Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3322

Concepts for Integrating DevOps Methodologies with Model-Driven Cloud Management Based on TOSCA

Johannes Wettinger

Course of Study: Software Engineering

Examiner: Prof. Dr. Frank Leymann

Supervisor: Dipl.-Inf. (FH) Isabell Schwertle
Dipl.-Inf. Tobias Binz

Commenced: May 2, 2012

Completed: October 23, 2012

CR-Classification: C.2.4, D.2.11, D.2.12, K.1

IBM Confidential

Abstract

The paradigm of Cloud computing introduces new approaches to manage IT services and applications. Those approaches overcome traditional IT infrastructure and service management. One of the main goals of Cloud computing is to automate the whole management of IT services in order to reduce costs and to make the execution of management tasks less error-prone. To make this happen, Cloud providers offer proprietary tools to create and manage services in the Cloud. However, when services get more complex it is hard to manage them because those tools aim to be simple and thus provide limited functionality only. In addition, a particular service that was built based on a certain Cloud offering is bound to this offering including all its management aspects. Consequently, a service cannot be easily moved from one Cloud provider's infrastructure to another one's infrastructure.

Today, tools and frameworks implementing so called "DevOps methodologies" can be used to realize management of Cloud services without binding a service to a particular Cloud provider. Nevertheless, complex services are still hard to manage by following those methodologies. To make such services manageable and enable automation of management tasks, a holistic service model is needed. Thus, model-driven Cloud management is an emerging paradigm to realize a holistic management approach for services in the Cloud. Because the DevOps approach and the model-driven approach are originating in different backgrounds, model-driven Cloud management does not cover some aspects of DevOps methodologies that are key for Cloud services.

This thesis is focused on integrating DevOps methodologies with model-driven Cloud management. The goal is to combine the strengths of both approaches in order to minimize the shortcomings of the individual approaches.

Contents

1 Introduction 11
- 1.1 Problem Statement . . . 11
- 1.2 Motivating Scenario . . . 12
- 1.3 Research Design . . . 12
- 1.4 Outline . . . 13

2 State of the Art 15
- 2.1 Background . . . 15
  - 2.1.1 Cloud Computing . . . 15
  - 2.1.2 Development and Operations (DevOps) . . . 16
- 2.2 IT Infrastructure and Service Management . . . 17
- 2.3 Automated Configuration Management: Infrastructure as Code . . . 18
  - 2.3.1 Overview of Products . . . 19
  - 2.3.2 CFEngine . . . 20
  - 2.3.3 Puppet . . . 22
  - 2.3.4 Opscode Chef . . . 23
- 2.4 Model-Driven Cloud Management . . . 25
  - 2.4.1 Topology and Orchestration Specification for Cloud Applications (TOSCA) 27
  - 2.4.2 IBM Common Cloud Stack . . . 31
  - 2.4.3 Canonical Juju . . . 33

3 Integrating Configuration Management with Model-Driven Cloud Management 35
- 3.1 The Running Example: Sugar as a Cloud Service . . . 36
- 3.2 Use Cases Based on the Running Example . . . 38
- 3.3 Bringing DevOps Methodologies into TOSCA . . . 39
  - 3.3.1 Support for Multiple Environments . . . 39
  - 3.3.2 Natural and Transparent Integration of Configuration Management . . . 40
  - 3.3.3 TOSCA Plans and Configuration Management . . . 44
  - 3.3.4 Evaluation . . . 45
- 3.4 Creating TOSCA Models for Existing Services . . . 46
  - 3.4.1 Model Creation Steps . . . 46
  - 3.4.2 Proposal for a Semi-Automatic Creation Procedure Based on Chef . . . 47
  - 3.4.3 Evaluation . . . 52
- 3.5 Realizing a TOSCA Container Fitting DevOps Methodologies . . . 53
  - 3.5.1 TOSCA Container Architecture . . . 54
  - 3.5.2 Model Transformation for Reusing Existing Artifacts . . . 56
  - 3.5.3 Evaluation . . . 56

4 Design and Implementation 59
4.1 Integration of Chef Artifacts into the Sugar Cloud Service Archive 60
4.1.1 Natural Integration 61
4.1.2 Transparent Integration 64
4.1.3 Combined Integration Based on a Preprocessor Component 65
4.1.4 Evaluation 66
4.2 Extension of IBM Common Cloud Stack to Process Chef Artifacts 67
4.2.1 Extension Based on an Existing Chef Server 68
4.2.2 Extension Based on Chef Solo 73
4.2.3 Evaluation 73

5 Conclusion and Future Work 75

A Pseudo Code to Generate a Topology Template 77

Bibliography 79

List of Abbreviations

API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CRM	Customer Relationship Management
CSAR	Cloud Service Archive
DevOps	Development and Operations
IaaS	Infrastructure as a Service
ITIM	IT Infrastructure Management
ITSM	IT Service Management
JAXB	Java Architecture for XML Binding
JSON	JavaScript Object Notation
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SOA	Service-Oriented Architecture
SQL	Structured Query Language
SSH	Secure Shell
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
WSDL	Web Services Description Language
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

List of Figures

2.1 Cloud Computing Service Models Forming a Stack [MG11] 16
2.2 Evolving Maturity of IT Function [Sal04] 19
2.3 Configuration Management Products: Chain of Inspiration [GHS10] 20
2.4 Architecture Overview of CFEngine in a Client/Server Scenario 21
2.5 Architecture Overview of Puppet in a Client/Server Scenario 22
2.6 Architecture Overview of Chef in a Client/Server Scenario 24
2.7 Architecture Overview of Using Hosted Chef 24
2.8 Deployment of a Web Application Based on Infrastructure as Code 26
2.9 Sample Infrastructure Topology for a Web Application Based on Virtual Machines 26
2.10 Higher-Level Infrastructure Topology for a Web Application 27
2.11 TOSCA Service Template Structure [Org12b] 28
2.12 TOSCA Node Type Definition: Structure (above) and Example (below) [Org12b] 29
2.13 TOSCA Topology Template Structure 30
2.14 An Instance of the Topology Template Outlined in Figure 2.13 30
2.15 A TOSCA Plan Linked to the Service Topology [BBLS12] 31
2.16 Process Service Template Using Common Cloud Stack: Simplified Logical Flow 32
2.17 Structure of the “mysql” Charm [Can12a] 34

3.1 Concepts Described in Chapter 3 in a Formally Defined Scope 35
3.2 Sugar as a Cloud Service: Overview, Roles, and Dependencies 37
3.3 Logical Topology of the Sugar Cloud Service 38
3.4 Example for Using Requirements and Capabilities in TOSCA [Org12b] 40
3.5 Deployment of the Sugar Cloud Service Using Configuration Management . . . 41
3.6 Sugar Cloud Service Topology with “Install” and “Configure” Artifacts Attached 42
3.7 TOSCA-Based Example for How to Process Custom Artifacts Ideally 43
3.8 Overview of the Semi-Automatic Model Creation Procedure 48
3.9 Example for Step 1.1: Generation of Node Type Definitions 49
3.10 Example for Step 1.4: Generation of the Topology Template 50
3.11 Example for Step 2.1: Refinement of the Topology Template 51
3.12 Four Major Building Blocks of a TOSCA Container 54
3.13 High-Level Overview of the TOSCA Container Architecture 55

4.1 Sample Node Type Definition: MySQL Server in the Sugar Service Template . 60
4.2 Class Diagram for the Modified JUnit Test Case to Generate the Sugar CSAR 63
4.3 Use Case for a CSAR Preprocessor Component for Chef-Specific Artifacts . . . 66
4.4 CSAR Deployment Using IBM Common Cloud Stack and Chef Server 68
4.5 Class Diagram for the Extended TOSCA Importer Part that Generates Plug-Ins 69

4.6 Screenshot of the Topology of the Imported Sugar Cloud Service Archive . . . 70
4.7 Screenshot of the Deployed Sugar Service Instance 71
4.8 Screenshot of the Chef Server's Management User Interface 72
4.9 CSAR Deployment Using IBM Common Cloud Stack and Chef Solo 73

List of Listings

2.1 CFEngine Policy Snippet: Ensure Root Login is Disabled [Zam12, p. 31] 21
2.2 Puppet Manifest Snippet: Ensure Properties of a Configuration File [Loo11, p. 6] 22
2.3 Chef Recipe Snippet: Ensure MySQL Server is Installed 23

4.1 Chef-Specific Artifact Template to Implement MySQL Server's "Install" Operation 62
4.2 Chef Artifact Type Definition 63
4.3 Alternative Run List Definition for Chef-Specific Artifact Template 63
4.4 Script Artifact Template to Implement MySQL Server's "Install" Operation . . 65

1 Introduction

Today, many providers in the field of Cloud computing attract people to create highly scalable applications and services using the providers' proprietary offerings. Those can be infrastructure offerings such as the Amazon Web Services [Ama12] or platform offerings such as Google's App Engine [Goo12]. Many people get attracted by those offerings because the entrance barrier to start creating services and applications is very low. The corresponding meta models are simple and the provided tooling is easy to use. This results in a very flat learning curve.

However, when the services get more and more complex, it is hard to manage them. It may be even impossible to move the service to another Cloud provider because of missing standards that ensure portability. The results are vendor lock-in and bad manageability. Thus, this thesis is focused on improving the manageability of services in the Cloud without abandon portability. Consequently, this thesis does not stick to specific Cloud offerings.

1.1 Problem Statement

As of today, the so called "DevOps methodologies" represent the leading paradigm of efficiently managing services and applications in Cloud scenarios. Those methodologies are mainly realized by service deployment and management aspects using a scripting language or a domain-specific language. This approach is not appropriate for managing complex services. As a consequence, the more complex a service gets, the harder it is to be managed using pure DevOps methodologies.

Thus, this thesis introduces the paradigm of model-driven Cloud management to overcome the limitations outlined before. However, it is pointed out that model-driven Cloud management does not completely replace the DevOps approaches. The model-driven approach does not target many aspects of DevOps methodologies because both paradigms are originating in different backgrounds. This is why the main goal of this thesis is to bring together the strengths of both worlds by integrating DevOps methodologies with model-driven Cloud management. To achieve this goal, major parts of this thesis are based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) [Org12b], an emerging standard to realize model-driven Cloud management.

1.2 Motivating Scenario

The motivating scenario for this thesis consists of the following aspects regarding a service offered and managed using the Cloud computing paradigm:

1. A service model is created by the service creator. Then, he publishes the service model to a "marketplace for service models."
2. The service provider retrieves the model from the marketplace, customizes it, and deploys it to an infrastructure that is offered in the Cloud. Afterward, the created service instance is constantly managed by the service provider.
3. A service consumer can access and use the managed service instance in the Cloud.

This scenario is the foundation for the running example presented in Section 3.1 and the related use cases outlined in Section 3.2. Both, the running example and the use cases are utilized to outline the concepts and prototype implementations described in this thesis.

1.3 Research Design

This thesis is focused on integrating approaches and practices that are state of the art today or that are currently emerging. The goal of the integration is to minimize their individual deficits by combining the strengths of those approaches. By strongly working together with experts at IBM, it is ensured that the running example, the corresponding use cases, and the concepts outlined in Section 3 are related to actual customer requirements and real-world scenarios. In addition, this thesis refers to existing research in the fields of DevOps methodologies and model-driven management of services.

The evaluations outline benefits and drawbacks for the concepts as well as lessons learned and open issues for the prototype implementations that are part of this thesis. Furthermore, the prototype implementations show that the key concepts can be realized in practice.

1.4 Outline

Chapter 1 *Introduction:*
High-level overview and motivation for the topic of this thesis

Chapter 2 *State of the Art:*
Paradigms, technologies, and products that are state of the art today

Chapter 3 *Integrating Configuration Management with Model-Driven Cloud Management:*
Concepts for integrating configuration management with model-driven management of services in the Cloud while configuration management implements DevOps methodologies

Chapter 4 *Design and Implementation:*
Prototype implementations of key concepts outlined in Chapter 3

Chapter 5 *Conclusion and Future Work:*
Summary and potential next steps to follow up the integration of DevOps methodologies with model-driven Cloud management

2 State of the Art

This chapter provides an overview of paradigms, standards, technologies and software products that are currently state of the art or research topics. They are key to understand the concepts presented in this thesis. References are included for getting more detailed information on the different topics.

2.1 Background

In order to provide the necessary background, two key paradigms in the context of this thesis are introduced. These are Cloud computing as well as DevOps (development and operations).

2.1.1 Cloud Computing

The Cloud paradigm is attracting a lot of attention at the moment. In the IBM Tech Trends Survey 2011, more than 90 percent of 2,000 IT professionals stated "that in five years Cloud computing will overtake traditional on-premises computing as the primary way organizations acquire IT [IBM11]." Cloud computing is not a new technology in contrast to what many people think. In fact, it is a new concept of delivering computing, network, and storage resources [Cat10]. The basic idea of Cloud computing is to provide a model that follows the "everything as a service" paradigm: virtualized resources such as instances of operating systems, virtual networks and disks, middleware platforms, and business applications can be provided and consumed at a large scale in the Cloud [LKN+09]. By standardizing these virtualized resources the provisioning costs can be reduced massively. Thus, Cloud computing can be described as the industrialization of IT systems and services: it will fundamentally change the way of using IT resources [Ley09].

Because the Cloud paradigm is still young, it is not easy to find a general and broadly accepted definition of Cloud computing. [VRMCL08] studied more than 20 definitions for the Cloud. In this thesis the proposed consensus definition is used: *"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 resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs."* In addition, the Cloud-related terminology in this thesis is based on the NIST definition of Cloud computing [MG11]. Figure 2.1 shows the stack of service models as they are defined in [MG11]. It illustrates the fact that Cloud services can be provided at

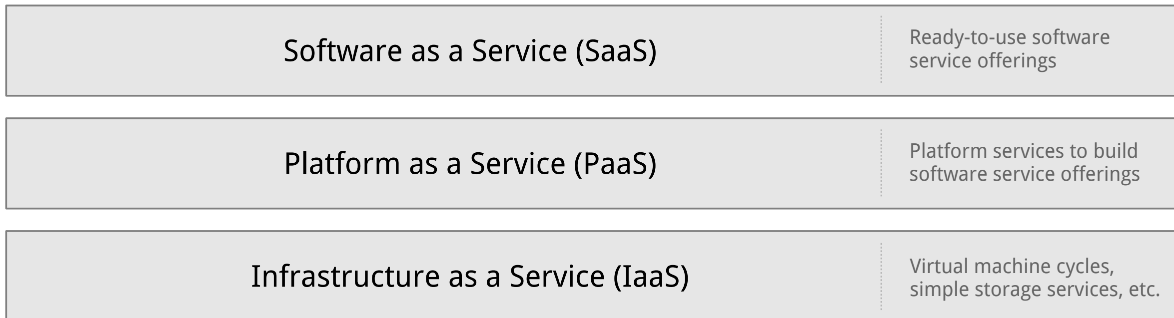

Figure 2.1: Cloud Computing Service Models Forming a Stack [MG11]

different levels. As an example, an IaaS provider offers virtual machine instances whereas a SaaS offering is a ready-to-use software service that the customers can use without worrying about the underlying platform or infrastructure.

Cloud services usually own a complex architecture consisting of many components. The IBM Cloud Computing Reference Architecture [IBM11] is an example for a holistic approach of how to build and manage applications in the Cloud. Beside the stack of Cloud services, the reference architecture cares about many important topics linked to professional Cloud service delivery such as service creation and integration, operational and business support services, security, resiliency as well as performance. The three guiding principles for IBM's reference architecture show the key points for establishing Cloud services: (1) the efficiency principle means to reduce the costs for each service instance hour and to decrease response time for provisioning resources in the Cloud. (2) Target of the lightweight principle is to establish more standards to reduce costs for management tasks. (3) Finally, the economies-of-scale principle states that capital expense, operating expense, and time to market should be minimized by sharing infrastructure and management facilities across Cloud services.

2.1.2 Development and Operations (DevOps)

Many organizations are facing severe problems in changing software that is running in production environments [HM11, Sha11]. Today, the cycle time is often in the magnitude of weeks or even months. This is the time from making the decision to apply a change to this change making it into production [HF10]. Often, the process to bring changes into production is error-prone because the process itself is neither documented properly nor automated [Sha11]. Thus, the process is not repeatable and it is expensive to change something [HM11]. It is even challenging to deploy software into a testing or staging environment because many manual and often undocumented steps need to be performed. It is key to reduce the cycle time to be able to react on problems in time and to deliver new features fast to satisfy the customer requirements. To make the whole release process more reliable and to reduce the cycle time to the magnitude of hours or minutes, a high degree of automation is needed [HF10, p. 3 ff.]. Automation concerns all the steps from building and testing the software to the actual release. The overall

target is to implement the automation by establishing reliable and repeatable processes [HM11]. Therefore, the execution of these processes has to be fast and deterministic.

