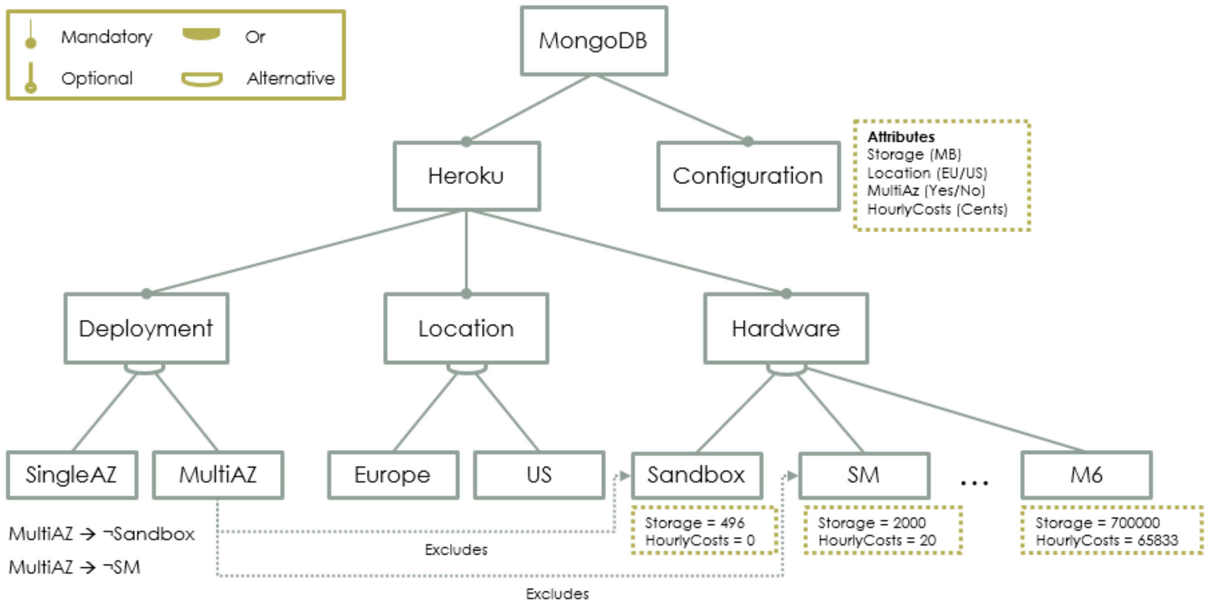


Figure 5.4: The MongoDB Feature Model. The 'Configuration' node is not part of the solution space but is used solely for the constraints through using the attributes. The prefix text used in the text file (e.g. **Heroku_Deployment**_MultiAZ) is removed for better readability.

The names of the nodes are rather long and contain a lot of duplicate text. This is required for parsing the file and because node names need to be unique. Would another provider be added to this feature model that also has a `SM` hardware configuration, we would otherwise have duplicate names.

In the previous chapter, we analyzed which constraints are currently used within the academical literature for automated cloud selection. Cost and hardware capacity were by far used the most. Supported constraints in the prototype are the following;

- **Provider** (Azure, Amazon, Heroku, Local)
- **CSharpVersion** (4.0, 4.5)
- **PHPVersion** (5.4, 5.5, 5.6)
- **RAM** (In MegaBytes)
- **CPU** (In number of cores)
- **HourlyCosts** (In cents/1000) (e.g. 0,018 == 18 cents)
- **Location** (Europe, US)
- **Platform** (Linux, Windows)

One constraint we zoom in is the pricing. If we look at the pricing page of AWS EC2 instances[39], we can see many different options. Three instance types exists, namely on demand, reserved and spot instances. Through spot instances it is possible to bid for unused server capacity, which is typically cheaper than the other two options. In addition Amazon provides Volume Discounts. This in total is difficult to successfully put into a single model. Different criteria exist for which instance type is best for

[39] http://aws.amazon.com/ec2/pricing/

which situation. Only when these criteria are properly known and can be modeled in a quantifiable manner, it can be automated. This alone can be considered a study by itself.

Software application and implicit constraints. The cloud environment and any user-specific constraints are now modeled using feature models. We are only halfway there: the software application and its implicit constraints are modeled using TOSCA. The TOSCA standard describes the means to make connections between different application and environment components. For example, an application compute instance may require a compute container (PaaS), which subsequently may require a compute server (IaaS). One example of such a topology is depicted in Figure 5.5. Here we see that a deployment scenario is already known. For example, both the *Product REST API* and the *Product Database* will be deployed on the Amazon AWS cloud provider. Of course, we desire such properties to be decided upon automatically. We therefore need to take a step back, and first model the application without such specifics.

A YAML[40] notation is used to model the application's components. What we see in Code Listing 5.2 is an example of the TOSCA Simple Profile in YAML, Version 1.0[41]. TOSCA can be used in both XML and YAML. The reason we choose to use YAML is because this YAML version is newer than the currently available XML version.

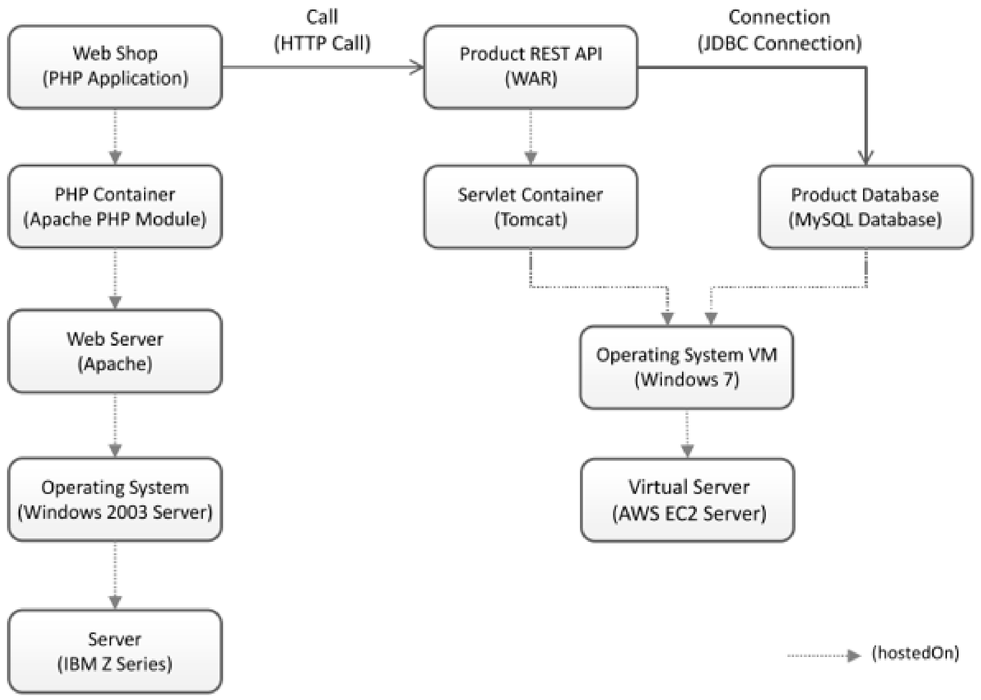


Figure 5.5: An example of a TOSCA topology. Taken from (Binz et al., 2014).

```
application:
  benchmark:
    type: compute
```

[40] http://yaml.org/

[41] http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html

```
      language: csharp
      properties:
        name: ORTEC.Benchmark.WebUI
        source: path\to\source
      requirements:
        database_endpoint:
          node: db
          relationship: ortec.relationships.csharp_to_mongo
      constraints:
        Provider: "== 2"
        CSharpVersion: "== 45"
        Location: "== 1"
    worker:
      type: worker
      language: csharp
      properties:
        name: Worker1
        source: path\to\source
      requirements:
        database_endpoint:
          node: db
          relationship: ortec.relationships.csharp_to_mongo
      constraints:
        Provider: "== 1"
        CSharpVersion: "== 45"
        Location: "== 2"
    db:
      type: mongo
```

Code Listing 5.2: An example of a TOSCA file with three nodes (`benchmark`, `worker`, and `db`) modeled in YAML.

In addition, we consider YAML to be easier to read, which eases the explanation of how TOSCA is concretely used within this method.

The above YAML file is a model of an existing application developed within ORTEC. It shows three different components, namely `benchmark` (a compute type), `worker` (a worker type), and `db` (a MongoDB type). The names could be replaced with any other name, and the model would still be valid. For both the `benchmark` and `worker` node, we can see a requirement named `database_endpoint`. This name, again, can be whatever the creator deems to be a good description. This requirement points to an existing node (`db`), and in addition has a 'type' of `relationship`. In both cases, the `relationship` is one from C# code to MongoDB. This information is required to know what kind of actions need to be performed to properly setup this relationship. The `language` property of both the `compute` and `worker` components is in this example `csharp`. This information is required to be able to call the correct deployment scripts at a later stage. Other programming languages can also be deployed using this method.

The implicit constraints should by now already be apparent. First, a `compute` type node will require a different container than a `database` type node. Therefore, this property constraints the number of possible deployment locations for this node. Second, the `language` property implies another subset of possible containers. A C# application will not be able to run within an Apache container setup for PHP.

The explicit constraints are easier (one might say, 'explicitly') visible. Again, both the `benchmark` and `worker` node contain a list of explicit constraints. This notation will seem very familiar, as we already saw this notation in the previous sub-section as we described the constraints notation within feature models. Indeed, the notation used in the TOSCA file is the same, as these constraints only exist to be injected into this feature model. As the feature models only support integers for attributes, we see a consequence of this in the TOSCA file. The `Location: "== 1"` denotes Europe, whereas the integer `2` would denote the US. As for the `provider` constraint, the `1` denotes Azure whereas a `2` denotes Amazon AWS. We consider this approach to be sufficient for a prototype – a friendlier User Interface would be preferred above working with the above files. This however is considered outside of scope for this research.

A very critical property is each component is the list of properties. Properties such as the name or the location of the source code are listed here. As we will see later, during the deployment process this list is extended with run-time properties. An example is an endpoint for a database, or an IP address for a compute instance. This information is required for relationships to be made, e.g. inject the database endpoint into the compute instance before deploying this node. After deployment, a new TOSCA file is generated containing these new properties, which may be required for automatically destroying the nodes.

When comparing the above file with the TOSCA specification, we can note some differences. First, the constraints are modeled completely different. Within TOSCA, the constraints are specified in Code Listing 5.3.

```
target_filter:
  properties:
    - num_cpus: { in_range: [ 1, 4 ] }
    - mem_size: { greater_or_equal: 2 }
  capabilities:
    - os:
          properties:
            - architecture: x86_64
            - type: linux
            - distribution: ubuntu
```

Code Listing 5.3: Constraint modeling as defined by the TOSCA specification.

This snippet is an exact copy of an example within the TOSCA specification, and its notation is quite different from our `constraints` list. The reason for this change is simple: it eases the integration with feature models. Our parser could be rewritten to support the exact TOSCA specification; however time is saved by using a simpler notation that requires less sophisticated parsing.

A larger, more conceptual difference is the use of node types. TOSCA specifies that an application and its container are two different nodes. In Figure 5.5, for example, we can see on the left-top a PHP application that requires a PHP container. Within our implementation, these nodes are specified as one node. A first reason for this design choice is the mere fact that the amount of required nodes is shrunk by half, which improves readability and maintainability. Second, because much information about the container nodes (read: the cloud services) are stored in the feature models, specifying these nodes in TOSCA is rather trivial. Therefore we decided to use an is-a relationship by using inheritance instead of a requires-relationship by adding an extra `requirement` for the PaaS container.

Now that we have modeled an application, and have modeled the cloud environment through the use of a set of feature models, we can combine these models to generate a deployment scenario for the modeled application.