This thesis is focused on a central part of the problem stated in this section, namely the interface between the developers and the operations personnel. In many organizations the developers transfer the software to the operations team after it is built and tested. The operations personnel then has to deploy the software into the production environment. But the developers do not communicate much with the people at operations. The result is that for the operations team it is cumbersome to do the deployment because there is no properly defined process. Thus, the deployment becomes a time-consuming and error-prone process.

The DevOps movement addresses exactly this problem. DevOps is a portmanteau of development and operations, which emphasizes the need for communication and collaboration between the software developers and the operations personnel [Smi11]. The basic idea behind DevOps is to effectively integrate all participants in the overall process from designing the software to the actual release. Every single person has responsibility for this overall process, not just for one individual part of the process [HF10, p. 28].

The philosophy behind the DevOps movement is to bring agile methodologies into the world of IT operations and infrastructure management [HF10, p. 279]. As a result, DevOps leads to a much more effective and flexible collaboration between the different teams. This is the foundation to make the overall process more reliable and faster. Automation is enabled by introducing the concept of "Infrastructure as Code." This concept is based on the fact that almost any action on the infrastructure level can be automated programmatically [Smi11]. The following Section 2.3 discusses this approach in detail.

Beside the production environment it is very likely that there are more environments for developing, testing, and other purposes [HM11]. To develop and test effectively, it is key that all environments are identical or at least very similar to the actual production environment [HF10, p. 278]. Otherwise it cannot be guaranteed that a piece of software, which is being developed in the development environment will run in the production environment seamlessly.

This need for several similar environments is a typical use case for implementing DevOps methodologies [HF10, p. 279]. The creation of an environment should be a fully automated process, which can be parametrized. The parameters are used to satisfy the requirements for the different environments. As an example, for a usual testing environment there is no disaster recovery solution needed. Thus, the topology of a testing environment is less complex than the production environment's topology.

2.2 IT Infrastructure and Service Management

There is no doubt that information technology (IT) has become an essential element for almost any organization today. IT is no longer an additional offering to simplify certain tasks and perform them more efficiently. Without an IT infrastructure that is up and running and producing correct results, a company will run out of business very soon [Sal04]. Business

processes are the heart of a company. Without an IT infrastructure these processes cannot run at all. [GHS10]

Traditionally, IT is all about providing technology. The IT function within an organization is focused on optimizing the management of the in-house infrastructure. The goal is to effectively provide IT assets such as servers and network connectivity to the organization's departments. This is the first basic stage, which follows the "IT infrastructure management" paradigm (ITIM) [Sal04].

The next step for the IT function as a technology provider is to turn into a service provider, following the "IT service management" paradigm (ITSM). Thus, the focus is moving from pure technology to service orientation, which results in a much more holistic approach to provide information technology. Beside the functional requirements, IT services that are delivered to internal and external customers have to meet several non-functional requirements in categories such as security, performance, and availability. Furthermore, service-level agreements are needed to define the targets regarding quality and costs [Sal04]. The step toward ITSM is of special importance for the whole business because many business processes highly depend on IT services and cannot run without them. This is why the IT function has to become more customer-oriented [GDQC09]. Leaving the isolated island of pure technology, IT has to be managed like a real business including all aspects the customers care about [WCEH09].

There are best practice standards for ITSM. One of the most prominent is the Information Technology Infrastructure Library (ITIL) [APM12, JHMO07]. These standards are key to enable the delivery of IT services of high-quality at reasonable costs [GDQC09]. A case study for implementing a centralized IT service management model based on ITIL is presented in [TCST09].

[SNS07] outlines the approach of a service-oriented infrastructure. This is basically the idea of leveraging virtualization and grid technologies on the infrastructure level. As a result, services based on the concept of service-oriented architecture (SOA) [Erl05] can be realized consequently.

Finally, the third and final stage is IT governance. This is when the IT function becomes a business partner, which allows to realize new business opportunities. The goal is to completely integrate the IT processes with the corresponding business processes, thereby lifting the overall quality of the service to a higher level [Sal04]. Figure 2.2 shows the evolving maturity of the IT function within an organization following the paradigms outlined in this section.

2.3 Automated Configuration Management: Infrastructure as Code

Managing IT services including the underlying infrastructure is a complex and challenging task [GHS10, NS11]. A single service usually consists of different technologies, programming languages, and architectural approaches. In addition, there are several ways of how the different components exchange information in terms of network protocols, communication paradigms, and so on [GHS10]. This results in a heterogeneous infrastructure, which is hard to manage. Furthermore, changing requirements lead to frequent publication of new releases.

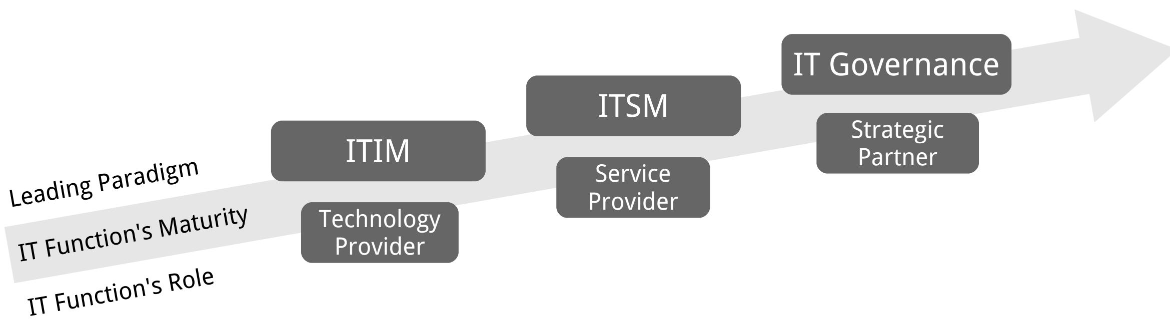

Figure 2.2: Evolving Maturity of IT Function [Sal04]

For development and testing purposes the same application or service has to be instantiated using different configurations [GHS10].

To achieve reliable management of such an infrastructure, management processes have to be automated whenever possible. This can be realized by introducing configuration management [GHS10, NS11]. Tool support is needed to implement the automation, thereby realizing automated configuration management.

One prominent example for a means to realize automation is a domain-specific language. It is an executable language, which is focused and specialized on a very specific application area. The goal of such a language is to express domain knowledge efficiently by introducing abstractions that fit the domain [GHS10]. The following sections present three products for implementing automated configuration management, hereafter also referred to as configuration management. These are CFEngine [CFE12c], Puppet [Pup12e], and Chef [Ops12d]. All of them introduce a domain-specific language as a level of abstraction for describing and managing IT infrastructure in a platform-independent manner. Thus, these products follow the paradigm of Infrastructure as Code [NS11, p. 2 ff.], meaning that the management aspects of the IT infrastructure can be described using a specific language. The “code” that specifies the topology and processes of the infrastructure, is based on this language. Many management aspects such as deployment or termination of an IT service can be highly automated by following this paradigm.

2.3.1 Overview of Products

Those three well established products [CFE12a, Pup12b, Ops12b], namely CFEngine, Puppet, and Chef, have been selected as being representative of tools, which enable automated configuration management. For all three of them both open source and commercial variants are available. The open source variants are free to use, whereas the commercial variants offer additional features, support, and solutions. There is a large and active community around each product [DJV10, p. 12]. These communities provide support and reusable artifacts such as platform-independent deployment scripts to be used with the particular product they belong to. Furthermore, all three products support the leading platforms of today including Windows,

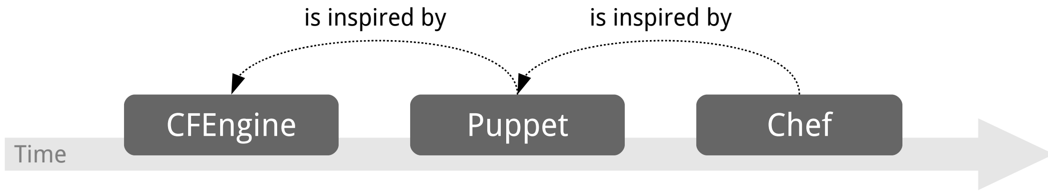

Figure 2.3: Configuration Management Products: Chain of Inspiration [GHS10]

Mac OS X, Linux, and other Unix variants [DJV10, p. 10]. Thus, by looking at CFEngine, Puppet, and Chef representatively, many relevant use cases can be covered from small-scale scenarios based on simple software deployments to very large-scale and business-critical IT services.

Viewed historically, the products are related to one another in a chain of inspiration as shown in Figure 2.3. Even if Puppet is not based on CFEngine directly and Chef does not use Puppet's code base, there are mutual influences [GHS10]. The Puppet developers have previously worked with CFEngine, whereas the founders of Opscode used Puppet intensively [Ops12c, sect. *FAQ – How is Chef different than Puppet?*]. Opscode is the company that developed Chef. At the end of the day, CFEngine, Puppet and Chef are distinct and competing products. Despite their architectural differences, the development of all three products is active and they all try hard to catch up with their competitors.

2.3.2 CFEngine

Initially released in 1993, CFEngine is an ongoing research project at Oslo University College to realize configuration management [Bur05]. CFEngine uses a declarative domain-specific language to express system configurations as a desired state. These expressions are called policies. An example for such a declarative policy is: *Ensure that file `/etc/ssh/sshd_config` contains the line `PermitRootLogin no`.* On a higher level it could be something like *Ensure that MySQL [Ora12] is installed on all database servers* [Zam12, p. 23 f.]. Such a policy is refined by associating more fine-grained policies with it.

CFEngine's current product generation (CFEngine 3) is completely based on a theoretical model called promise theory. [Zam12, p. 25 ff.] outlines the consequences of applying this theory to CFEngine. The basic idea is to have autonomous clients promising their own behavior without having a central server, which enforces these promises on the clients. Of course, the clients' behavior can be influenced from outside. As an example, a central server can transfer a policy to one or more clients. But in the end, each individual client decides how to realize a particular policy. The server does not send explicit commands to the clients such as *Add the line `PermitRootLogin no` to file `/etc/ssh/sshd_config`.* The advantage of implementing the promise theory is that each client consistently ensures its compliance to the current set of policies. Listing 2.1 shows a sample policy snippet for CFEngine.

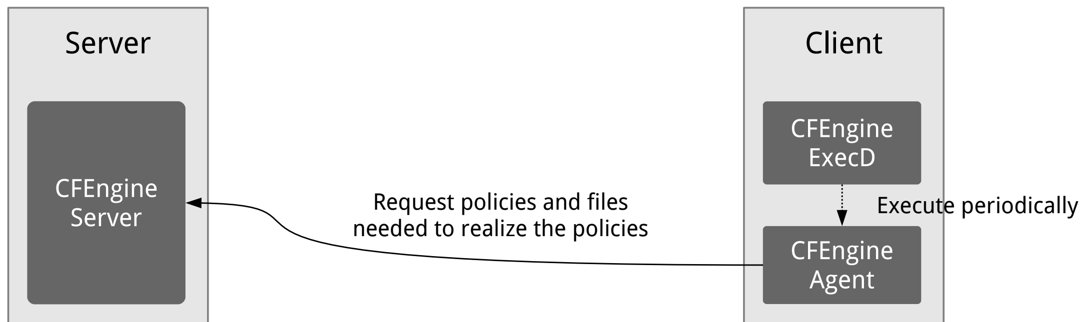

Figure 2.4: Architecture Overview of CFEngine in a Client/Server Scenario

Another architectural principle of CFEngine is convergent configuration. The target system does not have to reach the desired state expressed in a particular policy immediately after the first attempt to realize the corresponding policy [Zam12, p. 27]. Instead, the current state of the target system is constantly compared with the desired state. If current and desired state differ, CFEngine repeats trying to realize the policy on the target system. Step by step, the target system is getting closer to the desired state. Various dependencies are typical for preventing CFEngine from realizing a policy immediately. As an example, the start of an application server may have to be delayed until the related database is up and running, so the actual application can access the data inside the database.

Figure 2.4 outlines the simplified architecture of CFEngine [GHS10]. On the server side the *CFEngine Server* component holds all the policies. The *CFEngine ExecD* component executes the *CFEngine Agent* periodically on the client side. Afterward, the *CFEngine Agent* requests the relevant policies and realizes them on the client. [Zam12, p. 27 ff.] provides more details about the architecture and operation of CFEngine. Another recommended starting point to dig deeper into CFEngine is [CFE12b].

Today, there are three product variants of CFEngine available [CFE12c]: (1) the community edition is open source software and free to use. This edition provides the core components of CFEngine only. (2) The enterprise edition is the commercial variant of CFEngine. It is specially designed for enterprise scenarios. Thus, it offers additional features such as a graphical administration interface, a REST-based Web service API and native support for Windows. (3) The embedded edition can be compiled into any type of electronic device such as routers or smart phones. Thus, many embedded devices can be managed using CFEngine.

Listing 2.1 CFEngine Policy Snippet: Ensure Root Login is Disabled [Zam12, p. 31]

```
files:
  "/etc/ssh/sshd_config"
    comment => "Ensure root login is disallowed",
    edit_line => replace_or_add(".*PermitRootLogin.*", "PermitRootLogin no");
```

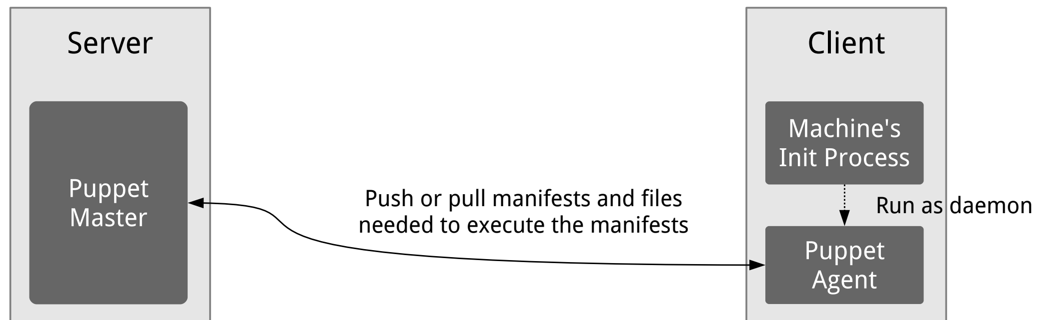

Figure 2.5: Architecture Overview of Puppet in a Client/Server Scenario

2.3.3 Puppet

Puppet's initial release was published in 2005 as open source software [Pup12e]. To express configuration definitions Puppet provides a declarative language, which follows an object-oriented approach. These definitions are called Puppet manifests. The goal is to make these manifests reusable and easy to understand [Loo11, p. 1 ff.]. Similar to CFEngine's policies, the manifests describe the desired state of the target system. Multiple manifests are bundled in a catalog, which gets executed on the target system. As an example, Listing 2.2 shows a manifest snippet to ensure certain properties of a configuration file.

One of Puppet's central architectural principles is that execution of manifests is idempotent [Loo11, p. 2]. That means it does not matter if a single manifest is executed exactly once or more often. In case an error occurs during the first run of a particular manifest, the whole manifest is getting executed again and again until the target system is in the desired state.

The simplified architecture of Puppet is outlined in Figure 2.5. The *Puppet Master* runs on the server side to store the manifests and make them available for the clients. On the client side, the *Puppet Agent* executes the manifests. The agent itself gets started as a daemon – a service running in the background – during the machine's start-up procedure [Loo11, p. 2 ff.]. The Puppet documentation [Pup12c] is a helpful resource to obtain more details to understand and use Puppet.

Puppet Labs offer two product variants [Pup12e]: (1) the open source edition of Puppet is publicly available and can be used for free. (2) The enterprise edition adds advanced features

Listing 2.2 Puppet Manifest Snippet: Ensure Properties of a Configuration File [Loo11, p. 6]

```
file { '/etc/ntp.conf':
  mode => '640',
  owner => root,
  group => root,
  source => '/mnt/nfs/configs/ntp.conf',
}
```

such as a graphical user interface and automated configuration auditing. For both variants Puppet Forge [Pup12d] is a place to publicly share pre-built configurations for Puppet.

In addition to the product variants outlined before, Puppet Labs offer a framework called Marionette Collective [Pup12a]. With Puppet only it can get very cumbersome to automate a deployment and management scenario, which includes a large number of machines. Thus, the goal of Marionette Collective is to support the management of clusters and collections of machines. The framework can be used in conjunction with both Puppet and Chef.

2.3.4 Opscode Chef

In 2009, the initial release of Chef [Ops12d] was published by Opscode, a startup at that time. With this, Chef is still a young product with a short history. However, both the Chef community's activity [Ops12a] and the customers of Opscode [Ops12b] confirm the relevance of Chef in the market.

Configuration definitions in Chef are called recipes. One or more recipes are bundled in a cookbook. Similar to Puppet's manifests and CFEngine's policies, recipes express the target state of a system [GHS10]. Listing 2.3 provides a sample recipe snippet to ensure that a certain package is installed. Furthermore, recipes can contain regular Ruby code. As a result, imperative statements can be included in recipes. This is a key difference to Puppet and CFEngine. In addition to cookbooks, roles can be defined. A particular client can have zero or more roles. Afterward, recipes can be associated with one or more roles. This mechanism allows to link recipes to clients without assigning them directly [GHS10].

The founders of Opscode used Puppet intensively before they started to create Chef [Ops12c, sect. *FAQ – How is Chef different than Puppet?*]. As a result, the architectural principles of Chef are similar to Puppet. Thus, executing a recipe is also idempotent as described in Subsection 2.3.3. When creating new recipes including imperative statements, idempotence may have to be ensured explicitly. The basic architecture of Chef is outlined in Figure 2.6: the *Chef Server* stores all the cookbooks and role definitions. The server component manages a "run list" for each machine that is registered. A particular run list assigns recipes and roles to a machine. In addition, the *Chef Server* can be used as a central data store to exchange information between different clients. For this purpose, Chef's so called data bags can be used [Ops12c, sect. *Data Bags*]. On the client side, the *Chef Client* component runs as a daemon and periodically pulls the corresponding recipes from the server. Furthermore, the *Chef Client* ensures that the recipes are executed correctly on the target system [Ops12c].

Listing 2.3 Chef Recipe Snippet: Ensure MySQL Server is Installed

```
package "mysql-server" do
  version "5.5.25"
  action :install
end
```

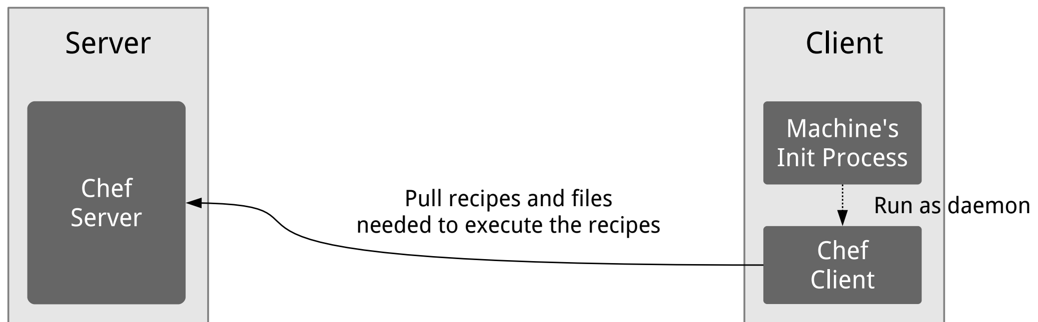

Figure 2.6: Architecture Overview of Chef in a Client/Server Scenario

Opscode offers three variants of its configuration management product. (1) Hosted Chef is a Software as a Service offering to manage the IT infrastructure of an organization without installing and managing an own Chef Server [Ops12c, sect. *Architecture – Hosted Chef*]. The infrastructure does not have to be on premise. It can also consist of virtual machines run by an Infrastructure as a Service provider. Figure 2.7 provides an architecture overview of using Hosted Chef. (2) Private Chef offers the same set of features as Hosted Chef does, but it runs inside the organization's own infrastructure. (3) Most of the product functionality is offered by Open Source Chef for free [Ops12d]. For such a young product, the community of Chef [Ops12a] is remarkably active in sharing cookbooks and experience as well as providing support.

In contrast to CFEngine and Puppet, Chef uses an internal domain-specific language, not an external one [GHS10]. The foundation of an internal domain-specific language is an existing programming language. [GHS10] outlines the differences between internal and external domain-specific languages. In case of Chef, Ruby is the underlying programming language. This architectural principle allows to extend Chef easily in many ways [Ops12c]. In addition, many development tools designed for Ruby can be used for creating Chef artifacts such as recipes or extensions. The fact that Chef uses an internal domain-specific language based on Ruby is the main reason why the prototype implementations developed as part of this thesis are based on

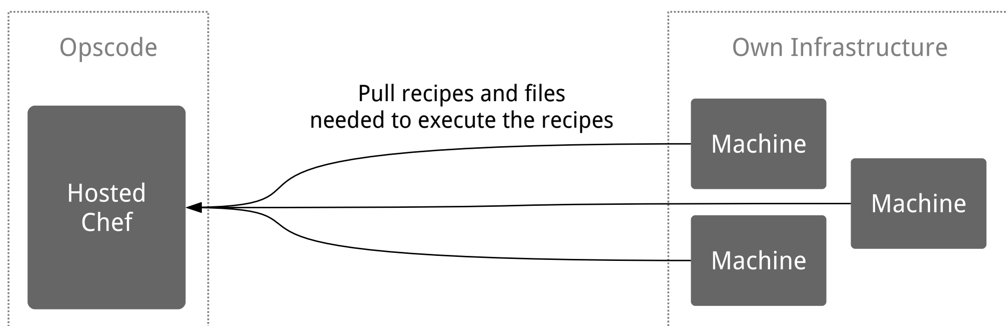

Figure 2.7: Architecture Overview of Using Hosted Chef

Chef. Ruby is a widespread programming language with much tool support available [Rub12]. In addition, Chef's open source variant includes many advanced features such as a graphical user interface for free. For CFEngine and Puppet, these are available only in the commercial variants. This fact makes Chef a good choice to be used in research.

Similar to what Marionette Collective [Pup12a] is in the Puppet world, Spiceweasel [Ops12e] is a Chef-based command line tool to ease the management of a large number of machines. But Spiceweasel is much simpler and more limited compared to Marionette Collective. The configuration of a collection of machines can be described using a document written in either "JavaScript Object Notation" (JSON) [JSO12] or "YAML Ain't Markup Language" (YAML) [YAM12]. Then, several Chef commands are being generated based on this document to perform the actual deployment.

2.4 Model-Driven Cloud Management

Section 2.3 outlined the paradigm of Infrastructure as Code. The advantages of implementing this paradigm were stated and they are widely recognized. But managing a large and complex topology of different machines with plain configuration management tools can be cumbersome and time-consuming. Even the infrastructure for providing a usual Web application can become complex very quickly: there are several technologies needed to realize load balancing, caching and a full-text index. To specify such an infrastructure a lot of code is being created. That code is based on a domain-specific scripting language. As a result, it is hard to keep the code structure clean. Every single change becomes a risk because it is hard to estimate the consequences of that particular change [NS11, p. 5 ff.].

The pure paradigm of Infrastructure as Code does not provide a holistic meta model for describing and managing such a topology. At the end of the day, Infrastructure as Code means to have a set of lower-level artifacts, which are basically declarative scripts. Those artifacts can be linked to one another and assigned to nodes in the topology. Additional tools such as Marionette Collective [Pup12a] and Spiceweasel [Ops12e] provide more convenience to manage a large number of nodes. Though, they do not lift configuration management tools to the level of model-driven management and do not introduce a holistic meta model for managing complex topologies. Figure 2.8 illustrates a sample deployment of a typical Web application based on the paradigm of Infrastructure as Code.

From a perspective of pure configuration management, a node is always a physical or virtual machine. But the real topology of an infrastructure to run a particular application is more than a set of virtual machines connected to one another. On each machine a number of software components are running. Those components can be connected to components running on other machines. Figure 2.9 outlines a sample infrastructure topology for a Web application. It illustrates a much broader understanding of nodes and relationships. This understanding leads to a higher-level model-driven approach for describing the underlying topology of a service or application. Such a model is not limited to express relationships between different software components hosted on virtual machines. As an example, Figure 2.10 shows a topology where two nodes are provided by services offered by a Cloud provider. Furthermore, additional aspects

Figure 2.8: Deployment of a Web Application Based on Infrastructure as Code

that are relevant to a particular topology can be modeled. Two examples are embedding the operating system layer and expressing networking aspects.

This model-driven approach is especially key for applications and services running in the Cloud. In contrast to a simple Web application, their underlying topology is much more complex because Cloud offerings have to be highly scalable. In addition, they may be connected to several other services and applications. The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an emerging standard [Org12a], addressing exactly this need to describe a topology. But TOSCA is not only focused on the topology definition. Processes can be defined to support the whole lifecycle of an application, including deployment, maintenance, and termination [Org12b]. Thus, TOSCA enables model-driven management of Cloud services and applications. The following sections provide more details about TOSCA and the IBM Common Cloud Stack [Dix12], an implementation that is able to process service definitions based on TOSCA. In addition, Canonical Juju [Can12c] is being introduced. Juju is a service orchestration framework to realize model-driven management of Cloud applications. Although the meta models of Juju and TOSCA have similarities, the framework does not support TOSCA-based service definitions. But service orchestration tooling such as Juju can

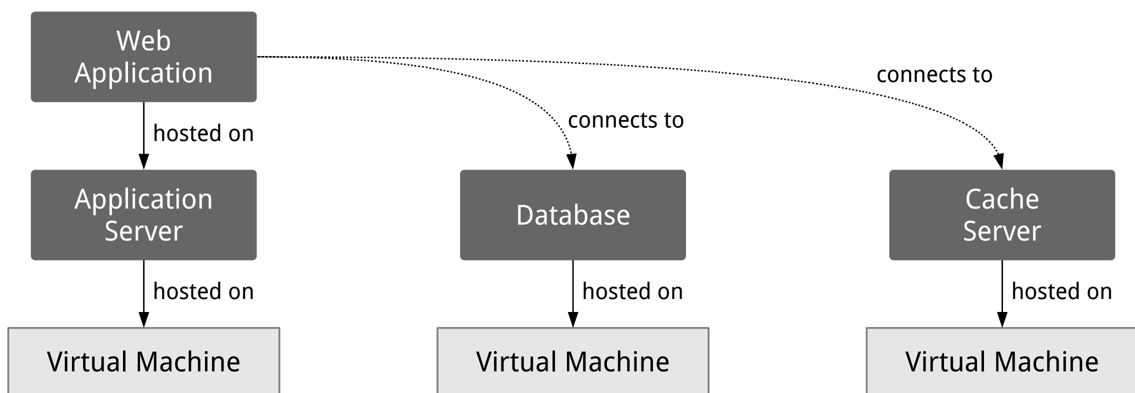

Figure 2.9: Sample Infrastructure Topology for a Web Application Based on Virtual Machines

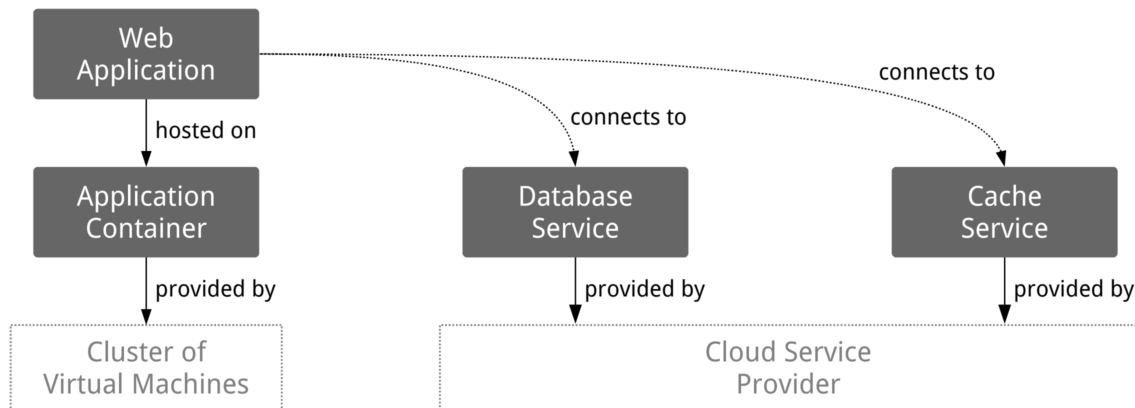

Figure 2.10: Higher-Level Infrastructure Topology for a Web Application

be used to deploy and manage services that are described using TOSCA. Concepts of how to achieve this goal are outlined later in Section 3.5.

In terms of Cloud management, there are products and approaches available beside TOSCA such as VMware's virtualization and Cloud management [VMw12], OpenNebula [MLM11], OpenStack [Ope12], CloudStack [Cit12], and others [HKJ+09, Clo12]. But those are primarily focused on the lower-level infrastructure. Their goal is to simplify the management of virtualized resources such as virtual machines or storage resources. Another approach of model-driven management of IT services is outlined in [BCGG10]. It is focused on managing the orchestration of existing services on a business process level, whereas TOSCA's goal is to describe a service template including the structure and behavior of a particular service offering [Org12b]. This thesis is referring to TOSCA because of its holistic approach to implement model-driven Cloud management. The emerging standard is supported by a number of prominent companies in the industry such as IBM [IBM12b], SAP [SAP12], and Hewlett-Packard [Hew12] as well as research institutions such as the Fraunhofer Society and the Institute of Architecture of Application Systems at the University of Stuttgart.

The model-driven approach of managing Cloud applications and services is not a contradiction to code-driven configuration management. Of course, these two worlds are overlapping in terms of their paradigms to solve problems in Cloud management. However, the goal of this thesis is to bring together the strengths of both worlds and thereby enabling the realization of DevOps methodologies for Cloud applications. The concepts of how to achieve this goal are outlined in Chapter 3.

2.4.1 Topology and Orchestration Specification for Cloud Applications (TOSCA)

As mentioned previously in Section 2.4, the Topology and Orchestration Specification for Cloud Applications [Org12b] is an emerging standard to establish a holistic approach of model-driven management of Cloud applications. This approach supports one of TOSCA's main goals,

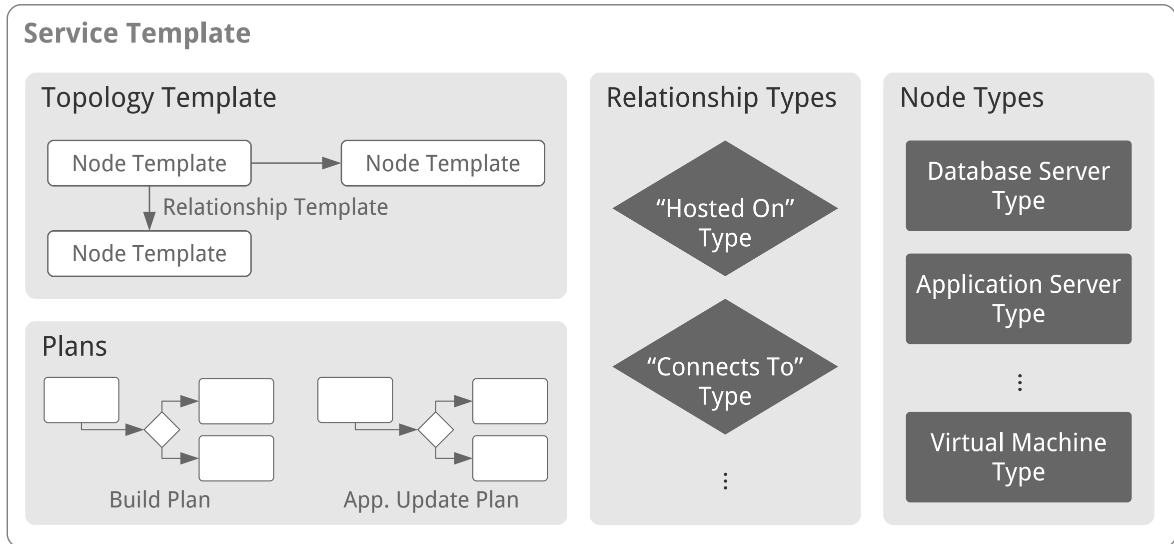

Figure 2.11: TOSCA Service Template Structure [Org12b]

namely to make Cloud services portable [BBLS12]. TOSCA provides an abstract and platform-independent meta model to create Cloud application models that are portable. Before TOSCA was introduced, research focused on migrating services from one Cloud environment to another without taking the portability of management aspects into account [LFM+11, BLS11].

Meanwhile, TOSCA is supported by a number of prominent companies [Org12a] and research institutions, as mentioned previously. The specification is a meta model to describe an IT service including its internal structure and how to manage it. The whole meta model is technically defined by an XML schema definition [Org12b]. It prescribes the structure of a service template based on TOSCA. As outlined in Figure 2.11, a service template consists of four key parts:

Node Types are defined as components to be used inside the topology template. A single node type can be instantiated once or several times as a node template inside the topology template.