5.4.2 Automated Selection

Figure 5.1 shows how the automated selection step takes three different inputs. These inputs are modeled using TOSCA and feature models as has been described in the previous sub-section. The main goal of this step is to convert the received application model into a deployment scenario. Considering the received constraints, the cloud environment is analyzed to find the most suitable location and configuration for deployment.

As a separate feature model exists for each node type, a first implicit constraint is adhered through selecting the correct feature model based on the node. Therefore, a MongoDB deployment scenario selection – selected by using the MongoDB feature model - will not be generated for an SQL Server node. A similar approach, simply matching node types with feature model names, was described in section 4.1.1 though in that situation, an extra ontology layer was used (Wittern et al., 2012). Next, the explicit constraints described in the application model are dynamically injected into this feature model. Because we decided to use a different notation for the constraints within TOSCA, we can easily copy and paste the constraints into the feature model. Already, we have a valid feature model that can be processed by the analyzer.

Figure 5.6 shows an example of generating a valid scenario for a sql server component. The application contains three components and two types: two compute instances and one sql server instance. This requires us to load two feature models that are used to generate the scenarios. Both feature models are subsets of the complete models. The sql server feature model is loaded by matching its name with the component type.

The previous sub-section already mentioned the Configurator, a component that *"provides an interface for the feature components to communicate it the de/activation of a set of non–core features."* (Trinidad et al., 2007). Our non-core features (e.g. variations of the cloud services) are modeled both using nodes and using attributes. Figure 5.4 shows how the geographical location of the component is modeling using nodes. The same figure also shows how the storage size of the MongoDB is modeled using attributes. Both types of variability are converted into attributes of the Configurator, through which we can use a single node to set all our constraints. The Configurator can therefore activate or deactivate nodes through its set constraints. Figure 5.6 shows only a few attributes to improve readability. In addition, some of the feature model rules that set these attributes are listed. The rules for the compute type show that the combination of the OS platform (Windows or Linux) with the selected hardware configuration decides the hourly costs. Attributes can therefore depend on the value of more than one other attribute. Other Boolean operators, such as OR, NOT or IIF (if and only if) are supported as well. The representation of the cloud environment is therefore very flexible.

Figure 5.6 shows how one component of an initial application model is transformed into a feature model. This example highlights SQLServer, which contains two explicit constraints. First, an implicit constraint is adhered by selecting the correct feature model, which again is based simply on matching the node type with the feature model name. Next, the explicit constraints available within the application model are injected into the feature model. A more complete example, including such constraints, was provided in the previous sub-section. In Figure 5.6 we see both the textual and diagram representation of these constraints. The implications, shown below the feature model diagram, match with the provided constraints and through that select the valid nodes and attributes.

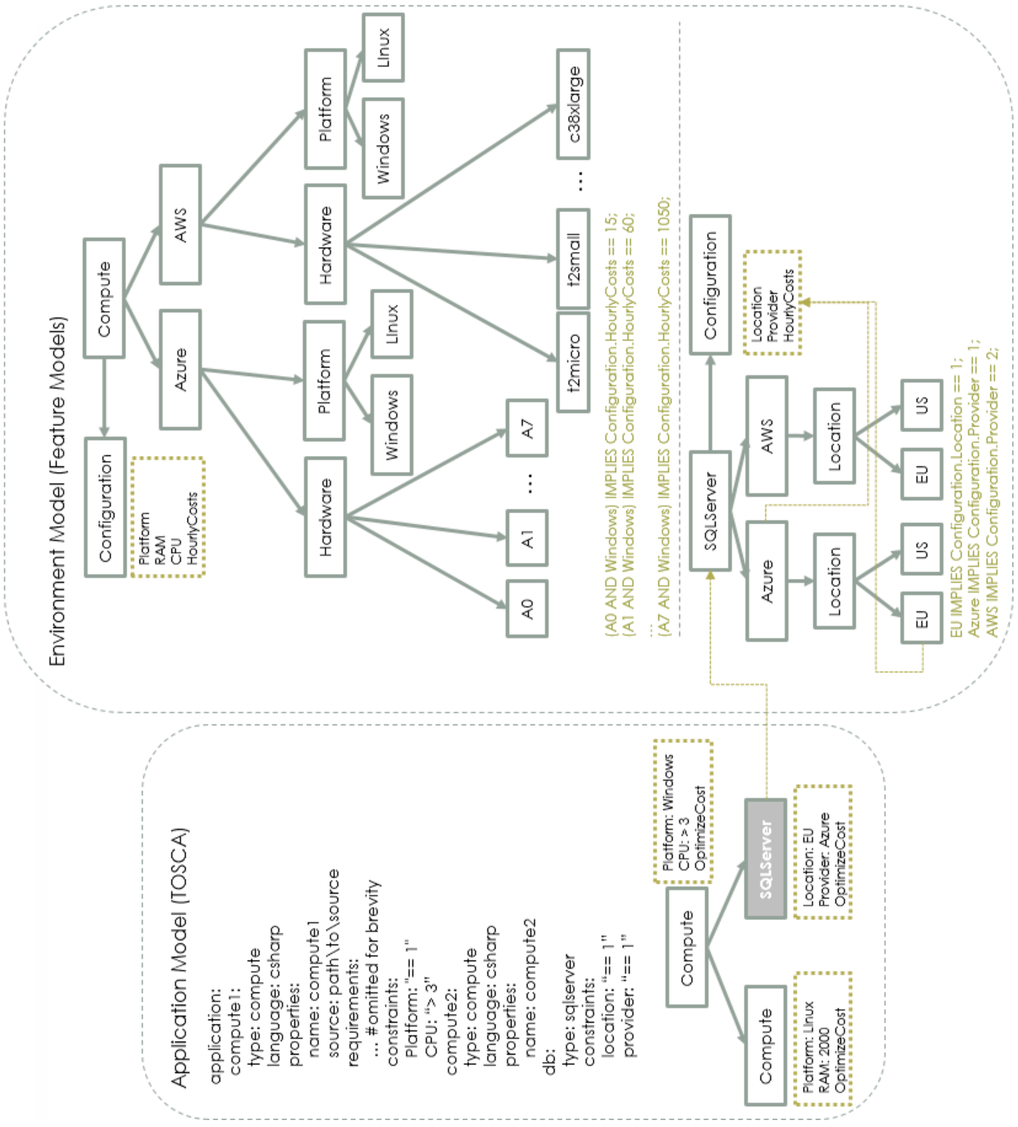
After the feature models for each component has been analyzed, we will most likely end up with a number of valid deployment scenarios. Provided that the SQLServer node needs to be deployed on Azure within Europe, each hardware configuration supported by Azure will still be valid, resulting in multiple solutions. Unfortunately, the used analyzer does not support an 'optimize' function, meaning we are unable to automatically select, for example, the cheapest scenario. Therefore, a number of custom product selectors have been written, that take each valid product and return, in this example, the scenario with the lowest cost. Figure 5.7 depicts the entire process.

Conflicting constraints may of course return no viable deployment scenarios at all. This may happen with faulty constrains (*at most* 1000MB and *at least* 2000MB RAM together is rather problematic) or with too optimistic constraints. Finding a MongoDB server for free with at least 5000MB storage is not going to return any results either. In the event this happens, the process simply stops: the selector returns the message that the provided constraints are too strict and need to be changed.

The final step is to generate a valid TOSCA node. After the product selector has selected one or more valid deployment scenarios[42], we need to transform the acquired information into a new text file. An example of such a file is Code Listing 5.4, which is a possible result of processing Code Listing 5.2.

Everything new or changed compared with the previous file is bold. Whereas the previous file contained a constraint `Location: "== 1"` we can now see a `location` property with the value `EU`. Luckily, the automated selector correctly interpreted the constraint and selected a deployment location within Europe. Some properties that are not required for deployment are also added, such as `hourlycosts`, `ram` and `cpu`. The product selector uses these properties to find out what the optimal scenario is, given the constraints.

[42] Some product selectors were written that might return multiple viable deployment scenarios. One such selector returns, for example, a range of different hardware configurations that can be tested for performance. This should allow questions such as "should I select two 'medium' or one 'large' hardware configuration" to be answered. The implementation of this idea is performed in another research and not discussed in this thesis.


Environment Model (Feature Models)
Compute
Configuration
Platform
RAM
CPU
HourlyCosts
Azure
AWS
Hardware
Platform
Windows
Linux
A0
A1
...
A7
t2micro
t2small
...
c38xlarge
(A0 AND Windows) IMPLIES Configuration.HourlyCosts == 15;
(A1 AND Windows) IMPLIES Configuration.HourlyCosts == 60;
...
(A7 AND Windows) IMPLIES Configuration.HourlyCosts == 1050;
SQLServer
Configuration
Location
Provider
HourlyCosts
Location
EU
US
EU IMPLIES Configuration.Location == 1;
Azure IMPLIES Configuration.Provider == 1;
AWS IMPLIES Configuration.Provider == 2;
Application Model (TOSCA)
Compute
Platform: Windows
CPU: > 3
OptimizeCost
SQLServer
Location: EU
Provider: Azure
OptimizeCost
Platform: Linux
RAM: 2000
OptimizeCost

```
application:
 compute1:
  type: compute
  language: csharp
  properties:
   name: compute1
   source: path\to\source
  requirements:
   ... #omitted for brevity
  constraints:
   Platform: "== 1"
   CPU: "> 3"
 compute2:
  type: compute
  language: csharp
  properties:
   name: compute2
db:
 type: sqlserver
 constraints:
  location: "== 1"
  provider: "== 1"
```

Figure 5.6: Selecting a deployment scenario for the SQL Server Node. The SQL Server Feature Model is loaded. The configuration attributes now need to match with the application component constraints. Selecting the Azure provider will set the Configuration 'provider' attribute to '1'. Selecting EU for either Azure or Amazon will set the Configuration 'Location' attribute to '1'. Now, Azure + EU will match the component constraints. The 'OptimizeCost' constraint is adhered by selecting the cheapest hardware configuration (not modeled in this example, and is performed after initial analysis).

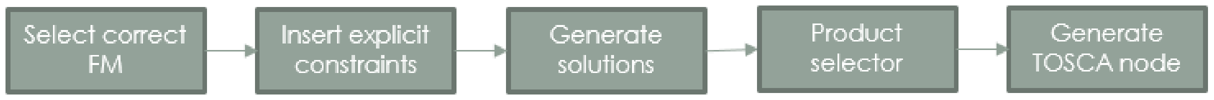


Figure 5.7: An overview of the steps within the automated selection.

```
benchmark:
      type: ortec.nodes.aws.compute
      language: csharp
      properties:
        name: ORTEC.Benchmark.WebUI
        source: path\to\source
        location: EU
        hardware: t2micro
        hourlycosts: 13
        ram: 1000
        cpu: 1
      requirements:
        database_endpoint:
          node: db
          relationship: ortec.relationships.csharp_to_mongo
    worker:
      type: ortec.nodes.azure.worker
      language: csharp
      properties:
        name: Worker1
        source: path\to\source
        location: EU
        hardware: A0
        hourlycosts: 15
      requirements:
        database_endpoint:
          node: db
          relationship: ortec.relationships.csharp_to_mongo
    db:
      type: ortec.nodes.heroku.mongo
      properties:
        location: US
        hardware: Sandbox
        hourlycosts: 0
```

Code Listing 5.4: A TOSCA file ready for deployment. The deployment scenario generator has inserted the bold text.

The `type` of each node has also been modified to inject the selected provider on which to deploy. Where earlier the benchmark node was of type `compute`, its new type is now `ortec.nodes.aws.compute`. In other words, the node will be deployed to Amazon AWS. The former two properties (`ortec` and `nodes`) exist for name spacing as to avoid any name collisions with other vendors who create nodes. This is part of the TOSCA specification. As was described earlier, in our implementation this node does not *require* an AWS PaaS Compute Service, but *is* an AWS PaaS Compute Service. Through inheritance by type, the correct scripts will be executed to properly deploy this component on Amazon AWS, which we will see in the next sub-section.