Relationship Types can be instantiated as relationship templates inside the topology template to express any kind of relation between two particular nodes.

Topology Template: To define the topological structure of an IT service, a topology template is defined consisting of node and relationship templates.

Plans can be defined to describe management processes. As an example, a build plan can be defined to describe the deployment process of the IT service.

Because nodes are the core components of a topology description in TOSCA, Figure 2.12 explicitly outlines the structure and an example of a node type. First, properties can be assigned to a node type. As an example, two properties named "Username" and "Password"

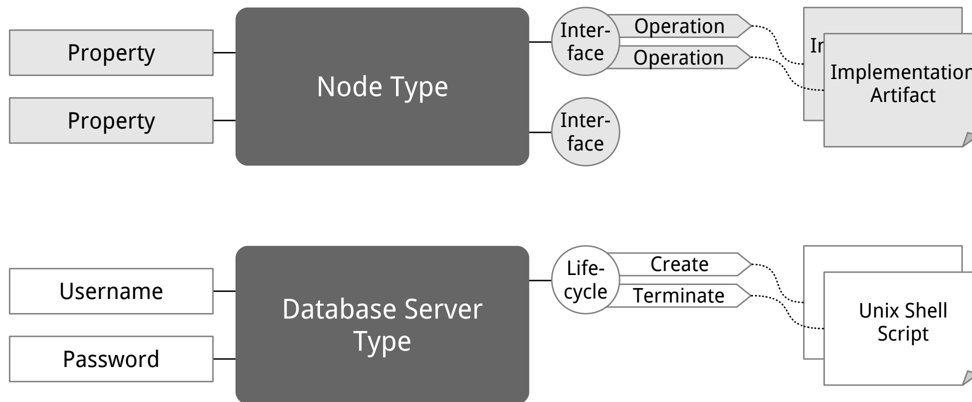

Figure 2.12: TOSCA Node Type Definition: Structure (above) and Example (below) [Org12b]

can be defined for a database server node type to enable database access. Second, a node type can have interfaces. A particular interface can provide operations, which define the possibilities of interaction of a node of the given node type. To come back to the example of a database server, the corresponding node type can own a "Lifecycle" interface providing two operations: "Create" and "Terminate" the database server. As yet, the node type definition is abstract and does not say anything about how an operation is implemented. Thus, one or more concrete implementation artifacts can be linked to an operation. For instance, an implementation for the "Create" operation could be a Unix shell script to install the database server. In addition there could be another script attached to the same operation, which performs the equivalent actions on Windows-based systems. Attaching several implementation artifacts to a particular operation improves the portability of the service template. Relationship types between nodes are defined similarly to the definition of node types. Further details can be obtained from [Org12b].

In addition to implementation artifacts that are attached to a particular node type operation, deployment artifacts can be attached to a node type. Its purpose is to place pieces of software on the machine where the node type is deployed on. Deployment artifacts are typically referenced in implementation artifacts and plans. [Org12b] outlines further details on deployment artifacts.

The topology template is created based on the previously defined node types and relationship types. Figure 2.13 outlines the structure of a topology template. The basic idea is to create an instance of a template based on the corresponding type. As an example, the "Virtual Machine Template" is an instance of the "Virtual Machine Type." The same is true for the relationship "Hosted On." Node templates can have concrete default values for the node type's properties. As an example, the default value of the "Username" property defined in "Database Server Type" could be "admin" for the "Database Server Template." A relationship template defines a concrete relation between two particular node templates.

Figure 2.14 shows an example of an instance of the topology described using the topology template in Figure 2.13. It basically consists of two virtual machines having a stack of

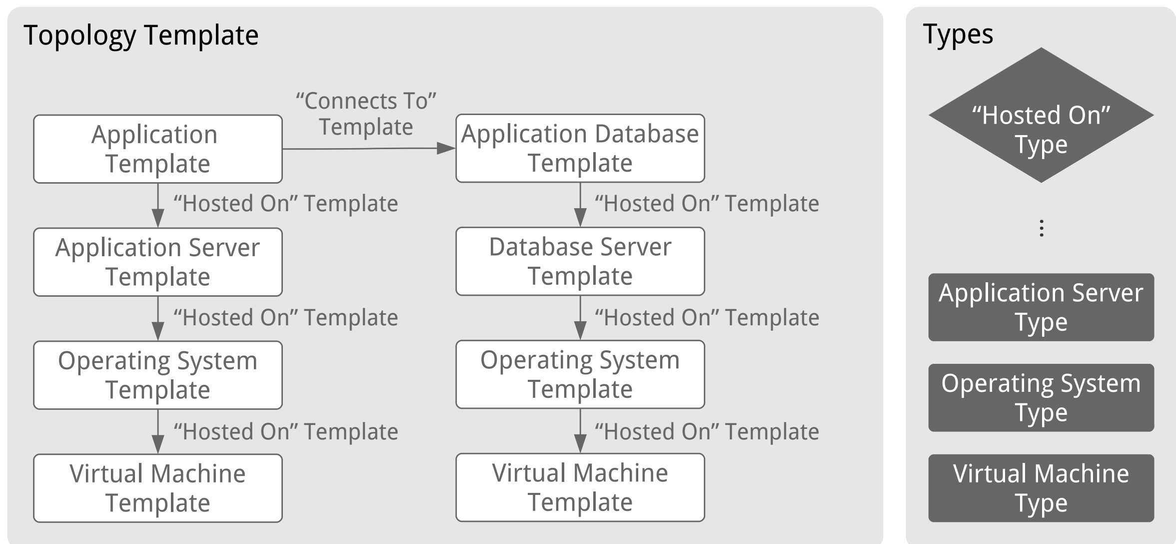

Figure 2.13: TOSCA Topology Template Structure

software components deployed on them. The goal is to run both the application itself and the corresponding application database. As yet, TOSCA does not provide a means to describe such a concrete service instance [BBLS12]. Furthermore, TOSCA does not define any restriction to what a node type or relationship type actually represents [Org12b]. Obviously, a node type can be a virtual machine or any software component running on the machine. But a node type could also represent a network to which a particular machine is connected to. Another option could be a node type, which represents a platform offered by a Platform as a Service provider. Of course, there are many more possibilities of how to utilize nodes and relationships in TOSCA.

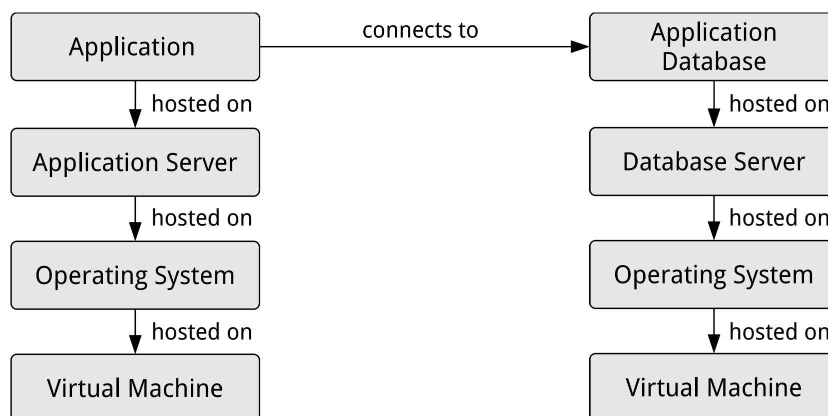

Figure 2.14: An Instance of the Topology Template Outlined in Figure 2.13

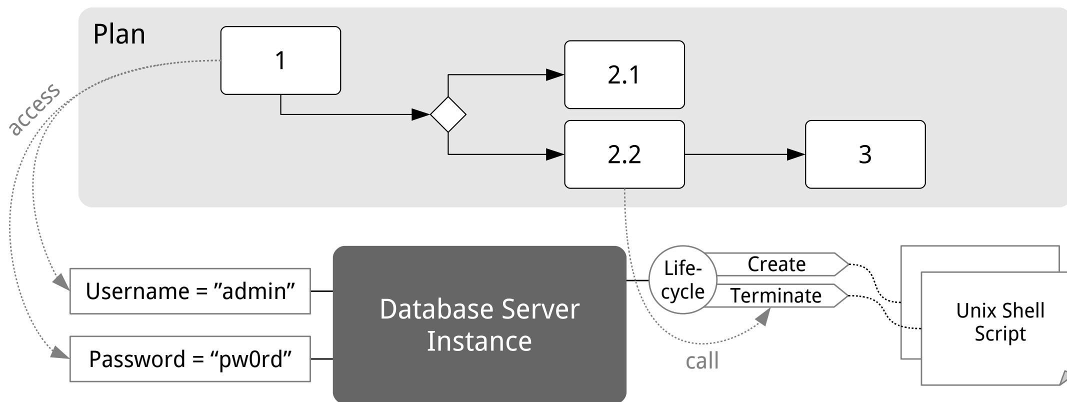

Figure 2.15: A TOSCA Plan Linked to the Service Topology [BBLS12]

Beside TOSCA's means to define the internal structure of a Cloud application, plans can be included into the service template. Plans are basically workflows that describe operational and business-related management tasks. TOSCA does not explicitly define the language to be used for creating plans. But in terms of portability it is strongly recommended to use an established and standardized workflow language such as Business Process Model and Notation (BPMN) [Obj12] or Web Services Business Process Execution Language (BPEL) [Org12c]. Figure 2.15 outlines the concept of how plans can interact with the deployed service topology. For instance, node properties can be accessed and operations can be called to realize the plan's goal. Plans in TOSCA are discussed in [BBLS12] in more detail.

All the files that are referenced inside the service template such as scripts and binaries get packed together with the service template into a Cloud service archive (CSAR). The CSAR is completely self-contained, that is, it contains everything to deploy and manage the Cloud service that is specified by the included service template.

The software that is able to process Cloud service archives is referred to as a TOSCA container. In its most simple form, a TOSCA container consists of two main components: (1) the topology instance manager is in charge of instantiating the topology template. This is done by provisioning and managing the resources needed to realize the topology. (2) To process the plans attached to a particular service template, a workflow engine is needed. The following Subsection 2.4.2 presents a product that provides a TOSCA container.

2.4.2 IBM Common Cloud Stack

The IBM Common Cloud Stack [Dix12] is a stack of software components to enable model-driven Cloud management. The stack is based on several IBM products that support the management of Cloud applications. The main goal of the Common Cloud Stack is to consolidate the heterogeneous and overlapping Cloud management offerings of IBM on the levels of Infrastructure as a Service and Platform as a Service. Three offerings, built on one another, are based on the stack's modular architecture:

SmartCloud Entry offers basic Cloud management including an intuitive self-service interface. It is focused on the level of Infrastructure as a Service.

SmartCloud Provisioning is based on SmartCloud Entry and adds several features such as lifecycle management for virtual machine images and the possibility to deploy standardized middleware and applications. These standardized software components provide platform services to the application components built on top of them.

SmartCloud Orchestrator is still under development. Based on SmartCloud Provisioning, it is focused on enabling service orchestration.

Another important goal of the Common Cloud Stack is to provide a TOSCA container. The current container implementation is still a prototype. It is based on both the IBM Workload Deployer [IBM12c] and the IBM Business Process Manager [IBM12a]. The first one is in charge of managing the topology instances whereas the second one provides a workflow engine to process plans.

Specific plug-ins can be created to add functionality to the Common Cloud Stack. This possibility represents the extensibility model of the Common Cloud Stack. Furthermore, virtual application patterns can be created to build a topology consisting of nodes and relations. Each node and each relation is an instance of the implementation provided by a plug-in. The deployment of such a topology results in provisioning of several virtual machines that are necessary to realize the topology. The "Maestro Agent" component gets installed on each machine. This agent retrieves and executes scripts that are part of the plug-ins. Those scripts represent the actual implementation of the nodes and relations that are part of the topology.

The meta model of the Common Cloud Stack differs from TOSCA's meta model. As a result, a TOSCA importer component is needed inside the Common Cloud Stack. Its purpose is to transform the Cloud service archive including the service template into a model, which can be processed by the Common Cloud Stack. Figure 2.16 presents the simplified logical flow of processing and deploying a Cloud service archive. The TOSCA importer component generates one plug-in for each node type and each relationship type in the service template.

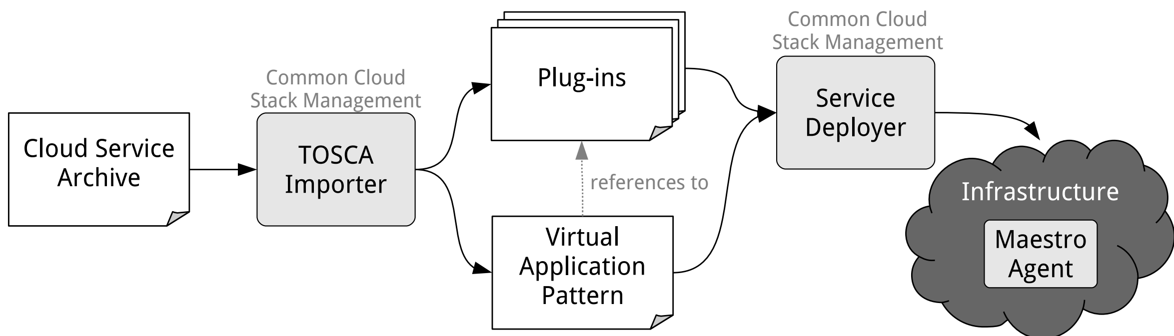

Figure 2.16: Process Service Template Using Common Cloud Stack: Simplified Logical Flow

The topology template gets translated into a virtual application pattern. This pattern holds references to the generated plug-ins that represent the node types and relationship types. In the second step, the service is being deployed using the service deployer component and based on the internal service model including the virtual application pattern and all plug-ins that are referenced in the pattern. Inside the target infrastructure the "Maestro Agent" is deployed on each virtual machine to install and configure software components. The agent processes the deployment artifacts and implementation artifacts of the node types and relationship types that are instantiated on a particular machine.

2.4.3 Canonical Juju

Juju [Can12c] is a framework, which provides service orchestration tooling. It was published by Canonical and is publicly available as open source software. The main goal of Juju is to provide a framework for sharing and reusing DevOps best practices in the form of self-contained units. Those shareable and reusable units are called "charms." A single charm represents a service model that can be instantiated. Thus, a charm is named appropriately to the service it represents. Examples for charm names are "mysql" or "wordpress." The Charm Browser [Can12a] enables access to charms that can be processed by Juju. The purpose of the Charm Browser is to offer a central repository, which can be populated and accessed by the community. Today, all charms in the Charm Browser are designed to be instantiated on a machine that has Ubuntu Linux [Can12d] installed. Currently, Juju supports all clouds that offer an OpenStack interface [Ope12] or an API that is compatible to Amazon Web Services [Ama12]. Support for more platforms and target clouds is in progress.

After instantiating at least two charms as two service instances in a target cloud, they can be connected by adding a relation. The prerequisite for adding a new relation between two services is that one provides an interface that the other requires. To scale a service up and down, additional service instances can be added and removed [Can12b]. These mechanisms to modify the topology by adding and removing relations as well as scaling up and down services makes Juju more powerful than configuration management tools introduced in Section 2.3. Thus, Juju is a framework to realize model-driven Cloud management.

Figure 2.17 outlines the structure of a charm. The charm's specification is described in the "metadata.yaml" file. It specifies the interfaces that are provided and required by this particular charm. The "config.yaml" file specifies the properties to configure a service instance based on this charm. In addition to those two files, several "hooks" are bundled with the charm [Can12b]. A hook is an arbitrary executable. It can, for instance, be a shell script, a Python script, or a helper script that calls a configuration management tool. The hooks implement the whole lifecycle of a service instance. When a new service instance is being instantiated, the "install" hook gets executed to install and configure the service instance. Then, the "start" hook is executed. When a new relation is being added between two service instances, the corresponding "relation-changed" hook gets executed for both service instances. As an example, when an application service instance connects to the "db" interface of an instance of the "mysql" charm outlined in Figure 2.17, the "db-relation-changed" hook is being executed. [Can12b] provides

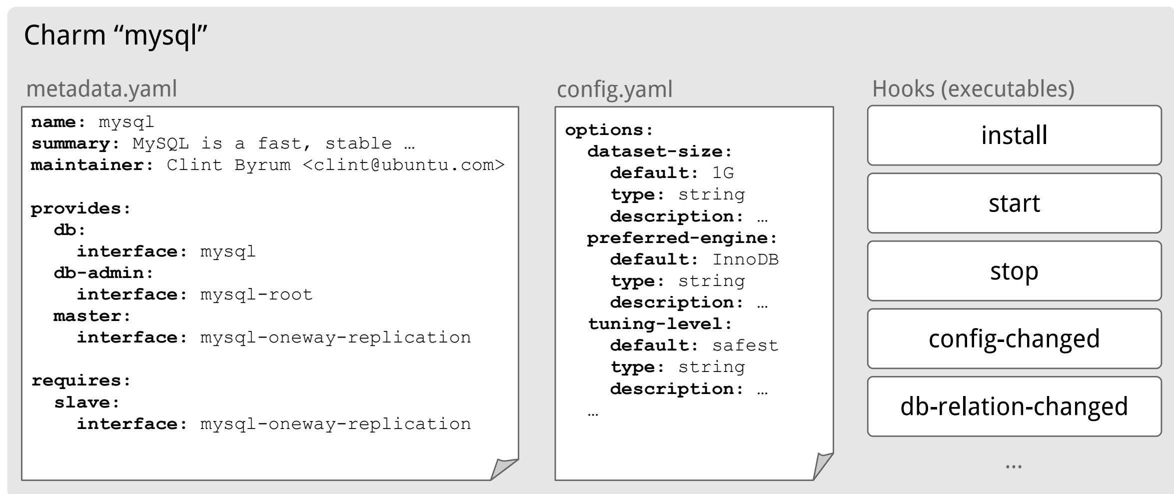

Figure 2.17: Structure of the "mysql" Charm [Can12a]

an overview of all hooks that can or have to be implemented. Some of them are mandatory, others are optional.