5.4.3 Automated Deployment

The next step as presented in Figure 5.1 is the automated deployment of a deployment scenario. The previous sub-section showed us how a deployment is generated, which serves as the single input for this next step. The previous chapter shows the distinction between stacks, CSBs, and orchestrators to tackle the portability and interoperability issues within the cloud environment. As the orchestrator proved to be the most flexible option, an orchestrator is implemented that can potentially use any CSB or stack that already exists.

Overall, the orchestrator is able to parse and analyze the given TOSCA deployment scenario, after which it executes stand-alone scripts in the correct order to perform the actual deployment of each component. Initially, the attempt was made to use Cloudify. The previous chapter has shown that its TOSCA-like blueprints and built-in support for a number of stacks align well with the goals of our automated deployment. Unfortunately, using Cloudify proved to be problematic. Default applications provided for testing material more often than not were unable to be deployed. Installations got stuck and provided either none or unhelpful debugging information. A lack of resources on the internet, such as on Stack Overflow and the official forums, did not give us the means to overcome these problems.

The decision was therefore made to build our own TOSCA parser. As we do not require all functionalities defined within TOSCA, only a subset of the specification is supported. The next chapter dives further into the actual implementation of the method.

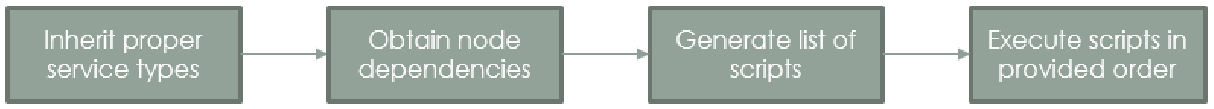

Inherit proper service types
Obtain node dependencies
Generate list of scripts
Execute scripts in provided order

Figure 5.8: An overview of the steps within the automated deployment

Four different steps, listed in Figure 5.8, are taken during the automated deployment. First, cloud service information is retrieved by inheriting the proper TOSCA service. Section 3.4 already explained the concept of inheritance within TOSCA. As an example, Code Listing 5.5 displays the `ortec.nodes.aws.compute` node type.

```
ortec.nodes.aws.compute:
  interfaces:
    create: aws.compute.deploy.ps1
    destroy: aws.compute.destroy.ps1
```

Code Listing 5.5: An example of a TOSCA node type.

Two different scripts are inherited. The `create` and `destroy` interfaces are what TOSCA defines as 'lifecycle operations'. Such operations can be triggered by an orchestrator; this prototype only supports the `create` and `destroy` operations. The `create` script is executed when deploying a component. The `destroy` script is executed when destroying an existing deployment of this node type. Both scripts in the example are Windows PowerShell scripts[43], currently the only type of scripting supported by the prototype. In theory, anything can be executed, ranging from scripts in Python, Ruby, or JavaScript to entire executables. No other information about the AWS compute type is modeled as all other information is contained within the feature model.

[43] https://technet.microsoft.com/en-us/scriptcenter/powershell.aspx

A relationship between two components will also contain scripts. An example of a relationship between a C# and sql server node is depicted in Code Listing 5.6.

```
ortec.relationships.csharp_to_sql:
  interfaces:
    post_configure_target: csharp.sql.setconnectionstring
    post_configure_source: sql.whitelistinboundip
```

Code Listing 5.6: An example of a TOSCA relationship type.

The first script is executed after the target node of the relationship, which is the node that is referenced to in a relationship (if a `compute` node requires a `db` node, the `db` node is the target). We see that after a `db` node is configured (deployment + any other scripts), a `setconnectionstring` script is run. This will inject the database connection string into the application so it is able to communicate with the database. Next, after the `compute` instance has been configured, a `whitelistinboundip` script is executed. This will whitelist the now-known ip-address of the `compute` instance at the `db`. This is a sql server specific requirement, which because of TOSCA's type-template inheritance system we only need to define once.

The second step of automated deployment is to obtain the dependencies of each node, and use these to generate the correct steps in which to execute the scripts of each node. If a compute instance requires a database, the database scripts are executed first. If a compute instance requires two databases, the scripts of both databases are executed in parallel (assuming no other dependencies exist).

Third, having acquired all scripts and dependencies, we can generate the correct order in which to execute all scripts. Figure 5.9 shows in what order the scripts are executed. Each script is optional, and each script can be overridden. The difference between *configure* and *deploy* in Figure 5.1 is now clearly visible as well. A node can be configured before and after deployment if it is part of one or more relationships. The example of injecting a database connection string into a compute node after the database is deployed is a much occurring scenario.

Finally, with the correct order of scripts to execute known, these scripts can be executed. During runtime, the node scripts receive the properties of its respective node. Relationship scripts receive the properties from both source and target node. All scripts are able to add properties to its received nodes, something displayed more clearly in Figure 5.10. Here, we see how initial sets of properties for two nodes are supplemented as the scripts are executed. After the first script, the database its endpoint is known which is then used by the second script to be injected in the compute node's settings. Scripts three and four show a similar use case with the compute node its ip-address.

After deployment, the `destroy` operation can be executed to remove all installed components. In addition, would a deployment fail due to an unforeseen error, the `destroy` operation is called for each node that is already deployed.

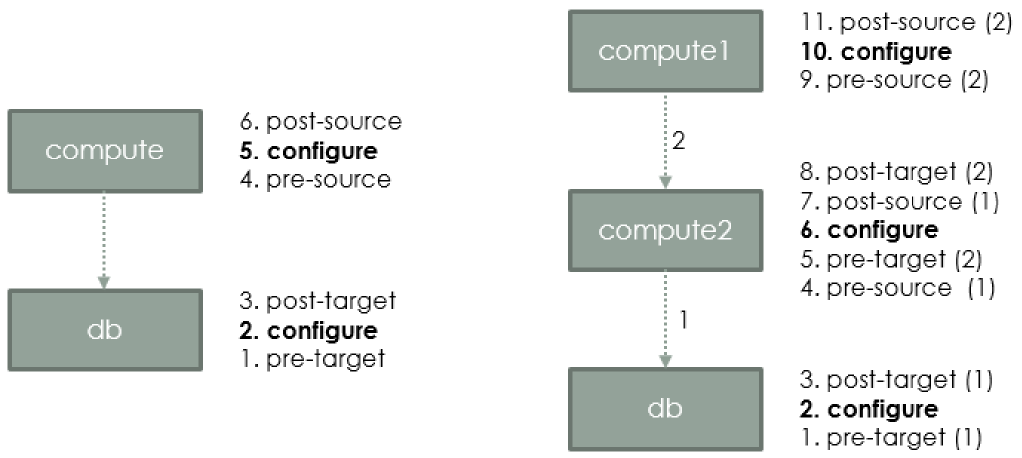


Figure 5.9: Two examples of generating the proper execution of scripts. The bold script are the actual deployment scripts (named 'configure' by TOSCA). The other scripts are relationship-scripts. The left example has only one relationship, which can in total contain four scripts. The right side is a little more complex as one node is both a source and a target. The numbers after the scripts denote which relationship this script is for, as mentioned by the numbers beside the dependency lines.

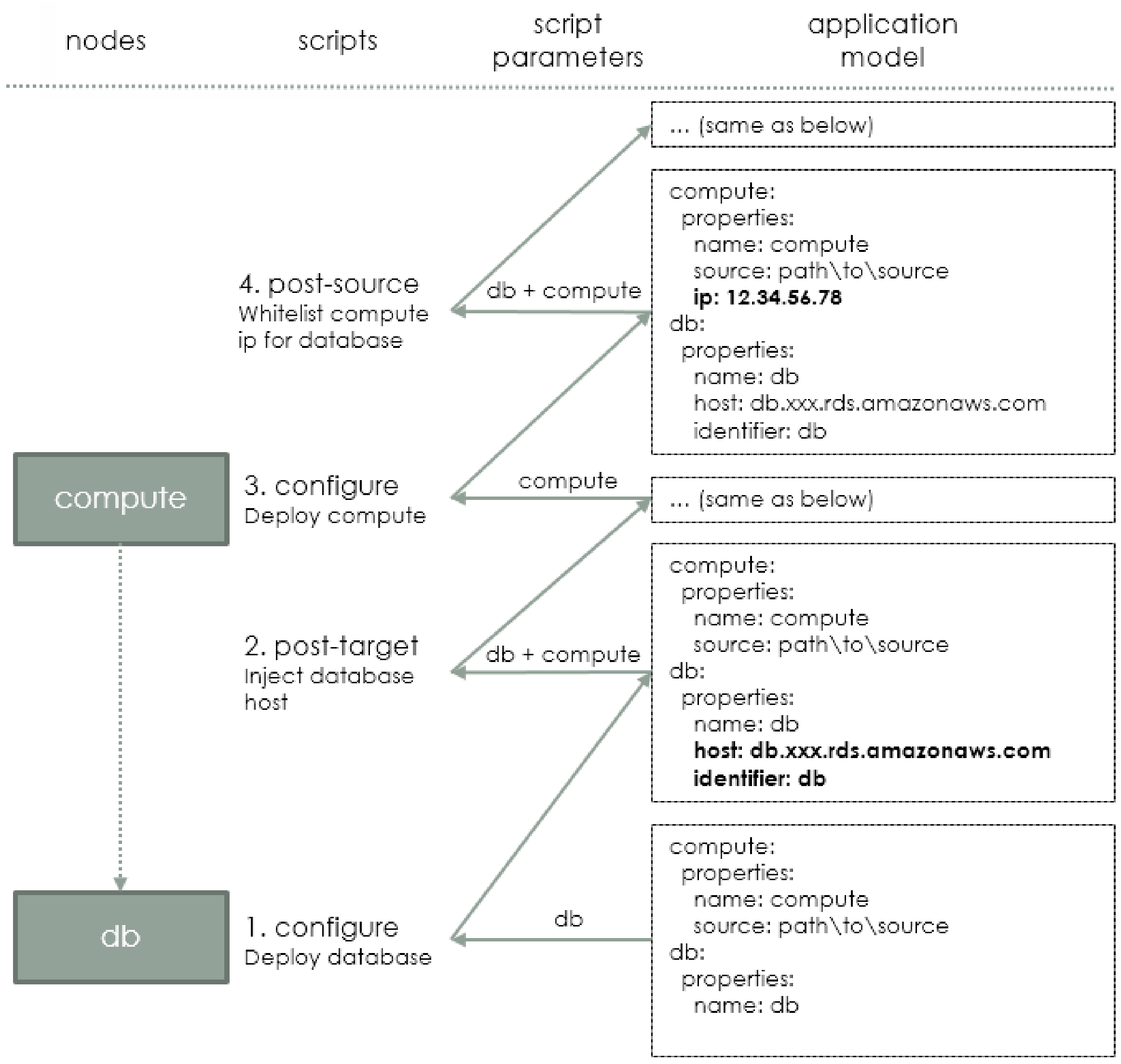


Figure 5.10: The process of deploying a compute and database node. Four scripts are executed and the diagram shows what these scripts do. The right side shows the changing state of the TOSCA file's node properties. All other details concerning these nodes have been left out for brevity. The bolded new properties are added after the deploy scripts have executed. The arrows pointing left show which properties are injected into the script – a relationship script receives the properties for both nodes. In this situation, the relationship scripts do not add any properties, leaving the file unchanged.

5.5 Discussion

This chapter has shown the workings of a method which aids in the (simultaneous) use of different cloud providers. In this section, we highlight the differences with existing methods, and analyze to what extent the used formats – TOSCA and feature models – are suited for the context of our research.

5.5.1 Distinctiveness

In section 4.1.1 we presented an overview of current research that uses feature models for the automated selection of a cloud deployment scenario. We again present this table below in Table 5.1, now with the addition of our own method.