Even though the charms that are currently available through the Charm Browser are designed to be instantiated on a machine running Ubuntu Linux, portable charms can be created and instantiated. This is possible because there are no limitations regarding the hooks' implementation [Can12b]. Unfortunately, it is not possible to bundle more than one implementation for a particular hook with a charm. If Juju would support including several implementations for a single hook, it would be much easier to create portable charms. As of today, all the logic that is necessary to achieve portability has to be inside the hook's implementation.

Beside that fact, Juju and TOSCA share some similarities regarding their meta model to describe a service topology. Node types in TOSCA can be compared to charms in Juju. Node type operations and their implementation artifacts can be compared to hooks of a charm. Of course, TOSCA's meta model is much more generic and has a much stronger position when it comes to portability. Although Juju does not support TOSCA, those similarities can support the implementation of a TOSCA container based on Juju.

3 Integrating Configuration Management with Model-Driven Cloud Management

In Chapter 2, three key points are made by analyzing the state of the art: (1) today, DevOps methodologies are realized in practice using configuration management tools such as Chef, Puppet, or CFEngine. Those tools follow the Infrastructure as Code paradigm. (2) Pure configuration management does not support higher-level management aspects such as describing a service topology consisting of nodes and relations. For more complex scenarios it is getting very hard to understand and maintain the "infrastructure code." Thus, management on a higher level is needed. (3) As a result, approaches are emerging that follow the paradigm of model-driven management of services and their underlying infrastructure. Because complex service topologies are especially present in the Cloud, this paradigm is also referred to as model-driven Cloud management. For instance, TOSCA, the IBM Common Cloud Stack, and Juju are following that paradigm.

This chapter is the core part of this thesis. It outlines key concepts to integrate DevOps methodologies into higher-level management of services and applications. The main goal is to bring together the strengths of both worlds: configuration management and model-driven Cloud management. Section 3.1 and Section 3.2 outline the running example and a set of

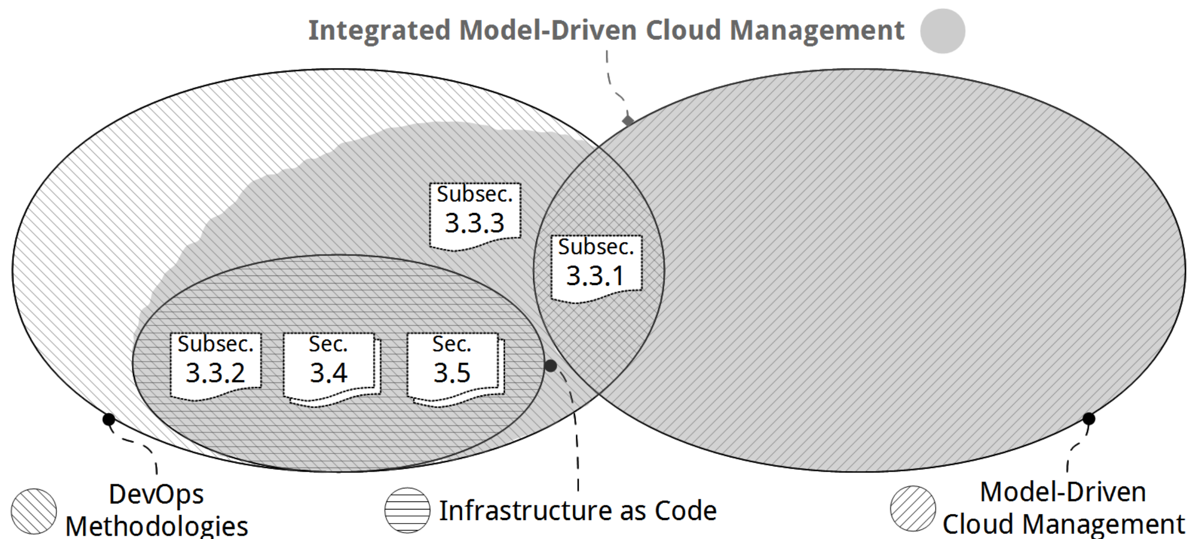

Figure 3.1: Concepts Described in Chapter 3 in a Formally Defined Scope

selected use cases. Those are the foundation for presenting the concepts. All the concepts that are outlined in this chapter are positioned inside the scope of "integrated model-driven Cloud management". Figure 3.1 presents how this scope is formally defined: originally, the intersection between the scope of DevOps methodologies and the scope of model-driven Cloud management is small. They share a few basic concepts only, such as the support for multiple environments as outlined in Subsection 3.3.1. Because this thesis is about integrating DevOps methodologies with model-driven Cloud management, the overall result is to extend the original scope of model-driven Cloud management. This extended scope is referred to as "integrated model-driven Cloud management" in Figure 3.1, shaded in gray. The scope of Infrastructure as Code is a proper subset of the scope of DevOps methodologies.

In addition, the scope of Infrastructure as Code is a proper subset of the "integrated model-driven Cloud management" scope: Subsection 3.3.2 presents concepts how to integrate configuration management artifacts that follow the paradigm of Infrastructure as Code into a TOSCA-based Cloud service archive. Then, Section 3.4 is about creating TOSCA models for existing services that are based on configuration management tooling, whereas concepts of how to realize a TOSCA container that fits the DevOps paradigm are presented in Section 3.5.

Outside the scope of Infrastructure as Code, Subsection 3.3.3 outlines how management plans are related to configuration management and why they are not in the scope of Infrastructure as Code. As a remark, the scope of DevOps methodologies is not completely included in the scope of "integrated model-driven Cloud management." This is because concepts and practices are part of the scope of DevOps methodologies that do not fit the paradigm of model-driven Cloud management. As an example, most of the products established in the DevOps world such as Chef and Juju provide rudimentary tool support by intention. Those tools are often designed to be used as plain command line tools [Ops12c, Can12b]. However, model-driven Cloud management is focused on providing graphical tooling to manage the models that can get complex.

3.1 The Running Example: Sugar as a Cloud Service

In order to make the examples in the following sections consistent, a running example is introduced. The whole scenario is based on Sugar Community Edition, a popular customer relationship management system developed by SugarCRM [Sug12]. It is a web application based on PHP [The12b] and a SQL database. The community edition of Sugar is publicly available as open source software.

Figure 3.2 presents an overview of the running example. There are four different roles involved:

The service creator is in charge of creating and maintaining the Cloud service archive. After the CSAR has been created, the service creator can publish it, for instance, on a marketplace.

The service provider retrieves the Cloud service archive. After retrieving the CSAR, the service provider customizes and deploys a service instance to the infrastructure. The service provider continuously manages the service instance.

The service consumer can access and use the service instance once it has been deployed by the service provider.

The Cloud provider is in charge of providing the infrastructure to where a Cloud service archive can be deployed.

The logical service topology of the running example is outlined in Figure 3.3. Two virtual machines are the foundation of providing Sugar as a Cloud service. The Sugar database is hosted on a MySQL server, which itself is running on top of the operating system. The operating system running on the other virtual machine hosts the Apache Web server [The12a]. On top of the Web server, the Sugar application is hosted. In addition, the Sugar application depends on the PHP module, which is hosted on the Web server, too. At the top level, the Sugar application is connected to the Sugar database. The topology outlined here is typically referenced when giving examples in this thesis.

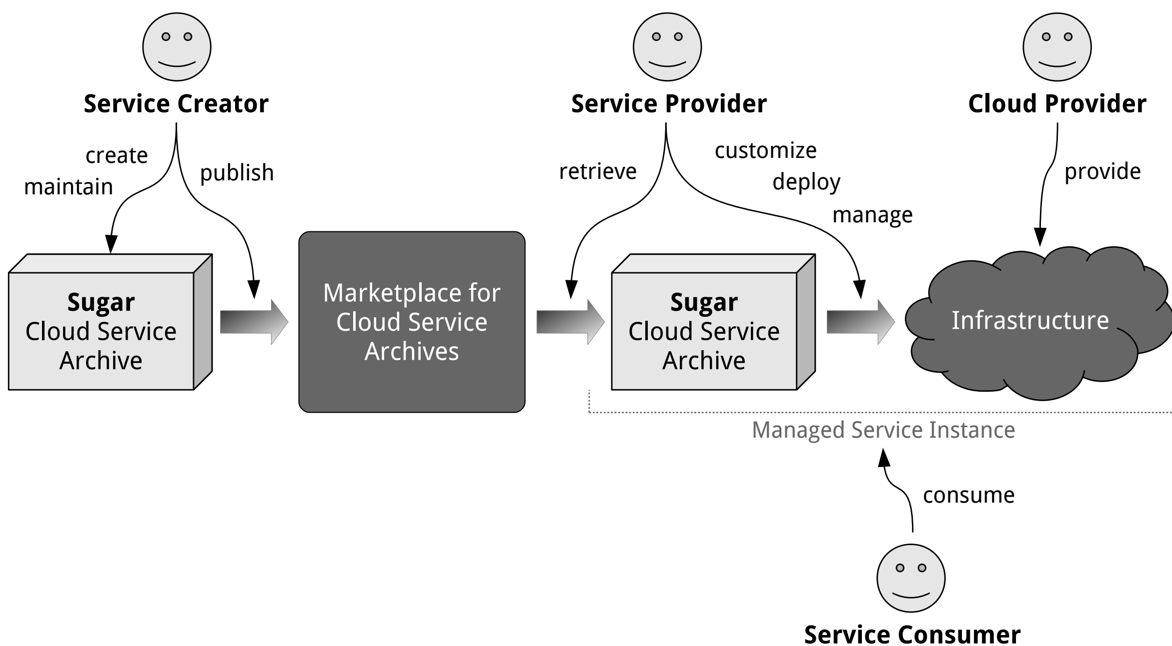

Figure 3.2: Sugar as a Cloud Service: Overview, Roles, and Dependencies

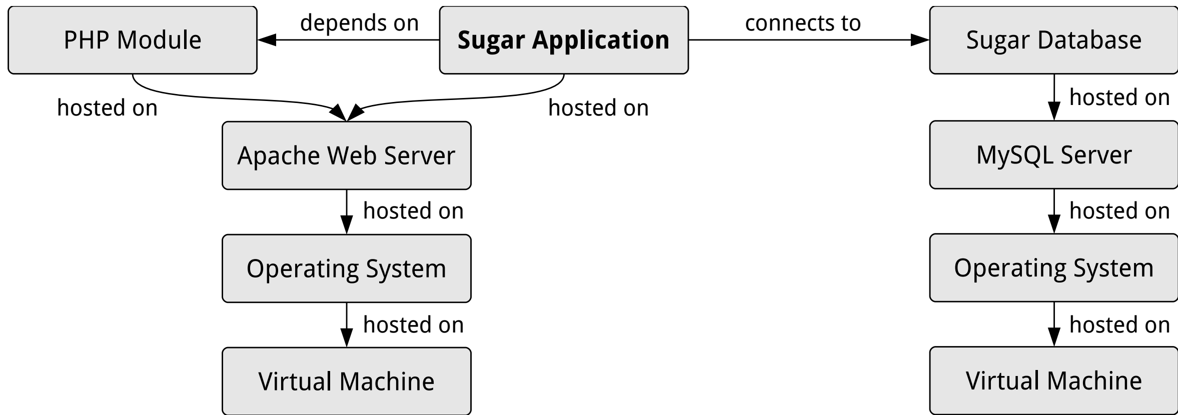

Figure 3.3: Logical Topology of the Sugar Cloud Service

3.2 Use Cases Based on the Running Example

To achieve the goal of integrating DevOps methodologies with model-driven Cloud management, this thesis is focused on the following use cases that are based on the running example outlined in Figure 3.2:

Creating and maintaining a Cloud service archive: The challenge is to create a CSAR, which is portable. As a consequence, specific assumptions regarding the target infrastructure should not be made inside the CSAR. The service topology and all management aspects should be specified in detail to enable a high degree of management automation. In addition, the Cloud service archive has to be well structured in order to be maintainable. This is especially important for upgrading the CSAR and upgrading service instances based on a particular CSAR.

Customizing a Cloud service archive: The CSAR has to be customizable to enable tailoring a service instance to the actual requirements. For instance, three differently tailored service instances may be created based on the same CSAR to provide the service in three different environments: development, test, and production.

Deploying a service instance based on a Cloud service archive: It should be possible to instantiate a service based on a particular CSAR in the Cloud. The CSAR should not prescribe on which specific Cloud infrastructure the deployment takes place. In addition, the CSAR should be self-contained, meaning that everything that is necessary to create an instance is part of the Cloud service archive.

Managing the deployed service instance: After a service instance has been deployed, it can be accessed and used by a service consumer. But the service instance needs to be managed continuously. Examples for management tasks are doing a database backup or scaling the service instance up and down.

Those use cases were identified together with experts at IBM based on actual customer requirements and real-world scenarios.

3.3 Bringing DevOps Methodologies into TOSCA

TOSCA is an emerging standard that enables model-driven Cloud management. However, it does not directly support DevOps methodologies. This section presents concepts to enable support for those methodologies in TOSCA.

3.3.1 Support for Multiple Environments

As outlined in Subsection 2.1.2, supporting several environments such as development, test, and production is a typical use case for applying DevOps methodologies. This is due to the requirement that the developers should have access to nearly the same kind of environment than the operations personnel. That is, creating multiple instances of the same service or application has to be a fast and highly automated process. Configuration management tools such as Chef or Puppet support the management of a service that is deployed in different environments. Those tools can create multiple instances of a service based on different environment configurations. Thus, a particular service instance can be customized with regard to the actual environment where it is running. For instance, the debugging option of the Web server in the Sugar scenario is enabled for the development and test environments; but it is disabled for the production environment. However, the environment-specific configuration of a service instance is limited to the scope of single nodes or sets of nodes as outlined in Section 2.4. This is due to the limitations of pure configuration management compared to model-driven Cloud management. An example for these limitations regarding the Sugar scenario could be the following: a single database node is fine for the development and test environment. However, the production environment needs a cluster of database nodes. That is, the service topology differs for the different environment. This is not supported by pure configuration management tooling. Thus, model-driven Cloud management is needed to support those kinds of scenarios.

Subsection 2.4.1 presented some of the key concepts of TOSCA, namely the definition of a service template including node types, relationship types, and a topology template consisting of node templates and relationship templates. In addition to those concepts, TOSCA supports the expression of requirement types and capability types. These types can be attached to a node type as requirement definition or capability definition. As a result, a particular node template has to meet certain requirements or provides capabilities, depending on the related node type definition.

An example of how requirements and capabilities can be used in a service template is outlined in Figure 3.4. Each requirement in the topology template has to be fulfilled by a corresponding capability. To actually fulfill a requirement, a relationship has to be created, which connects the requirement to its corresponding capability. Such a relationship can either be part of the topology template or it can be created when deploying a service instance.

Figure 3.4: Example for Using Requirements and Capabilities in TOSCA [Org12b]

Requirements and capabilities cannot only be functional ones. Non-functional requirements and capabilities such as scalability can be expressed, too. This mechanism enables TOSCA to support the DevOps scenarios presented before. The service template outlined in Figure 3.4 could be extended by adding a “Scalable Sugar Database Node Type” and a “Scalable Capability Type.” A corresponding “Scalable Capability Definition” is attached to the newly added node type. By doing this, there are two candidates to fulfill the “Sugar Database Requirement” of the “Sugar Application Node Type.” These are the “Sugar Database Node Type” and the “Scalable Sugar Database Node Type.” The former one would be appropriate to be used in a development or test environment, the second one could meet the needs of a production environment, because it implements a database cluster. The decision, which node type to choose can be made at deployment time depending on the actual requirements of the environment.