Publication	**FM strategy**	**Number of providers**	**Service level**	**Automated Deployment**	**Used parameters**
(Garcıa-Galán et al., 2013)	Multiple	1	IaaS	No	Cost, Capacity
(Quinton et al., 2013a)	Multiple	4	PaaS	No	Capacity
(Wittern et al., 2012)	Multiple	3	SaaS	No	Cost, Capacity
(Cavalcante et al., 2012)	Single	2	PaaS	No	Cost
This	Multiple	4 (1 local)	PaaS	Yes	Cost, Capacity, Location, Provider

Table 5.1: A copy of Table 4.1 with the addition of our own method.

Similar to most existing methods, we use the 'multiple' feature model strategy, meaning that we utilize multiple feature models instead of only one. In each of the existing strategies that uses multiple feature models, the cloud provider is placed on top. The research by (Cavalcante et al., 2012) comes close to our approach. Its feature model for a specific application is shown in Figure 5.11. Similar to our research, its cloud services are placed above the cloud providers. They did however, place a node on top of these cloud services, something we decided not to do as discussed in section 5.4.1 and described in Figure 5.3. Unfortunately, their paper does not go into any detail about the reasons and repercussions of this design decision. Their subtrees for the cloud services do not contain any inter-dependencies. In section 5.4.1 we already explained the rationale behind our design approach.

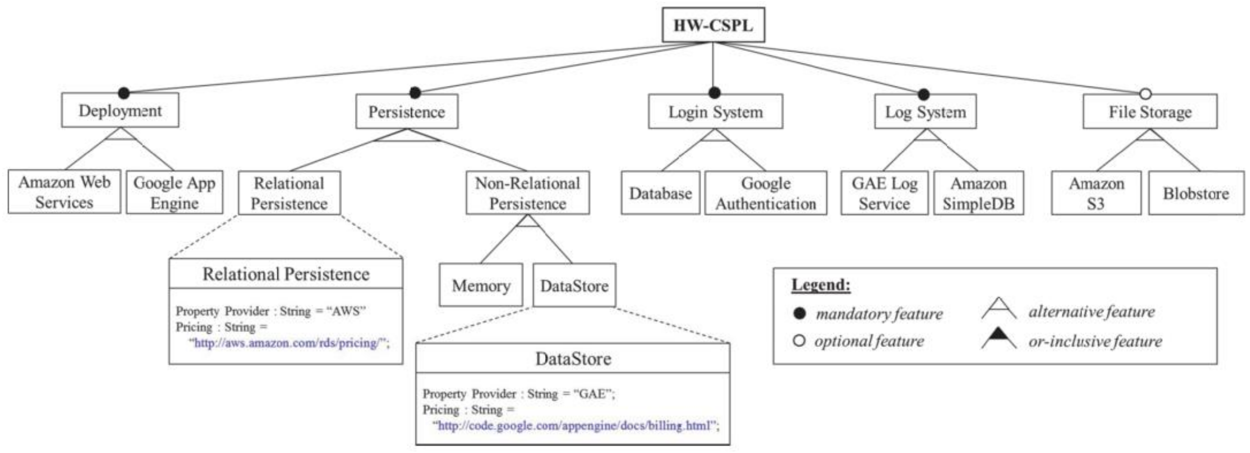


Figure 5.11: A feature model taken from (Cavalcante et al., 2012). This feature model is similar to our approach as cloud services are placed above the cloud providers.

Second, this is the first method that provides the geographical location of the deployment as a parameter. This slightly increased the complexity of our cost calculator, as costs often vary based on the selected location. As we only included the East-US and West-Europe locations within our feature models, we did not come across this specific situation. However, the next sub-section will show that modeling this in the feature model is not ideal and can be considered a limitation of the feature models.

Finally, this method is the first to supplement automated selection with automated deployment. No other research has actually processed the deployment scenario's that were generated by analyzing the feature models. This fact tells us that we are the first who can give a perspective on the integration of feature models with a method for automated deployment – in our case, TOSCA.

5.5.2 Feature models & TOSCA

We have seen that feature models can potentially contain a lot of information. With our set of supported variables within the feature models, we already grasped the boundaries of what is still somewhat practical to model. Code Listing 5.7 shows a small set of implications concerning the costs of the compute node type.

```
(AWS_Hardware_t2micro AND AWS_Platform_Windows) IMPLIES Configuration.HourlyCosts
== 18;
(AWS_Hardware_t2micro AND AWS_Platform_Linux) IMPLIES Configuration.HourlyCosts
== 13;
```

Code Listing 5.7: A set of implications that set the correct costs based on two variables: hardware and platform.

In this situation, the HourlyCosts of the compute node depends on two variables: the selected hardware configuration and the selected platform. The number of options given these two variables is the multiplication of the number of options for both. With seventeen hardware configurations and two supported platforms, this amounts to 34 different possible costs.

Would we now also add two geographical locations that actually have different costs, the total number of options would immediately rise to 68 (17 * 2 * 2; exponential growth). The new implications within the feature model will look as in Code Listing 5.8.

```
(AWS_Hardware_t2micro AND AWS_Platform_Windows AND AWS_Location_Europe) IMPLIES
Configuration.HourlyCosts == x1;
(AWS_Hardware_t2micro AND AWS_Platform_Linux AND AWS_Location_Europe) IMPLIES
Configuration.HourlyCosts == x2;
(AWS_Hardware_t2micro AND AWS_Platform_Windows AND AWS_Location_China) IMPLIES
Configuration.HourlyCosts == x3;
(AWS_Hardware_t2micro AND AWS_Platform_Linux AND AWS_Location_China) IMPLIES
Configuration.HourlyCosts == x4;
```

Code Listing 5.8: A new set of implications, given that the costs are also based on a new variable: geographical deployment location.

Again, the feature model is perfectly capable of modeling this information. The problem is however that modeling this is very error-prone. One solution is to use a different modeling format that supports a multi-dimensional solution space through an easier to manage notation. This new format can then automatically be converted to the above format, which is required by the FaMa analyzer. Unknown however are any performance costs to FaMa when adding so much extra implications.

Moving on from feature models, TOSCA is a solid standard for describing both an application and its underlying environment and infrastructure. It contains many concepts that are required to describe this domain such as nodes, relationships, and related lifecycle scripts. Many definitions for specific nodes and relationships are already included in the standards, including for example the concept of load balancers, floating IP's, databases and virtual machine's. This does not mean however that such definitions are 'ready to use'. For example, the database definition looks as in Code Listing 5.9.

```
tosca.nodes.Database:
  derived_from: tosca.nodes.Root
  properties:
    name:
      type: string
    port:
      type: integer
    user:
      type: string
      required: false
    password:
      type: string
      required: false
  requirements:
    - host:
        capability: tosca.capabilities.Container
        node: tosca.nodes.DBMS
        relationship: tosca.relationships.HostedOn
  capabilities:
    database_endpoint:
      type: tosca.capabilities.Endpoint.Database
```

Code Listing 5.9: The database node type as specified by TOSCA.

This example shows that a database will require a name, port, username, and password to be instantiated. In addition, it shows that the user and password are optional. Next, the specification shows that a database requires a database management system on which it needs to be hosted, and that a database has the 'database' capability.

Given the TOSCA standard and its concepts like properties, requirements, and capabilities, everyone who knows how a database works can create a Database type node such as the above. Within the context of our research, the following two items were the actual challenges related to TOSCA;

- *Generating TOSCA*. Based on the information modeled in the pre-TOSCA application model and the environment captured in feature models, a viable TOSCA file had to be generated. This was discussed in section 5.4.2. Though we took some shortcuts to ease the integration between the two standards (in particular, the modeling of the constraints), generating a viable TOSCA file is perfectly possible.
- *Executing scripts*. Based on the generated TOSCA file, the proper scripts had to be executed. This is where TOSCA has shown its true strength, as the application's model provides a clear to follow route for proper deployment of an entire application. Looking at the above database example that 'requires' a database host, we did make the decision to merge these two nodes into a single one to decrease the complexity of the application' model. This is still within bounds of the TOSCA specification.

Chapter 6: **Case Study & Analysis**

This chapter discusses two case studies and the analysis of the implemented method. First, we describe the two applications used for testing the method in section 6.1. Next, we analyze the development of the prototype in section 6.2. In section 6.3, we discuss the prototype and its future potential with three software experts. In section 6.4, we discuss the findings within this chapter.

6.1 Applications

Two applications have been used during the development of the prototype.

6.1.1 Benchmark application

The first application is a closed-source benchmark application developed by ORTEC. An administrator is able to build questionnaires. Users, who require a login given to them by the administrator, are able to fill in the questionnaire. ORTEC developers then define how the inserted data by the users is used to build reports for the administrators.

The application knows three different components. First is the UI component, used by both users and administrators of the application. This component has been built in C# on the server side and AngularJS on the client side. Next is the worker component, a C# application that processes requests received from the UI that do not require immediate feedback. Third is the Mongo database component, a document-store database and therefore different from the more traditional SQL database. Both the UI and worker connect to the database and therefore depend on this component. The UI inserts actions that need to be performed by the worker in the database, such as e-mails that need to be sent. The worker uses a polling mechanism to check every few seconds if any new actions are inserted into the database.

The UI and worker components can easily run within Windows Azure. Deploying both applications can be done manually within the development environment for .NET, Visual Studio. There is however no support for MongoDB within Azure. Therefore, manually installing the benchmark application is done by setting up a MongoDB somewhere outside Azure, and then manually adding the connection string to both other components. Automating this process is already of interest and with our prototype, we can easily deploy the UI on Amazon AWS as well, or generate an installable of the worker for local installation.

6.1.2 Wordpress

Wordpress is one of the best known PHP applications. Recent reports show that 23.9% of *all* websites use Wordpress (W3Techs, 2015). Though Wordpress started as a blogging application in the beginning of 2004, today it is much more than that and can manage entire websites, including static pages and many media types. Wordpress is especially well known for its good usability and extensibility through themes and plugins, which is why many non-technical people use the software.

A Wordpress installation contains two different components. The first is a PHP application that present the UI of the application. This component connects to the second MySQL component. The PHP application therefore depends on the MySQL component.

The first application was developed mainly using the .NET application framework. Most applications developed within ORTEC are built using this framework. This prototype is built using .NET as well; a large part of the research context has been within a .NET ‘atmosphere’. Therefore the decision was made to add this second non-.NET use case to test the prototype.

6.2 Analysis

Both application have gone through the entire process of modeling, selection, configuration, and deployment. Here, we discuss both applications as they go through this process.

6.2.1 Benchmark application

Code Listing 6.1 shows the application model for the benchmark application. It is modeled in TOSCA with some added example explicit constraints.

```
benchmark:
  type: compute
  language: csharp
  properties:
    name: ORTEC.Benchmark.WebUI
    source: path/to/source
  requirements:
    database_endpoint:
      node: db
      relationship: ortec.relationships.csharp_to_mongo
  constraints:
    Provider: "== 2"
    CSharpVersion: "== 45"
    Location: "== 1"
    CPU: "> 2"
    RAM: "> 2000"
worker:
  type: worker
  language: csharp
  properties:
    name: Worker1
    source: path/to/source
  requirements:
    database_endpoint:
      node: db
      relationship: ortec.relationships.csharp_to_mongo
  constraints:
    Provider: "== 1"
    CSharpVersion: "== 45"
    Location: "== 1"
    RAM: ">= 1000"
    HourlyCosts: "< 300"
db:
  type: mongo
  constraints:
    Storage: "> 10000"
```

Code Listing 6.1: The modeled benchmark application.

The constraints tell us that the `benchmark` node will be deployed on AWS. The `worker` node will be deployed on Azure. The `db` node will be deployed on the cheapest option, though we only support one MongoDB provider, Heroku. Both the `benchmark` and `worker` node will be deployed in Europe. Also, both these nodes require the C# 4.5 runtime. Would not all cloud providers support this run-time but for example, only older versions, this constraint would ensure the application to be deployed on a compatible run-time.