3.3.2 Natural and Transparent Integration of Configuration Management

The principles and limitations of configuration management were presented in Section 2.3. In addition, it was stated why configuration management tooling is used to realize DevOps methodologies. To overcome the limitations of configuration management, model-driven Cloud management was introduced in Section 2.4. However, the paradigm of configuration management targets several aspects that are not in the scope of model-driven Cloud management. A typical example for such an aspect is how to specify lower-level tasks in management scenarios [GHS10]. The specification of those tasks such as installing and configuring software components on a virtual machine should be simple and platform-independent. This is why integrating configuration management with model-driven Cloud management makes sense and provides additional value.

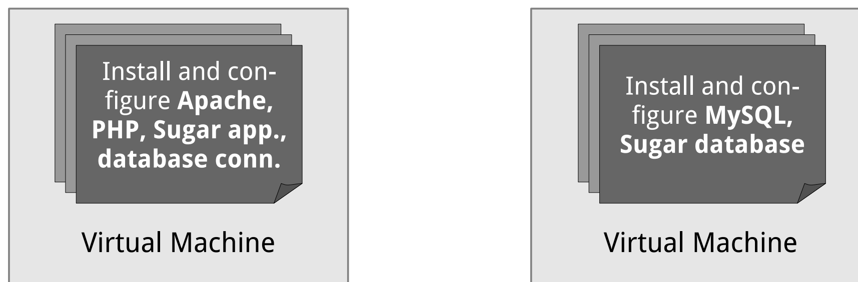

Figure 3.5: Deployment of the Sugar Cloud Service Using Configuration Management

As an example, Figure 3.5 presents how the Sugar Cloud service is deployed using a configuration management tool. There are two virtual machines provisioned. On the one machine, the Apache Web server, the PHP module, and the Sugar application get installed and configured by automatically running artifacts. The MySQL server and the Sugar database get installed and configured on the other machine. To finish the deployment, the application gets connected to the database. To sum it up, there are the following artifacts involved to deploy the service:

- Artifact to install and configure the Apache Web server
- Artifact to install and configure the PHP module
- Artifact to install and configure the MySQL server
- Artifact to install and configure the Sugar application
- Artifact to install and configure the Sugar database
- Artifact to set up the connection between the application and the database

Those artifacts are portable, configurable, and easily customizable [GHS10]. The focus of model-driven Cloud management is to define a topology model, which brings those artifacts in relation to one another as well as to define management plans. Figure 3.6 outlines the topology model for the Sugar Cloud service with artifacts attached to install and configure the software components that are running on top of the virtual machines' operating system. However, model-driven Cloud management does not have in its scope how to implement those artifacts. The straightforward approach is to implement shell scripts to install and configure software components. Because shell scripts are intended to be used to perform simple tasks on a specific platform, using a platform-independent scripting language such as Python or Ruby would be a better choice to create portable artifacts. Another two difficulties still remain: (1) it is hard to maintain the scripts because those scripting languages are general-purpose languages and do not directly support the domain of software installation and configuration. (2) It is hard to design the scripts in a way so that they can be ported to additional platforms without much effort. Those issues are the motivation for implementing the artifacts using configuration management tooling.

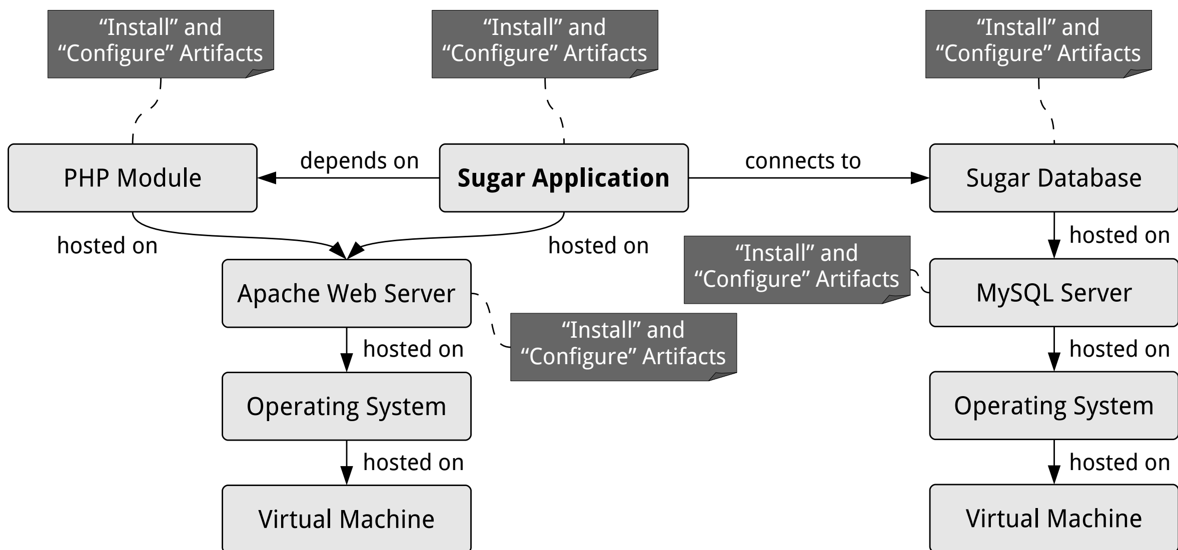

Figure 3.6: Sugar Cloud Service Topology with "Install" and "Configure" Artifacts Attached

In addition to the advantages outlined regarding portability and maintainability, configuration management products offer tools and mechanisms to ease the management of service topologies. As an example, additional artifacts can be assigned to nodes or roles owned by at least one node in the topology. Typical examples for activities realized by additional artifacts are updating an application, doing a database backup, or customizing the configuration of a deployed application.

For the popular configuration management tools such as Chef or Puppet, there are artifacts already available to install, configure, and manage many software components [Ops12a, Pup12d]. Consequently, they can be easily reused and combined by creating a service topology model with configuration management artifacts attached.

In the following paragraphs two different approaches are discussed to include configuration management artifacts in a service topology model. Those are (1) the natural integration and (2) the transparent integration. Furthermore, it is outlined how to combine those two approaches to benefit from the strengths of both.

The natural way of embedding configuration management artifacts into a service topology model is to define an artifact type, which fits the structure of the artifacts that are created based on a particular configuration management tool. For instance, a Chef artifact type needs to be defined to embed Chef cookbooks into a topology model. In TOSCA, there are standard implementation artifacts available such as script artifacts to include simple scripts. However, arbitrary types of implementation artifacts can be defined by creating an appropriate XML schema definition [Wor12b]. A TOSCA container that processes a service template including custom implementation artifacts then has to understand the corresponding artifact type.

Configuration management artifacts can also be embedded into a service topology model in a transparent way. As a result, the processor of a topology model does not have to understand custom artifact types. In practice, this approach of integrating configuration management can be realized by creating "wrapper scripts" that call the configuration management tool with corresponding parameters and references to the configuration management artifacts. Those wrapper scripts can be attached to the topology model using standard artifacts such as script artifacts in TOSCA. However, some constraints such as the configuration of the agent that processes the configuration management artifacts are hard-wired inside the wrapper scripts. The result is a loss of flexibility to process configuration management artifacts. In addition, it is hard to make the wrapper scripts portable. As a consequence, the transparent integration approach is not intended to be used solely. What makes much more sense is to combine it with the natural way of integrating artifacts. Then, the processor of a topology model can decide whether to process the standard artifacts while having less flexibility or to process the custom artifacts if the corresponding artifact type is understandable for the particular processor.

An example for the ideal way of combining the natural and transparent integration approach is presented in Figure 3.7. The Sugar Cloud service archive follows the natural way of integration and contains custom artifacts only, which are completely portable. This CSAR can be seamlessly processed by a TOSCA container that understands those custom artifacts. For TOSCA containers that do only understand standard artifacts, a preprocessor automatically enriches the CSAR with "wrapper artifacts" that are based on standard artifact types. Full flexibility can be retained by making the preprocessor component highly configurable: there are no hard-wired constraints included in the original Sugar Cloud service archive; the generated wrapper artifacts can be tailored to be processed by a specific TOSCA container.

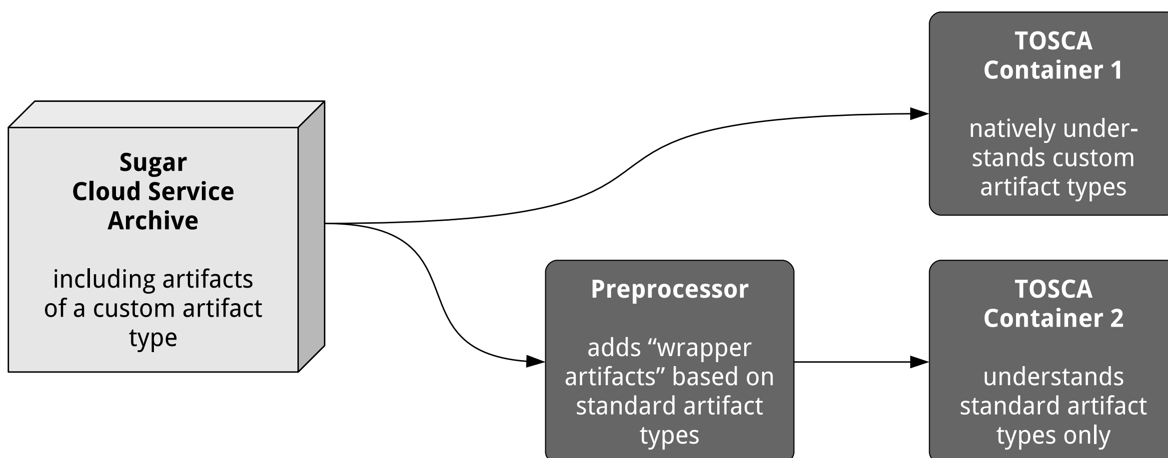

Figure 3.7: TOSCA-Based Example for How to Process Custom Artifacts Ideally

3.3.3 TOSCA Plans and Configuration Management

Management of services is not only about modeling the service topology. Especially for management tasks taking place after the actual deployment, the paradigm of model-driven Cloud management includes the idea of having executable process models to cover management tasks such as updating an application or doing a database backup. A typical example for those process models are management plans in TOSCA [Org12b]. [BBLS12] outlines how to make Cloud services portable using management plans. This is also referred to as the imperative approach of realizing model-driven Cloud management: management plans specify all management aspects of a Cloud service including its deployment in an imperative manner. The counterpart of the imperative approach is the declarative approach: the service model mainly consists of a service topology specification that is interpreted by the service management software. Both approaches can be combined by automatically generating management plans based on a declarative service model. This may work for generating a “build plan,” but it may be very hard or even impossible to generate plans for performing more sophisticated management tasks.

The concepts presented in this thesis are strongly focused on the declarative approach without generating management plans. This is because the DevOps paradigm does not realize management tasks by running predefined process models. Popular configuration management products such as Chef [Ops12d] or Puppet [Pup12e] do not include support to create and execute process models. Their way of realizing service management is to provide configuration management artifacts that can be assigned to at least one node. This approach may be fine for simple management tasks such as updating a package or applying a patch on a particular set of nodes.

However, when management tasks become more complex, including subtasks on different nodes that are depending on one another, a process model is needed. An example for such a complex management task could involve the following subtasks:

1. Stop the application instance
2. Stop the database instance
3. Backup the database content
4. Update the application instance
5. Start the database instance
6. Start the application instance

The subtasks can be implemented using configuration management artifacts, but for the management task as a whole, a process model needs to be defined. This is again because there is no topology model available on the configuration management level. Therefore, the process model specifies the dependencies between the subtasks as well as on which nodes which artifacts should be executed. Thus, process modeling languages such as BPMN [Obj12] or BPEL [Org12c] are candidates for creating such a process model.

To ensure portability of a TOSCA service template and the plans that are included, a plan should not reference a configuration management artifact directly. Instead, a plan should reference a node type operation such as “backup-database” that has a corresponding implementation artifact attached. Then, for instance, a wrapper artifact based on a standard artifact type can be assigned to a particular operation. By doing this, the plan does not have to be modified and can be executed in a TOSCA container that does not understand the originally attached configuration management artifacts. This is why the TOSCA technical committee [Org12a] is discussing about introducing the “Plan Portability API.” This ensures that plans can access an API that is provided by each TOSCA container.

3.3.4 Evaluation

This section discussed key concepts to bring DevOps methodologies into TOSCA. Subsection 3.3.1 covered the use case of deploying a service instance based on a Cloud service archive. It was outlined how to enable service deployment for multiple environments. The use cases of creating, maintaining, and customizing Cloud service archives were covered by Subsection 3.3.2, whereas Subsection 3.3.3 was focused on the use case of managing a deployed service instance. As a result, TOSCA benefits in several ways:

1. Those concepts are following the design principles of TOSCA and their implementation does not need any extension of TOSCA itself.
2. Multiple environments are supported by defining requirements and capabilities in a service template. Thus, there are no special requirements for the TOSCA container to realize this concept.
3. The integration of configuration management artifacts is the foundation for creating portable Cloud service archives. The natural integration of those artifacts is the preferred way. If a particular TOSCA container cannot process the corresponding type of artifacts, the transparent integration approach can be chosen. Implementing a CSAR preprocessor in that case would be an elegant solution.
4. There are many configurable configuration management artifacts publicly available that can be reused. Those can be included into a CSAR without modifying them.
5. Configuration management artifacts can also be used to perform simple management tasks. For more complex tasks those artifacts can be used in plans.

Some of the drawbacks of integrating configuration management with TOSCA are:

1. A small but additional layer gets introduced. Instead of integrating scripts as script artifacts that get directly called on the virtual machine, configuration management artifacts are included as implementation artifacts. For the natural integration approach the TOSCA container has to understand the custom artifacts in order to process them. For the transparent integration approach a preprocessor component has to be implemented. That is, in both cases additional complexity is added to the implementation to process configuration management artifacts.

2. Configuration management artifacts are created using a domain-specific language [GHS10]. Consequently, that specific language has to be learned when manually creating such an artifact.

This evaluation shows that it makes sense to embed configuration management artifacts into a Cloud service archive. In most cases, the benefits outweigh the drawbacks of realizing the concepts outlined in this section.

3.4 Creating TOSCA Models for Existing Services

Concepts for integrating configuration management artifacts into a Cloud service archive were discussed in Section 3.3. However, as of today there are many services managed using plain configuration management tooling [Ops12b, Pup12b] without applying the paradigm of model-driven Cloud management. The following subsections are focused on concepts to move running services into an environment that makes them manageable using the paradigm of model-driven Cloud management. The core part is about creating a Cloud service archive for a service, which is managed by configuration management tooling.

3.4.1 Model Creation Steps

Services that are managed using configuration management tooling usually utilize the management infrastructure provided by the corresponding configuration management product as outlined in Section 2.3. The assumption made here is that there is a central management component, which is connected to the agents running on the nodes. An agent retrieves the artifacts that it has to execute on the particular node where it is running and executes them. Thus, the central management component has to have a global view including the knowledge which artifacts are assigned to which nodes. Based on those observations regarding the architecture of configuration management tooling the following steps were identified to create a TOSCA-based service model for an existing service. These steps represent a generic approach to create a Cloud service archive including a TOSCA service template:

(1) Determine node type candidates: Usually, the configuration management artifacts to deploy and manage a particular software component are grouped in a container. For instance, cookbooks in Chef are such a container mechanism [Ops12c]. A cookbook consists of at least one recipe. These are the actual configuration management artifacts. For each container that includes an artifact, which is assigned to at least one node, a node type candidate is created.

(2) Create node type definitions: For each node type candidate determined in the previous step, a node type definition including the lifecycle operation “create” is created.

(3) Attach implementation artifact to each “create” operation: An implementation artifact is created for each “create” operation. The implementation artifact includes references to at least one configuration management artifact. Which artifacts are

referenced in which order is determined based on the assigned artifacts. For instance, if the two configuration management artifacts “mysql-install” and “mysql-configure” are assigned to a node, the implementation artifact references those two artifacts in the given order.

(4) Create topology template: For each node a stack of node templates is created. The stack consists of a “virtual machine” node template, an “operating system” node template on top, and more node templates on top based on the node types created in step two. Which node templates are added to the stack is determined based on the configuration management artifact assignments of the particular node. Inside the stack, each node template is connected to its underlying neighbor by a “hosted on” relationship template. This relationship template is directed from the node itself to its underlying neighbor. The “virtual machine” node templates are exceptions from this rule: they do not have an underlying neighbor, so there is no “hosted on” relationship template starting from a “virtual machine” node template.