We also set some hardware restrictions, with constraints for `RAM`, `CPU` and `Storage`. Finally, we set a `HourlyCosts` constraint for the `worker` node.

Based on the above file, we generate a deployable TOSCA-file. In Code Listing 6.2 we see the outcome.

```
benchmark:
  type: ortec.nodes.aws.compute
  language: csharp
  properties:
    name: ORTEC.Benchmark.WebUI
    source: path/to/source
    location: EU
    hardware: c3xlarge
    hourlycosts: 210
    ram: 7500
    cpu: 4
  requirements:
    database_endpoint:
      node: db
      relationship: ortec.relationships.csharp_to_mongo
worker:
  type: ortec.nodes.azure.worker
  language: csharp
  properties:
    name: Worker1
    source: path/to/source
    location: EU
    hardware: A1
    hourlycosts: 60
  requirements:
    database_endpoint:
      node: db
      relationship: ortec.relationships.csharp_to_mongo
db:
  type: ortec.nodes.heroku.mongo
  properties:
    location: US
    hardware: M1
     hourlycosts: 3333
```

Code Listing 6.2: A deployment scenario for the benchmark application. Generated given the input in Code Listing 6.1. Bold text denotes the new, automatically selected information.

Everything that is different compared with the previous file is bold. We can see that each of the constraints are adhered. Every node has received a valid type that will provide us with the correct deployment scripts through inheritance. First, with the MongoDB node being a requirement for both other nodes, this node will be deployed first. After, both other nodes will be deployed in parallel. This parallel deployment is considerable faster than sequential deployment, as both deployments take around 10 minutes.

A notable different outcome would have been generated when constraining the `Provider` to `10` for the `worker` role. `10` denotes a local deployment, meaning we would like to install this node locally as a windows service. The next step – automated deployment – would then not deploy the worker role on a cloud provider, but instead generate an installable *.msi file. Similar with a cloud deployment, this file would correctly be able to connect to the MongoDB.

Finally, after deployment, the deployment tool generates a new TOSCA file, again with new information. Instead of again displaying the entire file, we only show the additions to the node's properties in Code Listing 6.3.

```
benchmark:
  properties:
    ip: 54.76.174.177
db:
  properties:
    host: mongodb://heroku_appxxx:yyy@zzz.mongolab.com:29051/heroku_appxxx
    name: quiet-peak-7004
```

Code Listing 6.3: The new properties generated during the deployment of the benchmark application.

The first thing to notice is that the `worker` role did not receive any new properties. This is because this `worker` role did not receive an endpoint as this node is not being accessed by the other nodes. The MongoDB node received two properties. The `host` property is used by the other nodes to connect to the database. The `name` property is required to be able to destroy the node later. As we did not provide a name for the database, Heroku automatically chooses this.

6.2.2 Wordpress

The main reason for including the Wordpress application was to test the extensibility of the prototype. Before it was possible to deploy a PHP application, only .NET components were supported. Therefore, besides describing the workings of automatically deploying Wordpress, we also discuss the ease with which we were able to add support for the required components to the framework.

The first step was to extend the compute feature model with the PHP runtime. We support PHP on both Azure and AWS. Figure 6.1 shows all nodes that were added to the existing feature model in bold. Many of the deeper nodes are not shown for brevity.

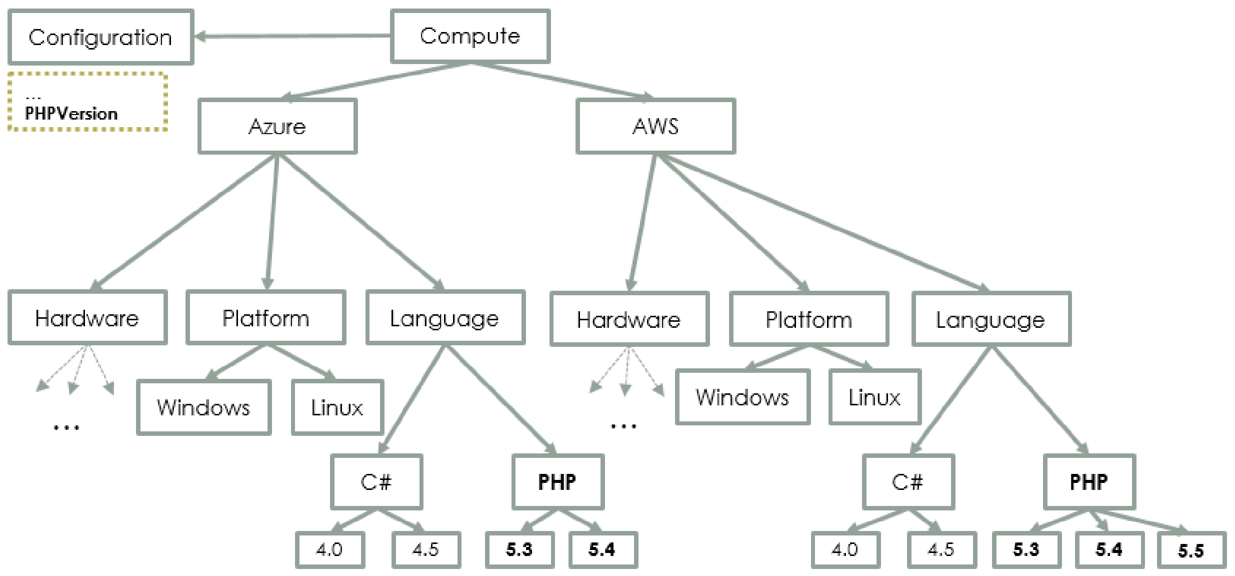


Figure 6.1: A subset of the compute feature model. Required additions for adding PHP support are bold. We can see seven new nodes and the Configuration Interface received a new attribute.

As we can see, the changes are relatively simple. No existing nodes needed to be changed; only additions were required. Of course, would we not have the 'Language' node for both providers, but instead directly a 'C#' node, the changes would have been bigger. It is therefore important to think ahead when constructing the feature model, and properly model it with future changes in mind.

The second step was to add MySQL support to the feature models. This is a completely different node type, thus a new feature model was constructed. Again, no existing nodes or in this case, entire feature models needed to be changed to add this component type. Most of the work went into translating the cloud environment characteristics, such as the AWS MySQL pricing information[44], into the feature model. For compatibility with the deployment scenario generator, the feature model is named 'mysql', now implicitly connected to a `nodes.ortec.[provider].mysql` TOSCA node type. Therefore, a third step is to create this node type, which is shown in Code Listing 6.4.

```
ortec.nodes.aws.mysql:
  properties: {}
  interfaces:
    create: aws.mysql.deploy
    destroy: aws.mysql.destroy
```

Code Listing 6.4: The new MySQL node type. This node type can now be inherited, and MySQL components can be deployed on Amazon AWS.

These changes allowed us to properly generate a deployment scenario. The fourth and final step is to add the correct scripts to support deploying a MySQL node, and destroying it. As a separate script file exists for each provider and node type combination, a new script is added for each provider that will support MySQL. None of the existing scripts needs to be altered.

[44] http://aws.amazon.com/rds/mysql/pricing/

6.3 Expert Validation

As defined in the research method, we conduct expert reviews to validate the workings of the prototype. Besides the practical advantages of using the prototype compared with the traditional process, we also discuss the two other subjects. First, discussions were held about the scope and assumptions made related to the prototype. We consider the feasibility of abstracting the heterogeneous PaaS environment into a commercial product. Second, we consider the potential of the prototype.

Three informal interviews were held with three experienced professionals from ORTEC. Each interviewee had a different position within the company, including a manager of technology and innovation, a software division manager, and a software engineer. Each participant had at least 7 years of notable IT experience. Discussion concerned business, practical and technological viewpoints.

Six questions were developed beforehand to give the interview a clear focus and a solid starting point from which to begin discussions. Each question is discussed separately. Quotes have been directly translated from Dutch, and thus will differ from the exact wordings used during the interview.

1. Would you consider PaaS 'more important' than IaaS?

Each of the interviewees agrees that PaaS knows many benefits compared with IaaS. Most emphasis is placed on the difference in maintenance. With PaaS, only the application needs to be maintained whereas with IaaS, the underlying operating system also needs to be kept up to date. This makes software development much more expensive. One interviewee mentions that "the *plumbing required for IaaS is too much of a hassle; VM's are no longer of this time".* A similar viewpoint is that *"if you're buying a service on which to deploy your software; then why not pay for maintenance while you're at it as well*".

It also very much depends on the core business of your company. Software development is very different from server maintenance, so if this latter is not your core business, you may want to outsource this to a cloud provider and use PaaS. As mentioned during the interviews, "*as a consulting company it is our business to build software as efficiently as possible. This does not include resources to maintain virtual machines"*.

One notable disadvantage of PaaS is that you are limited to the technology that is made available by the cloud provider. For example, the PaaS provider may not immediately support a new major version for a new programming language, whereas on IaaS, the cloud user installs this new version. When asked if this is indeed a practical limitation of PaaS, one interviewee noted that this scenario is "*seldom witnessed in real life*". Another interviewee notes that in this situation, "*a switch needs to be made to IaaS*". A similarly noted situation is that of legacy software, which may require old technology that is not supported by PaaS providers. *"Rewriting legacy software so it will work on PaaS is much more work than maintaining the infrastructure to keep it running"*.

Especially new projects are very much suited for PaaS. *"When a client comes to us with the request of building a new application, or extending an existing application, we will always build this on PaaS if all* *other constraints also allow us*". More suited for IaaS is high-performance software, as IaaS "*allows you to tweak much more on the OS level to optimize this specific application for the underlying OS and infrastructure*".

2. Why do you think not many PaaS brokers yet exist?

All interviewees question the feasibility of commercializing a PaaS broker or orchestrator. Different reasons are mentioned. *"Companies are not yet aware that [vendor lock-in] is a problem, and therefore that there may also be a solution"*, is one such reason. This argument is very similar to a famous quote by Steve Jobs: "*A lot of times, people don't know what they want until you show it to them"* (BusinessWeek, 1998). One reason for this argument may be that the market is still very young. Indeed, as another interviewee notes, "*the PaaS cloud is still in its infancy, and many are still too scared to use the cloud for many use cases*". This infancy is also a reason for the liquidity of the cloud. Because this domain is still very new, it constantly changes, as we already concluded in an earlier chapter.

A second argument is that initial development and maintenance of such a tool will be a lot of work, and therefore expensive to use. "*The question is whether people are interested in using a tool that will provide so many features, when they need only a few of those features*". *"Keeping your scope small is important, otherwise it becomes too expensive to both build and maintain the broker. The upkeep is huge*". The current prototype supports only a small set of cloud providers and services. Because of this, many applications will not be able to be deployed as a cloud service may be missing.

The most noted argument is that the cloud providers have no incentive to support such a tool. "*The cloud providers want vendor lock-in. They want their customers to stay*". This argument is also very much linked to the problem of maintaining the cloud environment model as the cloud providers do not expose their cloud services, prices, etc. through an API.

3. What features are missing from the prototype and how important are these?

One specific feature that all interviewees unanimously agree on to be missing is re-deployment. "*Fastening deployment time is great, but I will also need to update the application. Updating is done more often than initial deployment*". The prototype currently only supports initial deployment and destruction. Of course these combined can successfully 're-deploy' a compute service, but this will not work with databases as data will then be lost. In addition, this will be slower than actual re-deployment.

A second feature request is monitoring. *"How cool would it be if an application is pro-actively updated, based on some change in the cloud environment*". Also mentioned features are improved error handling (*"when something goes wrong, I want all information possible to see what is going on"*), improved rollbacks, and improved validation before and after deployment. As is noted by one interviewee: "*After deploying an application, how do I know for sure it is working and not only displaying an error page?*"

Another point that became clear during discussion of this question is that none of the interviewees is actually interested in automated selection. *"I can't think of an existing or previous client where automated selection would have been of use*". Thus, at least at ORTEC, only the automated deployment part of the prototype is considered of interest for future use.