(5) Optionally, refine topology template: The topology template can be refined by including additional relationship templates such as “depends on” or “connects to” to make the topology template more appropriately representing the actual service topology model. This step is optional because the basic topology template created as part of the previous step already represents the service topology model correctly. However, some aspects of the actual model are missing in the topology template. As an example, a connection between an application and its database may not be represented in the basic topology template created as part of step 4. In that case, a “connects to” relationship can be added to refine the topology template. Additional node types and node templates can be included, too.

(6) Optionally, create management plans: Based on the configuration management artifacts, plans can be added to the service template to specify complex management tasks as outlined in Subsection 3.3.3. This step is optional because in case of following the declarative approach described in Subsection 3.3.3, a service model does not need to contain management plans at all.

Those steps can be performed automatically, semi-automatically, or manually. In a real-world scenario, the data would have to be migrated separately. The model creation steps outlined before are focused on migrating the service structure including the topology. The result is a self-contained Cloud service archive, which can be deployed and managed using a TOSCA container.

3.4.2 Proposal for a Semi-Automatic Creation Procedure Based on Chef

Based on the generic model creation steps outlined in the previous subsection, this subsection presents a proposal for a semi-automatic creation procedure. The procedure is based on Chef to make it more concrete. However, the procedure can be adapted to other configuration management products such as Puppet, as long as their architecture is similar to Chef’s architecture. In addition, based on the current Common Cloud Stack implementation it is

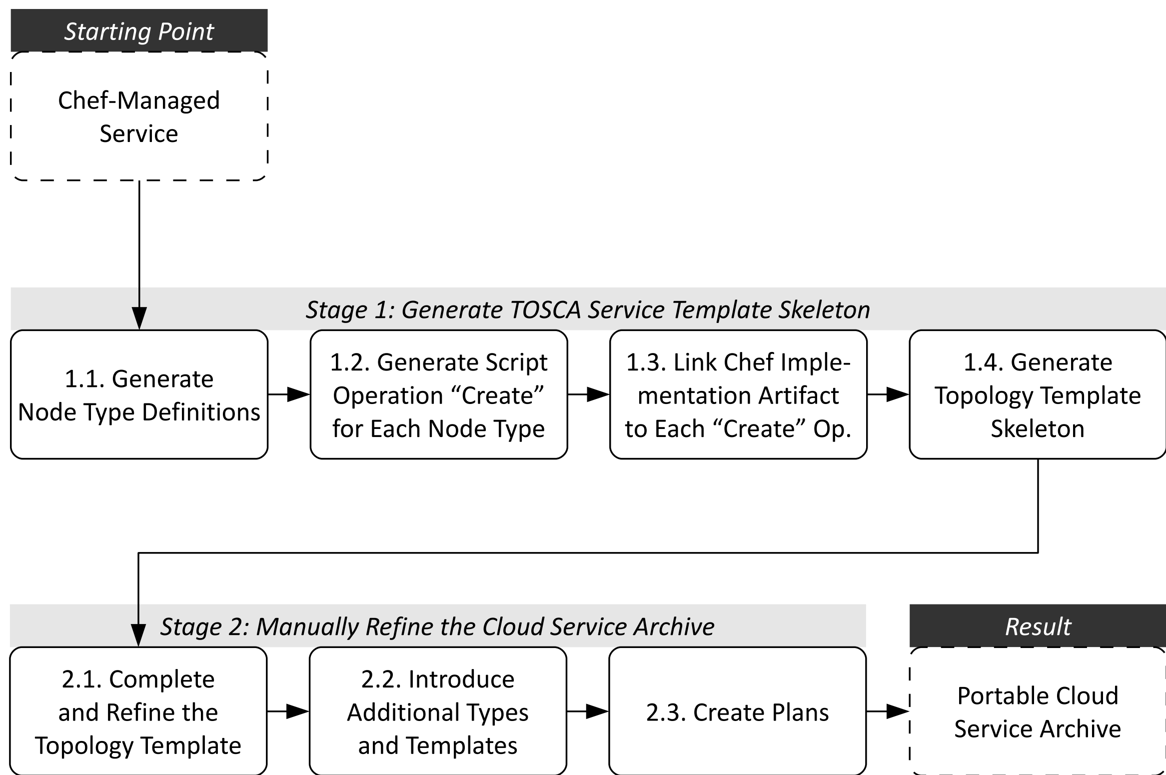

Figure 3.8: Overview of the Semi-Automatic Model Creation Procedure

assumed that the following node types and relationship types are predefined as TOSCA core types:

Node type "virtual machine:" A node template based on this type represents an instance of a virtual machine.

Node type "operating system:" A node template based on this type represents an instance of an operating system running on a virtual machine.

Relationship type "hosted on:" A relationship template based on this type means that the source node template is hosted on the target node template.

Relationship type "connects to:" A relationship template based on this type means that the source node template establishes a connection to the target node template.

Relationship type "depends on:" A relationship template based on this type means that the source node template depends on the target node template.

An overview of the whole semi-automatic creation procedure is presented in Figure 3.8. The individual steps of the procedure's first and second stage are based on the model creation steps outlined in the previous subsection. Starting from a service that is managed using Chef, the first stage of the procedure is about automatically generating a TOSCA service template

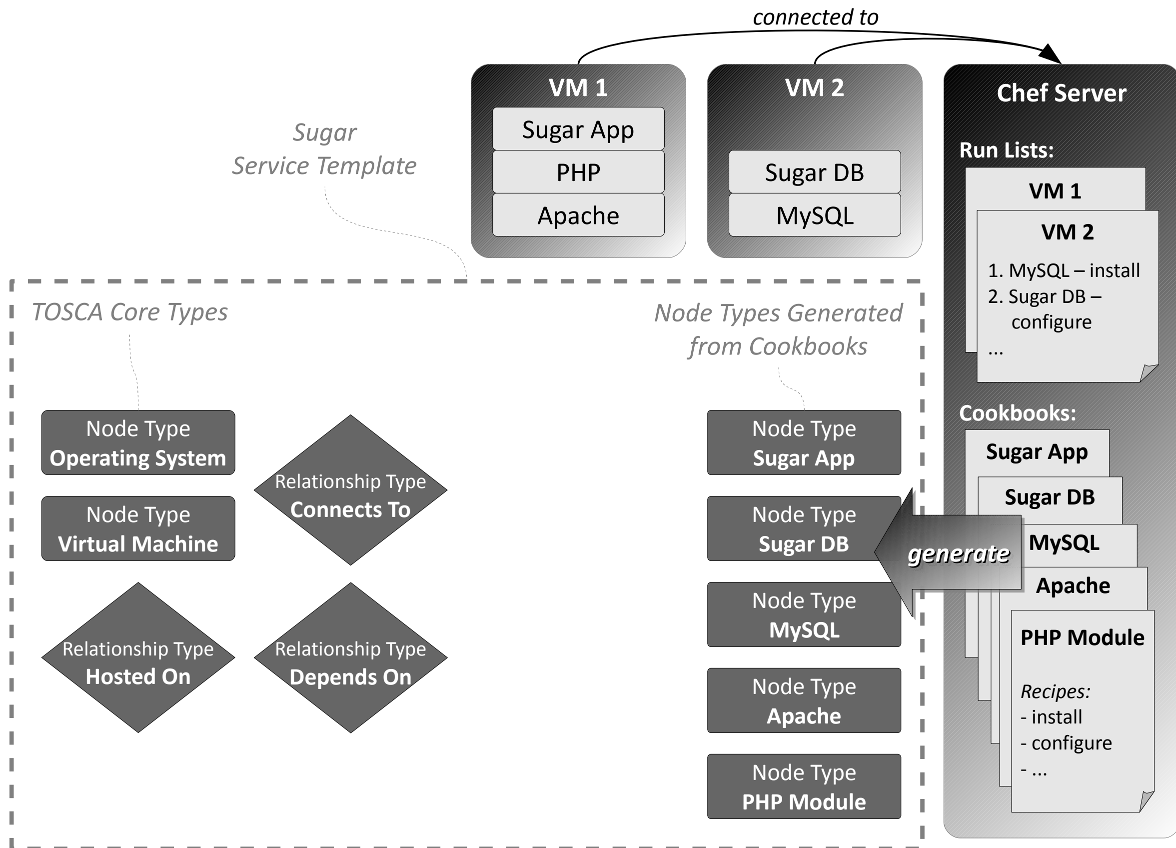

Figure 3.9: Example for Step 1.1: Generation of Node Type Definitions

skeleton. The second stage's goal is to manually refine the Cloud service archive. The outcome of applying the procedure is a portable CSAR. The following paragraphs explain the individual steps and provide examples for them. Those examples are based on the idea of creating a CSAR based on an existing Sugar service instance that is managed using Chef. The goal is to create the Sugar topology model that is outlined in Section 3.1.

In step 1.1, a node type definition for each cookbook is generated that fulfills the following prerequisite: there is at least one of its recipes referenced in at least one run list on the Chef server. Figure 3.9 presents an example for executing step 1.1. The node types that are generated in this step do not have any operations or implementation artifacts attached. This happens in the following two steps. Step 1.2 is about creating the lifecycle operation "create" for each generated node type. Then, a corresponding implementation artifact for each "create" operation gets generated and is linked to the operation definition in step 1.3.

The implementation artifacts generated in step 1.3 contain references to Chef recipes. The run lists on the Chef server are used to determine which roles and recipes are in charge of installing and configuring an instance of the particular node type. As an example, there may be a node that has the recipes "install-server" and "configure-server" in its run list. Both

of those recipes are part of the "MySQL" cookbook. Based on these information, those two recipes would be referenced inside the implementation artifact in the given order.

In step 1.4, the topology template is generated. It consists of node templates based on the generated node types and TOSCA's core node types as well as "hosted on" relationship templates between particular node templates. For each node that is registered at the Chef server, a "virtual machine" node template is being created. On top of each "virtual machine" node template, an "operating system" node template is being generated. Afterward, a "hosted on" relationship template is being created for each pair of "virtual machine" and "operating system" node template. Then, based on the run lists and the generated node types, more node templates and "hosted on" relationship templates are getting created on top of the operating system layer. Appendix A presents a piece of Java-oriented pseudo code to show the logic of step 1.4 in detail. An example for performing this step is presented in Figure 3.10.

Step 2.1 is the first one in the procedure's second stage. Its purpose is to refine the topology template based on the existing node types and relationship types. In contrast to the previous steps, this step cannot be performed automatically. The reason why it has to be performed manually is that the necessary information cannot be modeled using Chef. Thus, a human

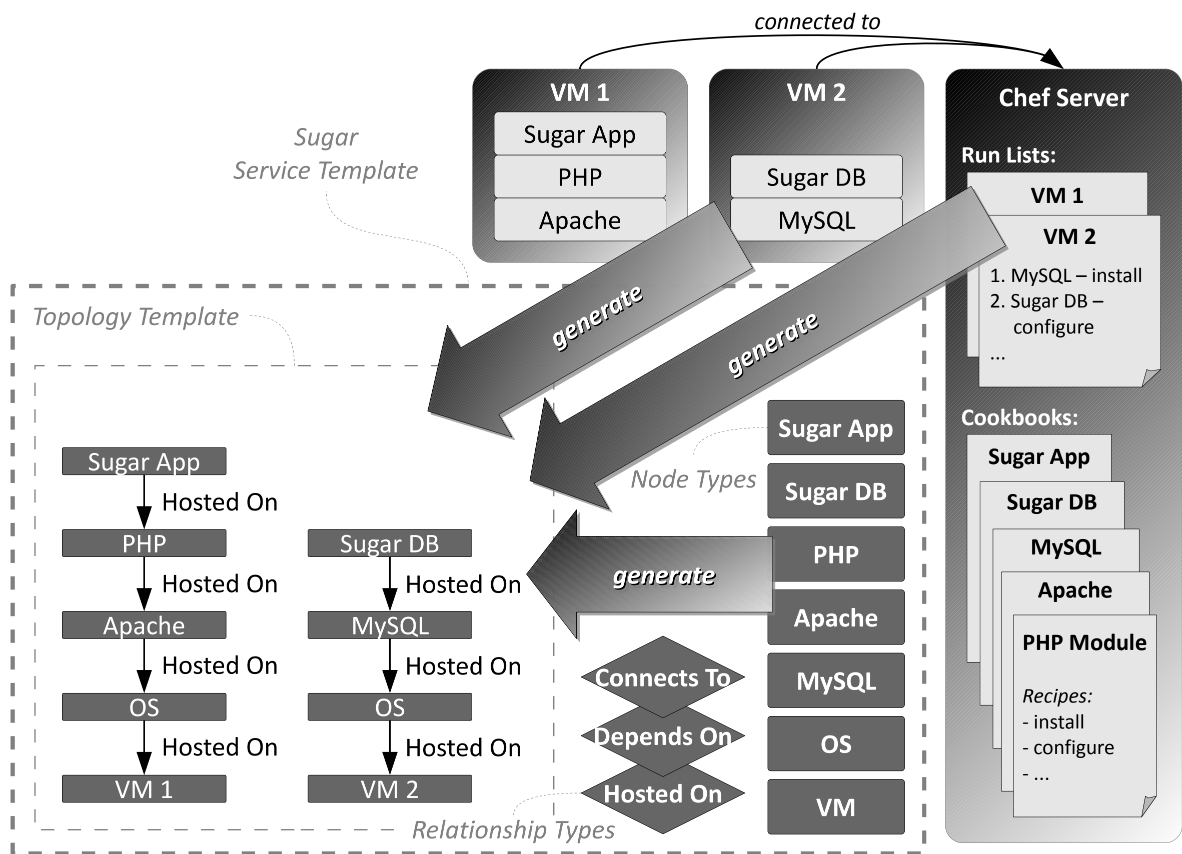

Figure 3.10: Example for Step 1.4: Generation of the Topology Template

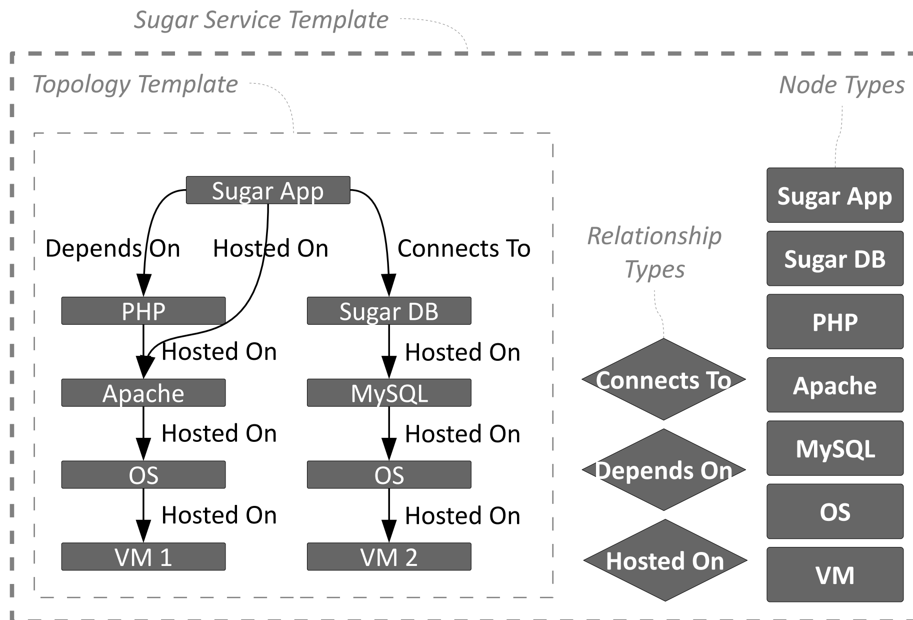

Figure 3.11: Example for Step 2.1: Refinement of the Topology Template

needs to refine the topology template by putting additional information into the generated model to make the model more appropriate to represent the logical service topology. Typical use cases for such a refinement are:

- Rearranging the "hosted on" relationship templates that were generated in step 1.4
- Including additional relationship templates such as "depends on" or "connects to"

Figure 3.11 outlines an example for refining the topology based on those two use cases: the Sugar application is not hosted on the PHP module what the generated model in Figure 3.10 says. In fact, the Sugar application is hosted on the Apache Web server and depends on the PHP module. Furthermore, the Sugar application establishes a connection to the Sugar database.

In step 2.2, additional node types and relationship types can be included. Existing types can be modified, for instance, by introducing additional operations and implementation artifacts or by modifying the implementation artifacts that were generated before. The topology template can be further refined based on those modifications and additions.

Management plans can be included into the Cloud service archive in step 2.3. For instance, a plan can be created that specifies how to update the Sugar application. In the end, the service template and all artifacts that are referenced in the service template get packed together in the Cloud service archive. This is the result of applying the semi-automatic model creation procedure: a portable CSAR.