4. To what extent do you believe the process [that this prototype implements] should be automated? Are there any actions you believe should continue to be performed manually?

This prototype supports the modeling, selection and deployment of a multi-component application. Modeling is currently done manually, whereas selection and deployment is successfully automated. All interviewees agree that the modeling of the cloud environment should be automated, as manually updating prices is very error-prone and is easy to be forgotten.

Two different opinions exist concerning the automation of the selection step. As the software engineer notes, "*Ideally, I'd like to see everything automated. Only if someone is wrong or unexpected, I'd want to be notified*". On the other hand, both other interviewees have a different viewpoint. *"I'd still want to make the decision myself; the automated selection may facilitate me in making a better decision"*.

There is more interest however for automated deployment. Of course, every software development these days has at least some form of continuous deployment in place. ORTEC is no different. *"The automated deployment should be integrated in our current DevOps processes"*. In fact, initial attempts were made to integrate both automated selection and deployment within TeamCity[45], a tool used for continuous integration of the software release lifecycle process. Due to time constraints, this project was unfortunately not finished.

5. What major roadblocks do you see in the near future when continuing to develop this prototype?

Especially maintainability is noted as a challenge. As the cloud environment changes quite often it will not be easy to keep supporting all features when cloud providers add, remove or alter services. The many differences between PaaS implementations is another challenge, and an interesting discussion is if we are abstracting away only apples, or perhaps different kinds of fruits. Are PaaS implementations enough alike to be put into a single model?

Compared with local development or IaaS deployment, PaaS can be a challenge to debug. Often many log files or error information may not be obtainable when using PaaS. When developing for a single PaaS provider, its ok to put time in this in figure out how exactly this provider works. When using a tool that abstracts away PaaS providers and a deployment fails or something goes wrong after deployment, it is very hard to debug this when not familiar enough with this PaaS provider.

6. Do you believe this prototype can deploy an application better (e.g., faster, more efficient) than the traditional method?

The interviewees are slightly optimistic about improved performance of the prototype compared with traditional means. Yes, now that the prototype supports both Azure and Amazon, applications deployed to those providers can be deployed more easily in an automated matter. Especially because different types of cloud services (e.g. compute and database) can be deployed. However, as was mentioned before, they are skeptical about further support for other cloud providers. In part because ORTEC will probably not use these providers, but also because adding more providers adds the burden of keeping more providers up to date.

[45] https://www.jetbrains.com/teamcity/

One problem the software engineer sees is how the prototype can be used. As this is still a prototype, no user-friendly UI has yet been developed (which they both understand). However, before actively using the prototype in production, it should be made more user friendly.

A last caveat mentioned is that the modeling of the cloud environment has taken some time. Currently this information is stored in a notation that is not very user friendly. Here as well there is a great need for a UI that eases the modeling of the cloud environment, the application, and the means to add constraints for automated selection and deployment.

6.4 Retrospect

Both the benchmark application and Wordpress have been successfully deployed using our prototype. Altering constraints may lead to a different deployment scenario, which is correctly executed by the automated deployment phase. We did come across a number of interesting practicalities that we will discuss here.

First, a seemingly minor difference between Azure and AWS leads to an interesting discussion. When deploying a compute component, it needs to be provided with a name. This name is then used to build the CNAME for the component (if it will have any); e.g., it's URL, such as *myapplication*.cloudapp.net for Azure or *myapplication*.elasticbeanstalk.com for Amazon AWS. AWS does not accept any dots in this name, and will return an error when dots are contained within the name. Azure does accept dots though – after successfully deploying an application on Azure may therefore unexpectedly fail on AWS, simply because of a dot. A similar situation is when deploying a sql server. AWS only allows the database name to be between 1 and 15 characters, whereas Azure allows many more characters. Again, deploying a database on Azure will be a success, but deploying the same component on AWS will fail simply because the name is too long. Multiple solutions for both these issues exists, each with their own problems though.

- Even when deploying a component to Azure, we can check if deploying this component to another provider such as AWS would generate any problems. If it does, we cancel deployment. This way, we wouldn't run into any surprises when deploying the component to another cloud provider at a later stage. This is not ideal though, as it's rather unartful to halt a deployment which will successfully deploy within the generated deployment scenario. What if the user specifically wants to deploy to Azure, and is limited by a constraint posed by another provider?
- We could automatically alter the component name to one that fits the requested or perhaps even all cloud providers. Altering such information behind the scenes can also be considered undesirable. A user may have a good reason to use this specific name and does not want it to change. If it needs to change, the user may want to do so as well.
- Transparency would ideally be the best solution: informing the user of any current or future incompatibilities with a cloud provider. This would increase the complexity of a seemingly unimportant issue though. Should the user be able to give a different name for each cloud provider – each adhering to the name constraints imposed by that specific provider? Should the

user be able to force a correct name for Azure, and let the software decide on a better name when deploying the component to another provider in the future?

A second interesting issue appeared when waiting for deployment to finish. Initially it seemed that Azure was able to deploy much more quickly compared with AWS. When a script executes the function that deploys the component – this function is part of the software development kit provided by a cloud provider – the script typically 'hangs' until deployment is 'finished'. Finished in this sense has different definitions though. For example, deploying a compute instance on Azure is 'finished' when the underlying virtual machine has been created and an ip-address has been assigned. At this time, in many cases we can safely continue deploying any next, depending component. The compute instance isn't actually working yet though at this point: the virtual machine is now booting and the application still needs to be installed and started. Typically it still takes around 10 minutes until the application is actually working from this point.

If a depending component only needs the ip-address of this component, this is OK. However if the depending component needs the previous component to be actually running it may become problematic. A question that arises is where we need to add the functionality that 'waits' until the first component is truly up and running. Is this a responsibility of the orchestrator or one of the component?

When defining the method we argued that the method needed to be both maintainable and extensible. Maintainability is important for the cloud environment, as we easily need to be able to modify its modeled information and modify the deployment scripts. The discussion of adding PHP and MySQL support to the prototype concluded that extending the model is easily achieved. However, the maintainability of the cloud environment model is far from ideal. Unfortunately, none of the supported cloud providers expose an API that we can use to automatically construct the environment model. Therefore there is no other way but to manually update the cloud environment. Finally, the information is stored in not very user-friendly text files. Therefore, the ease of changing this information can at least be improved through building a more user-friendly interface.

The scripts are more easily maintainable. Each lifecycle operation for a cloud service is defined in a separate script. Thus for example, deploying a compute role on Azure is defined within its own script file. The script files are named in a clear and consistent manner. Within these files, concerns are separated across different functions.

The same is true for the extensibility of the scripts. The clear separation of concerns and tasks makes it easy to add more functionalities. One caveat however is that when for example, adding a new cloud provider, a lot of knowledge is required about this cloud provider. Building the prototype required us to dive deep into the inner workings of both Azure and AWS. The Azure and AWS PowerShell development kits were studied extensively in order to be able to add the functionalities to the prototype. Of course, this alone is one assumption that led to the creation of this method: it would be ideal if only one interface needed to be studied, instead of a separate interface for each cloud provider.

The earlier chapters already introduced the notion of heterogeneity within the cloud environment. Each cloud provider is different, including different costs, interfaces, and different services that are provided. Of course, these differences go much further than differences in names, even for services that from the outside appear to be very much related. One such difference that is not easy to model is the option of replicating a sql server database across different locations. This is usually done to add redundancy in case an outage occurs at one location. In that case, another location can take over. For Amazon AWS, this setting can be turned either on or off. For Azure however, two different geo-replication options exist. The difference in both options lies in whether or not to make the redundant database readable. During a traffic peak database queries can then be routed to the database with the lowest current load. How to model this difference – and others – as constraints is interesting future research.

One such unexpected difference exists within the sql database provided by both Azure and Amazon AWS. Azure supports sql server, whereas AWS supports sql server, MySQL and others. Adding sql server support for both providers to the prototype may appears relatively simple at first. To use a sql server, all we need is an endpoint (a URL or ip-address), a username, and a password. However, getting access to that server requires us to set the proper security settings, such as adding the compute's node IP-address to a whitelist. Especially the security settings for Amazon AWS are complex to model. Included are security groups, Virtual Private Clouds (VPCs), subnets, and more. As TOSCA does not describe these concepts, we did not include these concepts in the topology but kept them within the scope of the scripts.

A final point to mention is the addition of a new product selector, the last step in the automated selection process as depicted in Figure 5.7. Currently the chosen product selector is the `LowestCostProductSelector`. However, when deploying a software application such as Wordpress that will very likely work on any PHP version, it would be of interest to select the latest PHP version, as well as the lowest cost. Both variables are independent, thus implementing this product selector will be a nice addition.

Chapter 7: **Discussion & Conclusion**

This thesis described the makings of an automated method that automatically selects and deploys an application within the heterogeneous cloud environment. The major goal was to learn more about the PaaS environment: how it can be modeled, how we can automatically choose between the options, and how we can seamlessly deploy an application on a set of different cloud providers. In addition, we added support for local deployment, where we create an executable that a user can run on its own machine.

In this chapter, we discuss the current results, recap our findings, and discuss potential future work.

7.1 Discussion

Though the method may seem perspicuous and solid on paper, through implementing it by means of a prototype some interesting issues became apparent. From the outside, cloud services from different cloud providers may seem very much alike, but within their implementations lie some less-obvious differences. These differences were already discussed in the previous chapter. Comparing sql server implementations of Azure and AWS mainly led to some practical inconsistencies that could be overcome. The issue of geo-replications options brought to light some bigger, conceptual differences. In addition, the Azure Cloud Services compared with Amazon Beanstalk contain differences in their configuration and settings.

The following limitations should be noted when considering the conclusions made that are based on the available context of this research.

- Only a small subset of the broad PaaS environment has been modeled and implemented within the prototype. Any conclusions made about the homo- and/or heterogeneity of the PaaS layer are based on this incomplete set of cloud providers and cloud services. Modeling and implementing may bring to light more issues that are of interest, which we do consider an important step towards a comprehensive PaaS ontology.

- The expert validation of the prototype was performed within a single company. The conclusions made in section 6.3 are therefore biased. Only little interest exists for using the automated selection step within our implemented process. Looking back at section 5.3 where we introduced two scenarios for using our prototype, ORTEC is therefore a company that is mostly interested in scenario number two. Interviews should therefore be held with companies that are interested in the first scenario, where the automated selection is also used.

Overall, the limitations teach us that a wider perspective is required. This research only grasps a small piece of what is to be an exciting new research direction.

7.2 Conclusion

This research was the first to implement the entire process of modeling, selecting, configuring, and deploying a software application within the heterogeneous PaaS cloud environment. Earlier research focused solely on modeling the cloud environment and adding automated selection capabilities. The difficulties of actually deploying the application were noted to be not without difficulty (Quinton et al., 2013a), but a concrete solution was never proposed. In addition, whereas most research is concerned with homogenization of the IaaS layer, this research was aimed towards the more heterogeneous PaaS layer.

Our method for complementing automated selection with automated deployment was achieved through adding the TOSCA standard for modeling the to-be-deployed software application. We modeled explicit constraints within the application model and were able to generate a TOSCA-compliant, deployable new application model. With the cloud environment modeled within feature models, we were able to simplify the TOSCA node types. Both TOSCA and feature models proved to be a successful method for modeling their respective domains.

Next, successfully performing the automated selection was achieved through the integration of feature models with TOSCA. Our deployment scenario generator transforms both implicit and explicit constraints modeled within TOSCA for use within the feature models. The used feature model analyzer, FaMa, provided us with a set of viable scenarios, out of which we select the cheapest one. Finally, we are able to generate a now fully TOSCA-compliant file that is ready for deployment.

Finally, automated deployment is achieved through the implementation of a cloud orchestrator. Our custom written TOSCA parser is able to successfully transform the received file into a set of to-be-executed scripts. Together, these scripts can not only deploy each component of a software application, but also wire them together. These scripts proved to be highly flexible and both extensible and maintainable.

Summarizing, we consider the following to be the main contributions of this research:

- A new and successful method is used to model the cloud environment within feature models. Using cloud services as the root of a feature model, we created a highly maintainable structure. We have also shown that it is unnecessary to create one large feature model, but that it is more practical to have a number of feature models and dynamically process the correct ones, given the components of the provided application.

- Feature models have been used to dynamically generate a valid, deployable TOSCA model of a software application. Earlier research already presented this as a missing feature within TOSCA (Brogi et al., 2014).

- While TOSCA is primarily being constructed for use within an IaaS environment, we have shown how we can use it for use within a PaaS context.

- A number of practical issues have been shown related to making a higher abstraction for different PaaS cloud services. These results should steer future work in a new direction, namely the specific modeling of cloud *variabilities* instead of *commonalities*, as we further discuss in section 7.2.2.

- Finally, existing research concerning automated cloud selection does not propagate its findings into actually deploying the application given the generated deployment scenario. This research tackles the full process of modeling, selecting, configuring, deploying, and even destruction.

We believe these findings are a solid basis on which to further research the homogenization of the PaaS cloud environment.

7.2.1 Research Questions

In Chapter 2 we defined a set of research questions that were explored in the later chapters. Here, we recap these research questions and provide a clear and concise answer to each of them.

SQ1: *"How can both the application and the environment be modeled to facilitate automated selection?"*

A literature review gave us a set of options for modeling both a software application and the cloud environment. TOSCA was especially suitable for modeling the application, with its focus on solving the portability and reusability problems within the cloud environment. Feature models, stemming from the highly variable software product lines paradigm, deemed very suitable for modeling the cloud environment. Existing research used different methods for modeling the cloud environment within feature models, though we decided to use a rather new approach that proved to be a success. Finally, combining these two approaches proved to be very fruitful. We were able to properly generate a valid TOSCA application model based on information retrieved from parsing the feature models.

SQ2: *"Which constraints can be modeled to automatically select a deployment scenario for software components?"*

The heterogeneity of the cloud environment contains many different approaches for similar objectives. Cloud services are alike, but different. Indeed, particulars about these services such as costs and hardware configuration differ much from one another. This even led us to the discussion of whether we are comparing apples with apples, or that perhaps different kinds of fruit are thrown into the basket. Finally though, we were able to make sound comparisons and therefore decisions, taking into account costs, hardware configurations and geographical locations.

SQ3: *"What methods exist to select a deployment scenario within the heterogeneous cloud environment including local deployment*?"

The synergy of both TOSCA and feature models came to its full potential in this phase. Both a technical and methodological challenge, all information from both models was combined into one or more viable deployment scenarios. Because the used feature model analyzer did not support finding an optimal solution, an extra tool was written that takes the cheapest configuration from each viable configuration returned by the analyzer. Using feature models, we were able to transform one, TOSCA-like application model into a deployable TOSCA file.

SQ4: *"How can the components of a software application automatically be configured and deployed?"*

Within but certainly also outside the cloud environment context, automated deployment is a hot topic. Many different mechanisms exist that promise to ease the deployment of software applications. The challenge within our research context was the setup a flexible environment that has the potential to deploy any software application to any cloud provider. Essentially, a TOSCA interpreter generates a set of scripts that are executed in a specific order. Components are deployed in the right order and relationship scripts wire these components together. Even the scripts layer is very flexible as potentially any script can be run to perform these actions. This flexibility gives us access to any library already written that supports any actions we need to execute.

Finally, the main research question had been defined as follows;

RQ: *"How can cloud selection, configuration, and deployment be fully automated?"*

Using the modeled application, the cloud environment and user-specified constraints, we are able to transform this information into an optimal deployment scenario. Next, using our own TOSCA-interpreter and a set of flexible scripts, we can use this deployment scenario to automatically deploy the provided application. The created method is easily extensible, highly maintainable and taught us a lot about the PaaS environment, something very much missing within the existing literature until now. A friendly user-interface would definitely ease the modeling of both the application and the cloud environment, as the current text-files used for processing are hard to maintain. Also, the modeled cloud environment is not easy to update and re-deployment is a missing and sought after functionality. We will later discuss these issues and label them as much needed future research.

7.2.2 Future Work

One challenge that became apparent during the modeling of the cloud environment was the amount of differences between seemingly alike cloud services. Implementations differ significantly and minor technicalities make the cloud environment even more heterogeneous than expected beforehand. Future research should embrace the idea that cloud providers seek to introduce functionalities not yet provided by their competitors. Instead of focusing on modeling the common denominator – something very much the focus within this research – a method should be constructed for modeling the variabilities between each cloud provider. Feature models have shown to be very flexible and can model a large variability of information. Future research should therefore definitely keep focus on this modeling technique.

Another topic worthy of investigation is the addition of local deployment to the solution space. In a sense, this is very incompatible with the PaaS environment, as deployment on a laptop or server is more similar to the IaaS layer. Though still very much an interesting topic, it may be more suited to combine this with an automated IaaS approach. As our method is already suited to include cloud IaaS support, thus future research could definitely focus on this.

The expert validation of the prototype made clear that re-deployment is currently a missing feature within the prototype. Future work should therefore definitely focus on implementing the `update` lifecycle as already defined within the TOSCA standard. Existing code needs to be replaced with new code, and databases should be updated – both their data and their schema – preferable whilst keeping the application up and running. Both cloud-specific and more general practices for (live) updates of software can be incorporated within the prototype. Existing research that focusses on switching components within feature models can perhaps be used (Cetina et al., 2008) and a link towards autonomic computing should definitely be made (Shaw et al., 2013).

During the expert validation it was also shown that little enthusiasm exist for commercializing the deployment scenario generator. Therefore, besides further researching the practicalities of this generator, its business value should be assessed. Through interviewing experts from a multitude of software development and consulting companies, specific use cases where the generator will be of value should be defined. These findings can then be used to further the development of the scenario generator in these specific areas.

References

Amazon. (2006). Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta [Press release]. Retrieved from http://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

Andrikopoulos, V., Binz, T., Leymann, F., & Strauch, S. (2012). How to adapt applications for the Cloud environment. *Computing*, *95*(6), 493–535.

Ardagna, D., Di Nitto, E., Mohagheghi, P., Mosser, S., Ballagny, C., D'Andria, F., … Sheridan, C. (2012). MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)* (pp. 50–56). IEEE.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., … Zaharia, M. (2009). *Above the clouds: A Berkeley view of cloud computing [Technical report]*.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., … Stoica, I. (2010). A view of cloud computing. *Communications of the …*. Retrieved from http://dl.acm.org/citation.cfm?id=1721672

Baliga, J., Ayre, R. W. a, Hinton, K., & Tucker, R. S. (2011). Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport. *Proceedings of the IEEE*, *99*(1), 149–167. doi:10.1109/JPROC.2010.2060451

Bartlett, A. A. (2012). Arithmetic, Population and Energy [Video file]. Retrieved from https://www.youtube.com/watch?v=sI1C9DyIi_8

Bedi, P., Kaur, H., & Gupta, B. (2012). Trustworthy Service Provider Selection in Cloud Computing Environment. *2012 International Conference on Communication Systems and Network Technologies*, 714–719.

Beloglazov, A., Abawajy, J., & Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems*, *28*(5), 755–768.

Benavides, D., Segura, S., & Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, *35*(6), 615–636.

Benavides, D., Trinidad, P., & Ruiz-cort, A. (2005). Automated Reasoning on Feature Models, *01*, 491–503.

Binz, T., Breitenbücher, U., Kopp, O., & Leymann, F. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services* (pp. 527–549). Springer New York.

Boehm, B., Brown, J., & Lipow, M. (1976). Quantitative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering*, 592–605.

Bousselmi, K., Brahmi, Z., & Gammoudi, M. M. (2014). Cloud Services Orchestration: A Comparative Study of Existing Approaches. *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, 410–416. doi:10.1109/WAINA.2014.72

Bova, T., & Petri, G. (2013). Cool Vendors in Cloud Services Brokerage.

Bradshaw, S., Millard, C., & Walden, I. (2011). Contracts for clouds: comparison and analysis of the Terms and Conditions of cloud computing services. *International Journal of Law and Information Technology*, *19*(3), 187–223.

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers.

Brogi, A., Soldani, J., & Wang, P. (2014). TOSCA in a Nutshell : Promises and Perspectives. *Service-Oriented and Cloud Computing*, 171–186.

BusinessWeek. (1998). STEVE JOBS: “THERE”S SANITY RETURNING’No Title. Retrieved May 27, 2015, from http://www.businessweek.com/1998/21/b3579165.htm

Buyya, R., Ranjan, R., & Calheiros, R. N. (2010). InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. *Algorithms and Architectures for Parallel Processing*, 13–31.

Caldiera, V. R. B. G., & Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of Software Engineering*, *2*(1994), 528–532.

Carlson, M., Chapman, M., Heneveld, A., Hinkelman, S., Johnston-Watt, D., Karmarkar, A., … Yendluri, P. (2012). *Cloud Application Management for Platforms*. Retrieved from http://cloudspecs.org/camp/CAMP-v1.0.pdf

Cavalcante, E., Almeida, A., Batista, T., Cacho, N., Lopes, F., Delicato, F. C., … Pires, P. F. (2012). Exploiting Software Product Lines to Develop Cloud Computing Applications, *II*, 179–186.

Cetina, C., Fons, J., & Pelechano, V. (2008). Applying Software Product Lines to Build Autonomic Pervasive Systems. *Software Product Line Conference, 2008. SPLC ’08. 12th International*, 117 – 126.

Claybrook, B. (2011). Cloud interoperability: Problems and best practices. Retrieved from http://www.computerworld.com/article/2508726

Clements, P., & Northrop, L. (2002). Software Product Lines: Practices and Patterns. Addison-Wesley.

Cloud Spectator. (2015). CLOUD VENDER BENCHMARK 2015: Part I Price Comparison Report Among 10 Top IaaS Providers Now Available at Cloud Spectator. Retrieved from http://cloudspectator.com/2015/01/cloud-vender-benchmark-2015-part-i-price-comparison-report-among-10-top-iaas-providers-now-available-at-cloud-spectator/

Cloud Standards Customer Council. (2014). Cloud Standards Wiki. Retrieved from http://cloud-standards.org/wiki/

Columbus, L. (2013). Predicting Enterprise Cloud Computing Growth. Retrieved from http://www.forbes.com/sites/louiscolumbus/2013/09/04/predicting-enterprise-cloud-computing-growth/

Cook, T. D., Campbell, D. T., & Day, A. (1979). *Quasi-experimentation: Design & analysis issues for field settings*. Boston: Houghton Mifflin.

Crago, S., Dunn, K., & Eads, P. (2011). Heterogeneous cloud computing. *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 378–385.

Cunha, D., Neves, P., & Sousa, P. (2014). PaaS manager: A platform-as-a-service aggregation framework. *Computer Science and Information Systems*, *11*(4), 1209–1228.

Dastjerdi, A., & Buyya, R. (2011). A taxonomy of QoS management and service selection methodologies for Cloud computing. *Cloud Computing: Methodology, Systems, and Applications*.

Dillon, T., Wu, C., & Chang, E. (2010). Cloud Computing: Issues and Challenges. *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 27–33.

DZone, I. (2014). The 2014 Cloud Platform Research Report [White paper]. Retrieved from http://library.dzone.com/whitepapers/dzone-research-presents-2014

Ferrarons, J., Adhana, M., Colmenares, C., Pietrowska, S., Bentayeb, F., & Darmont, J. (2014). PRIMEBALL: a Parallel Processing Framework Benchmark for Big Data Applications in the Cloud. *Performance Characterization and Benchmarking*, 109–124. Distributed, Parallel, and Cluster Computing; Databases.

Ferry, N., Rossini, A., Chauvel, F., Morin, B., & Solberg, A. (2013). Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. *2013 IEEE Sixth International Conference on Cloud Computing*, 887–894.

Flood, G. (2013). Gartner Tells Outsourcers: Embrace Cloud Or Die. Retrieved from http://www.informationweek.com/cloud/infrastructure-as-a-service/gartner-tells-outsourcers-embrace-cloud-or-die/d/d-id/1110991

Fogarty, K. (2011). Cloud Computing Standards: Too Many, Doing Too Little. Retrieved from http://www.cio.com/article/2409412/cloud-computing/cloud-computing-standards--too-many--doing-too-little.html

Garcıa-Galán, J., Rana, O., Trinidad, P., & Ruiz-Cortés, A. (2013). Migrating to the Cloud: a Software Product Line based analysis. *3rd International Conference on Cloud Computing and Services Science (CLOSER'13)*, 416–426.

Ghanam, Y., Ferreira, J., & Maurer, F. (2012). Emerging Issues & Challenges in Cloud Computing — A Hybrid Approach, *2012*(November), 923–937.

Grozev, N., & Buyya, R. (2012). Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*.

Heller, M. (2014). PaaS shoot-out: Cloud Foundry vs. OpenShift. Retrieved from http://www.infoworld.com/article/2608610/cloud-computing/cloud-computing-paas-shoot-out-cloud-foundry-vs-openshift.html

Herbst, N., Kounev, S., & Reussner, R. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*, 23–27.

Hevner, A. R., March, S., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105. Retrieved from http://www.springerlink.com/index/pdf/10.1007/s11576-006-0028-8

Hsu, C. (2014). A Cloud Service Selection Model Based on User-Specified Quality of Service Level. *ICAIT*, 43–54.

Iankoulova, I., & Daneva, M. (2012). Cloud computing security requirements: A systematic review. *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*, 1–7. doi:10.1109/RCIS.2012.6240421

Jansen, S. (2014). Autonomous Re-Deployment over Heterogeneous Cloud Ecosystems in a Multi-Objective Problem Space, 1–15.

Jula, A., Sundararajan, E., & Othman, Z. (2014). Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, *41*(8), 3809–3824.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, S. (1990). *Feature-oriented domain analysis (FODA) feasibility study [Technical Report CMU/ SEI-90-TR-21]*.

Kumar, V., Garg, K., & Quan, C. (2012). Migration of services to the cloud environment: Challenges and best practices. *International Journal of Computer …*, *55*(1), 1–6. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Migration+of+Services+to+the+Cloud+Environment+:+Challenges+and+Best+Practices#0

Kuyoro, S. O., Ibikunle, F., & Awodele, O. (2011). Cloud computing security issues and challenges. *International Journal of Computer Networks*, *3*(5), 247–255.

Lando, G. (2014). A Game of Stacks : OpenStack vs. CloudStack. Retrieved from http://www.getfilecloud.com/blog/2014/02/a-game-of-stacks-openstack-vs-cloudstack/#.VI75RSuG9HU

Lawton, G. (2008). Developing software online with platform-as-a-service technology. *Computer*, *41*(6), 13–15.

Lewis, G. A. (2012). The Role of Standards in Cloud- Computing Interoperability [Technical Note]. *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, 1652–1661.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., & Leaf, D. (2011). NIST cloud computing reference architecture. *Recommendations of the National Institute of Standards and Technology*.

Louridas, P. (2010). Up in the air: Moving your applications to the cloud. *IEEE Software*, *27*(4), 6–11.

Lucas-Simarro, J. L., Moreno-Vozmediano, R., Montero, R. S., & Llorente, I. M. (2013). Scheduling strategies for optimal service deployment across multiple clouds. *Future Generation Computer Systems*, *29*(6), 1431–1441.

Machado, G., Hausheer, D., & Burkhard, S. (2009). Considerations on the Interoperability of and between Cloud Computing Standards. *27th Open Grid Forum (OGF27), G2C-Net Workshop: From Grid to Cloud Networks*, 1–4. Retrieved from http://www.intercloudtestbed.org/uploads/2/1/3/9/21396364/considerations_on_the_interoperability_of_and_between_cloud_computing_standards.pdf

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing — The business perspective. *Decision Support Systems*, *51*(1), 176–189.

Martin, S. (2009). Moving Toward an Open Process on Cloud Computing Interoperability. Retrieved from http://blogs.msdn.com/b/stevemar/archive/2009/03/26/moving-toward-an-open-process-on-cloud-computing-interoperability.aspx

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing: Recommendations of the National Institute of Standards and Technology [Technical report]*.

Menzel, M., & Ranjan, R. (2012). CloudGenius: decision support for web server cloud migration. *Proceedings of the 21st International Conference on World Wide Web*, 979–988.

Merle, P., Rouvoy, R., & Seinturier, L. (2011). A reflective platform for highly adaptive multi-cloud systems. In *Adaptive and Reflective Middleware on Proceedings of the International Workshop - ARM '11* (pp. 14–21). New York, New York, USA: ACM Press. doi:10.1145/2088876.2088879

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. doi:10.1145/1118890.1118892

Moscato, F., Aversa, R., Di Martino, B., Petcu, D., Rak, M., & Venticinque, S. (2011). An Ontology for the Cloud in mOSAIC. *Cloud Computing: Methodology, Systems, and Application*, 467–486.

Nair, S. K., Porwal, S., Dimitrakos, T., Ferrer, A. J., Tordsson, J., Sharif, T., … Khan, A. U. (2010). Towards secure cloud bursting, brokerage and aggregation. *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, 189–196.

OASIS. (2013). Orchestration Specification for Cloud Applications Version 1.0. Retrieved from http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html

OMG. (2011). Cloud Standards Customer Council. Retrieved from http://www.cloud-council.org/press-release/04-07-11-pr.htm

Paraiso, F., & Haderer, N. (2012). A federated multi-cloud PaaS infrastructure. *5th IEEE International Conference on Cloud Computing*, 392–399.

Paraiso, F., Merle, P., & Seinturier, L. (2014). soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*.

Parkhill, D. F. (1966). *The Challenge of the Computer Utility*.

Pearson, S. (2009). Taking account of privacy when designing cloud computing services. *… Software Engineering Challenges of Cloud Computing*.

Petcu, D. (2011). Portability and Interoperability between Clouds : Challenges and Case Study (Invited Paper), (Section 3), 62–74.

Petcu, D. (2013a). Building Automatic Clouds with an Open-source and Deployable Platform-as-a-service. In *Cloud Computing and Big Data*.

Petcu, D. (2013b). Multi-Cloud: expectations and current approaches. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds* (pp. 1–6).

Petcu, D., Macariu, G., Panica, S., & Crăciun, C. (2013). Portable Cloud applications—From theory to practice. *Future Generation Computer Systems*, *29*(6), 1417–1430.

Porter, M. (2008). *Competitive advantage: Creating and sustaining superior performance*.

Quinton, C., Haderer, N., Rouvoy, R., & Duchien, L. (2013a). Towards multi-cloud configurations using feature models and ontologies. *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds - MultiCloud '13*, 21.

Quinton, C., Haderer, N., Rouvoy, R., & Duchien, L. (2013b). Towards Multi-Cloud Configurations Using Feature Models and Ontologies, 1–6.

Rimal, B. P., Choi, E., & Lumb, I. (2009). A Taxonomy and Survey of Cloud Computing Systems. *2009 Fifth International Joint Conference on INC, IMS and IDC*, 44–51. doi:10.1109/NCM.2009.218

Rong, C., Nguyen, S. T., & Jaatun, M. G. (2013). Beyond lightning: A survey on security challenges in cloud computing. *Computers & Electrical Engineering*, *39*(1), 47–54. doi:10.1016/j.compeleceng.2012.04.015

Sampson, L. (2012). A cloud broker can be a cloud provider's best friend. Retrieved from http://searchcloudprovider.techtarget.com/feature/A-cloud-broker-can-be-a-cloud-providers-best-friend

Saripalli, P., & Pingali, G. (2011). MADMAC: Multiple Attribute Decision Methodology for Adoption of Clouds. *2011 IEEE 4th International Conference on Cloud Computing*, 316–323. doi:10.1109/CLOUD.2011.61

Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., … Wuttke, J. (2013). Software Engineering for Self-Adaptive Systems : A Second Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 1–32.

Sheth, A., & Ranabahu, A. (2010). Semantic modeling for cloud computing, part 2. *Internet Computing, IEEE*, 81–83. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5496808

Silva, E. da, & Lucrédio, D. (2012). Software engineering for the cloud: a research roadmap. *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, 71–80.

Silva, Rose, & Calinescu. (2013). A Systematic Review of Cloud Lock-In Solutions. In *IEEE 5th CloudCom* (pp. 363–368).

Silva, Rose, L., & Calinescu, R. (2014). Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities. *CloudMDE 2014*.

Spiro, R. L., Stanton, W. J., & Rich., G. A. (2003). *Management of a Sales Force* (12th ed. B.). McGraw-Hill/Irwin.

Standard Coordinating Commitee IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE*, *121990*.

Stellman, A., & Greene, J. (2005). *Applied Software Project Management*. O'Reilly Media, Inc.

Sun, L., Dong, H., Hussain, F. K., Hussain, O. K., & Chang, E. (2014). Cloud service selection: State-of-the-art and future research directions. *Journal of Network and Computer Applications*, *45*, 134–150.

Thatmann, D., Slawik, M., Zickau, S., & Axel, K. (2012). Towards a Federated Cloud Ecosystem : Enabling Managed Cloud Service Consumption, 223–233.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2014). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, *47*(1).

Toosi, A. N., Calheiros, R. N., & Buyya, R. (2014). Interconnected Cloud Computing Environments : Challenges , Taxonomy , and Survey, *47*(1), 1–47.

Tordsson, J., Montero, R. S., Moreno-Vozmediano, R., & Llorente, I. M. (2012). Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems*, *28*(2), 358–367.

Trinidad, P., Cortés, A. R., Peña, J., & Benavides, D. (2007). Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. *Splc (2)*, 51–56.

Vaquero, L., & Rodero-Merino, L. (2008). A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, *39*(1), 50–55.

W3Techs. (2015). Usage of content management systems for websites. Retrieved May 15, 2015, from http://w3techs.com/technologies/overview/content_management/all/

Wadhwa, B., Jaitly, A., Hasija, N., & Suri, B. (2015). Cloud Service Brokers: Addressing the New Cloud Phenomenon. *Informatics and Communication Technologies for Societal Development*, 29–40.

Wadhwa, B., Jaitly, A., & Suri, B. (2014). Making sense of academia-industry gap in the evolving cloud service brokerage. *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices - SER&IPs 2014*, 6–9.

Wang, C., Wang, Q., Ren, K., & Lou, W. (2010). Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. *2010 Proceedings IEEE INFOCOM*, 1–9. doi:10.1109/INFCOM.2010.5462173

Wettinger, J., Andrikopoulos, V., Strauch, S., & Leymann, F. (2013). Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment. *2013 IEEE Sixth International Conference on Cloud Computing*, 478–485.

Wittern, E., Kuhlenkamp, J., & Menzel, M. (2012). Cloud Service Selection Based on Variability Modeling. *Service-Oriented Computing*, 127–141.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

Zeng, W., Zhao, Y., & Zeng, J. (2009). Cloud service and service selection algorithm research. *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation - GEC '09*, 1045. doi:10.1145/1543834.1544004

Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, *1*(1), 7–18.