3.4.3 Evaluation

Subsection 3.4.1 presented a set of steps to create a holistic TOSCA-based model for a service that is currently managed by configuration management tooling. The use cases of creating and customizing a Cloud service archive were covered by this approach. In fact, several benefits are related to moving from plain configuration management up to model-driven Cloud management:

1. There is a holistic model that specifies the service topology and the management aspects. Using plain configuration management, the model is limited to assignments of certain artifacts to particular machines. Relationships and complex management tasks cannot be modeled explicitly. There are no means to express complete topologies and management plans.
2. Because the management aspects are also part of the model, a service can be managed more efficiently compared to plain configuration management. The reason is that the management plans are placed inside the model and not documented separately.
3. The whole model can be packed together into a Cloud service archive. Using that self-contained CSAR, the service can easily be deployed and managed several times, in different environments, and on different infrastructures.
4. The lower-level aspects of the service can still be perfectly modeled using configuration management. An example for such an aspect is the realization of implementation artifacts. As a result, most of the artifacts and knowledge regarding configuration management remains valid. Model-driven Cloud management is focused on covering all the aspects that cannot be covered by plain configuration management. Thus, those two management approaches can be seen as complements.
5. Subsection 3.4.2 proposed and described a procedure to semi-automatically create a Cloud service archive based on a running service instance that is managed by configuration management tooling. Such a procedure can accelerate the migration toward model-driven Cloud management because several steps can be performed automatically. Others can be assisted with tool support.

Changing the management paradigm of a running service from plain configuration management to model-driven Cloud management leads to the following consequences:

1. Additional management components are introduced to realize higher-level management such as managing relationships and running management plans. This makes the stack of management software components more complex.
2. Consequently, the service provider, who is in charge of deploying and managing the service instances has to change his internal processes and procedures to manage such a service. This is because managing a service based on plain configuration management is very different from managing a service based on model-driven Cloud management.

In addition, there are two constraints regarding the semi-automatic procedure outlined in Subsection 3.4.2:

1. Operations and implementation artifacts are generated for node types only. From an ideal model's perspective, some of the logic should be better placed inside implementation artifacts of relationship types. As an example, the Apache Web server and the PHP module are installed on a particular node. To enable the Apache Web server to use the PHP module, the small Apache add-on "mod_php" is needed. To install and configure this add-on, the "Apache" cookbook includes the additional recipe "mod_php." In step 1.3 of the semi-automatic procedure, this recipe is included into the implementation artifact of the "Apache" node type. For an ideal model, it should be placed inside an implementation artifact attached to the "hosted on" relationship between the Apache Web server and the PHP module. This is because "mod_php" should be installed only in case a PHP module is hosted on an Apache Web server instance.
2. Generating one node type for each cookbook that is involved does not seem to make sense in all cases. The model could get polluted by node types and node templates that should not be modeled separately. As an example it is assumed that there is a "helper" cookbook that has a recipe "update-repo-cache" to update the software repository cache on a Linux-based machine. This recipe is assigned to each node in the first place of their run list. As a result, a "helper" node type is generated in step 1.1 and a "helper" node template is put on top of each "operating system" node template in the topology template. Thus, the generated model does not represent the service topology appropriately.

Of course, the first stage of the model creation procedure can be extended to deal better with more sophisticated cases. On the other hand, it depends how often such special cases occur in practice. Sometimes it may be faster to manually refine the aspects that make the model inappropriate.

3.5 Realizing a TOSCA Container Fitting DevOps Methodologies

The principles and methodologies of DevOps were outlined in Subsection 2.1.2. The philosophy behind the DevOps movement is to bring agile methodologies into the world of IT operations [HF10]. This makes the collaboration between the developers and the operations personnel much more flexible. To enable this flexibility, most of the deployment and management tasks have to be automated. Today, configuration management tooling is used to automate those tasks based on the paradigm of Infrastructure as Code. The next step is to utilize model-driven Cloud management to support collaboration and automation in DevOps scenarios. Thus, this section is focused on providing concepts of how a TOSCA container fitting DevOps methodologies can be realized.

3.5.1 TOSCA Container Architecture

One of the most straightforward approaches of implementing a TOSCA container is to build most of its components from scratch. However, this subsection is focused on outlining a TOSCA container architecture based on integrating tools that are already used in the DevOps world. This is the foundation for realizing a TOSCA container fitting DevOps methodologies. Figure 3.12 presents four of the major building blocks of a TOSCA container:

Management of Cloud Service Archives: Each TOSCA container implementation needs to store Cloud service archives that get loaded into the container in order to deploy and manage service instances based on those CSARs. In addition, the service template inside a CSAR needs to be parsed to create a service model that can be processed by other units in the TOSCA container such as the instance management unit and the inner management unit.

Instance Management of Infrastructure Resources: To run a service specified by a TOSCA service template, infrastructure resources have to be instantiated. Examples for such resources are virtual machines, a storage service, or a relation between two applications running on different machines. Furthermore, all instances have to be manageable by the TOSCA container. This includes monitoring and deprovisioning of running instances.

Inner Management of Infrastructure Resource Instances: In addition to managing infrastructure resource instances, a TOSCA container has to manage what is happening inside an instance, too. For instance, a TOSCA container has to install and configure software components on a virtual machine, which is an example for an infrastructure resource instance.

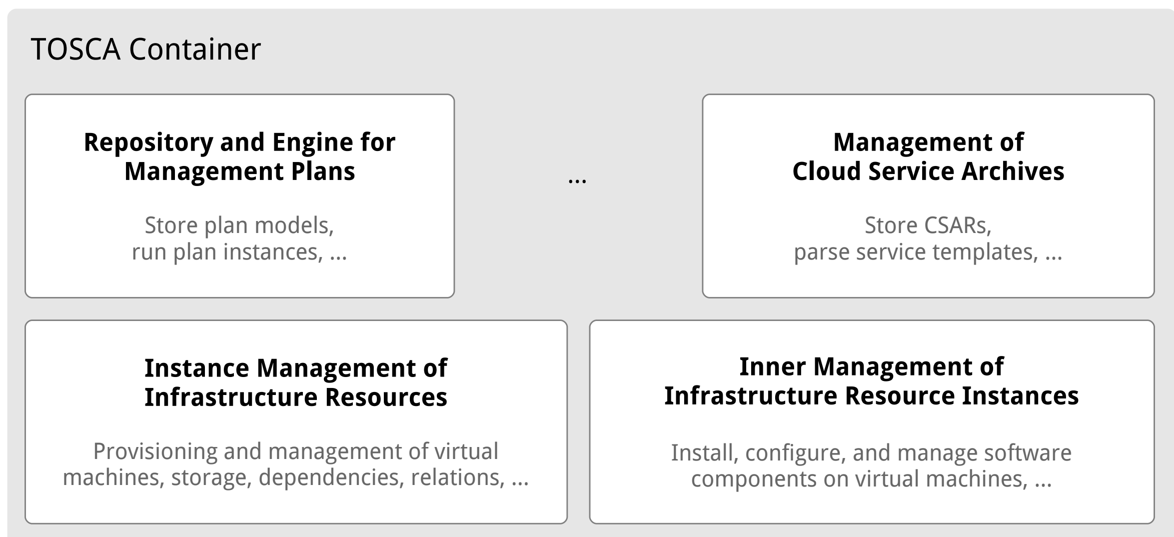

Figure 3.12: Four Major Building Blocks of a TOSCA Container

Repository and Engine for Management Plans: A TOSCA container needs to store management plan models in a repository. These models can be instantiated to run a particular management plan by the engine.

The three dots in Figure 3.12 indicate that additional building blocks may exist inside a TOSCA container. Of course, there are relations between those building blocks. For instance, the instance management unit uses the inner management unit to install and configure software components on a virtual machine that was provisioned before. Another example is the management plan engine: it can use the instance management unit and the inner management unit to provision a new virtual machine or to update an application on a virtual machine that is already running.

Figure 3.13 outlines the architecture of a TOSCA container fitting DevOps methodologies based on the four building blocks presented in Figure 3.12: (1) a model transformation component is needed as part of the CSAR management unit. This component transforms the TOSCA service template into a model that can be processed by the service orchestration tool or framework, which is used to realize instance management. The CSAR repository can be any kind of storage where the CSARs are stored. (2) The instance management of infrastructure resources is based on service orchestration tooling such as Juju [Can12c]. (3) To realize the inner management of infrastructure resource instances, a configuration management tool such as Chef [Ops12d] or Puppet [Pup12e] is used. As a result, the TOSCA container can process implementation artifacts that are included in the TOSCA service template using the natural integration approach described in Subsection 3.3.2. To enable processing of TOSCA standard artifacts such as script artifacts or standard deployment artifacts, the transformation component outlined before dynamically creates corresponding configuration management artifacts. (4) Management plans are usually realized using business process languages such as

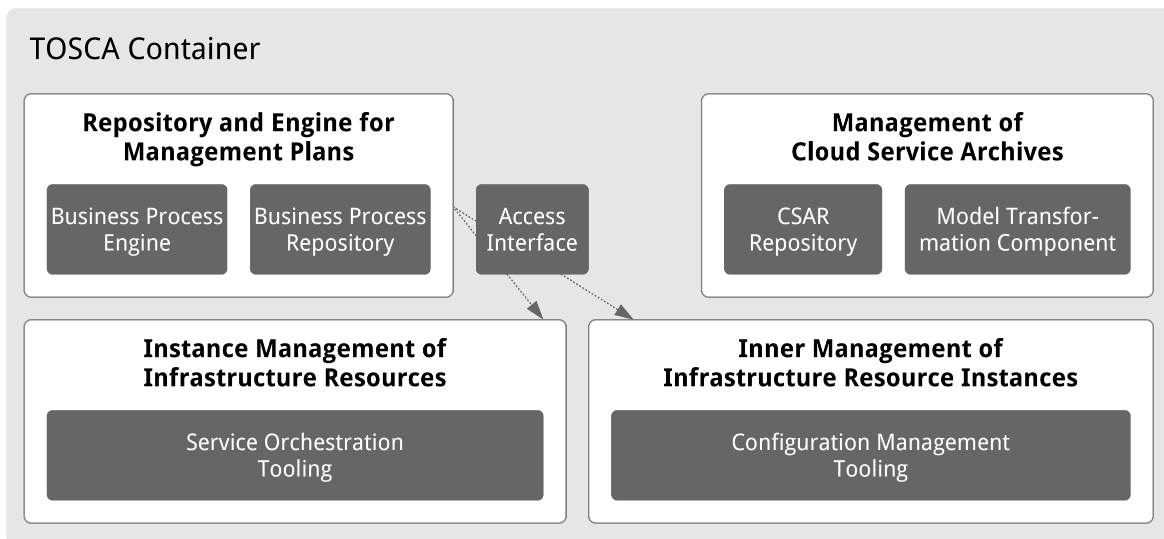

Figure 3.13: High-Level Overview of the TOSCA Container Architecture

BPEL [Org12c] or BPMN [Obj12]. Thus, a business process engine and a business process repository can be used to store, run, and manage TOSCA plans. In addition, an access interface is needed that enables the management plans to interact with the instance management and the inner management.

By realizing a TOSCA container using existing service orchestration tooling and configuration management products, those tools can be used directly to perform management tasks. This is an alternative approach to defining management plans using a business process language. This alternative approach can also be used to modify the service topology in an ad-hoc manner and to utilize the configuration management infrastructure for maintaining nodes in the service topology. For instance, nodes can be added to the topology and additional software components can be installed on existing nodes.

3.5.2 Model Transformation for Reusing Existing Artifacts

Because DevOps is much about automation, the DevOps communities [Can12c, Ops12a, Pup12d] of configuration management and service orchestration products are actively publishing and sharing reusable artifacts to manage applications and services. Subsection 3.3.2 outlined how configuration management artifacts can be embedded into a TOSCA service template. In that case, configuration management artifacts are attached to existing node type definitions. Another approach for reusing those existing artifacts is to generate node type definitions based on such an artifact. This goal can be achieved by performing a model transformation. As an example, Subsection 2.4.3 outlined the similarities between Juju charms and TOSCA node types. Thus, a model transformation tool can be implemented to generate a node type definition based on a charm. Another potential use case for implementing a model transformation is to generate a node type skeleton based on a Chef cookbook. Then, such a skeleton needs to be refined, for instance, by attaching additional implementation artifacts to realize all the lifecycle operations. This approach is similar to executing step 1.1, step 1.2, and step 1.3 that are part of the semi-automatic model creation procedure described in Subsection 3.4.2.

This approach of performing model transformations to reuse existing artifacts fits the TOSCA container architecture outlined in Subsection 3.5.1. The existing artifacts published by the DevOps communities can be used and integrated to realize a service based on model-driven Cloud management. The TOSCA container can easily process those artifacts. This is because the container is based on the configuration management and service orchestration tools for which those artifacts are created.

3.5.3 Evaluation

The goal of Subsection 3.5.1 was to outline the architecture of a TOSCA container fitting DevOps methodologies. This goal was achieved by realizing some of the container's major building blocks using tools and frameworks that are developed and used by the DevOps communities today. In addition, Subsection 3.5.2 described how to realize model transformations to reuse existing artifacts published by the DevOps communities. Those approaches covered the use

cases of deploying service instances based on a CSAR as well as managing the deployed service instances. The benefits of those approaches are:

1. A TOSCA container does not have to be implemented from scratch. Major building blocks can be implemented based on existing configuration management and service orchestration tools.
2. The entry barrier for people, who are already familiar with configuration management and service orchestration tooling is lowered toward model-driven Cloud management. They can still use their knowledge about those tools to manage services and applications. In addition, a holistic model is available that represents the service.
3. Services can be built following the paradigm of model-driven Cloud management. This can be done easily by reusing and combining existing artifacts published by the DevOps communities.
4. Management tasks are not limited to management plans only. The underlying tools on which the TOSCA container is based can be used to perform especially simple management tasks such as updating a software package on a set of virtual machines.

On the other hand, there are drawbacks for realizing those approaches:

1. Performing management tasks without defining management plans can increase flexibility. However, when the TOSCA service template does not contain management plans, the business process engine inside the TOSCA container cannot be used to perform management tasks.
2. The TOSCA container implementation depends on interfaces and meta models of external tools. It may be challenging to keep everything in sync, especially when new tool versions change meta models.
3. It is hard to manage services such as storage or higher-level platform services using configuration management and service orchestration tools. This is because they do not directly support the management of such resources. As a result, additional components need to be added to the TOSCA container to handle such resources beyond the scope of virtual machines.

This evaluation shows that the concepts outlined in this section can be used to create a simple TOSCA container implementation. When it comes to more advanced features, the effort for realizing those may be similar to implementing them for a TOSCA container that was built from scratch.

4 Design and Implementation

This chapter describes the implementation parts that realize prototypes for some of the concepts outlined in Chapter 3. In order to refer to the actual implementations the following notation is used in this chapter: *(Prototype Impl.: `<Directory Name>`).* The `<Directory Name>` points to the directory where the actual implementation is located on the compact disc that is attached to this thesis. The foundation for all the prototype implementations is a Cloud service archive that realizes the Sugar service outlined in Section 3.1. IBM developers created a JUnit [JUn12] test case for the development of the TOSCA importer component inside the IBM Common Cloud Stack. Subsection 2.4.2 outlined the role of this component. The JUnit test case generates a CSAR that realizes the Sugar service topology presented in Figure 3.3 in Section 3.1.

The core of the generated Sugar CSAR is the service template. It defines several node types such as “MySQL Server” and “Sugar Database.” In addition, requirement and capability types are specified; two examples are: “MySQL Database Container Requirement” and “MySQL Database Container Capability.” Instances of those are attached to node types as requirement and capability definitions. As an example, the “MySQL Server” node type has a “Databases” capability definition attached. The type of this definition is “MySQL Database Container Capability.” As a result, the “MySQL Server” node type can fulfill the MySQL database container requirement definition of another node type. Inside the Sugar service topology the “Sugar Database” node type owns such a requirement definition. Thus, the node templates “MySQL Server” and “Sugar Database” based on their corresponding node types can be linked by creating a “hosted on” relationship template between the requirement and the capability inside the topology template. Such a relationship template is included for each requirement of each node template to fulfill all requirements in the topology template. As an example for how the node type definitions look like, Figure 4.1 outlines the structure of the “MySQL Server” node type.

For the Sugar service template as a whole, it would not have been necessary to specify requirements and capabilities at all. The node types could have been defined without any requirement or capability definition. Then, a relationship template inside the topology template would connect two node templates directly. The drawback of omitting requirements and capabilities is that the node types are not completely self-contained; that is, they cannot be easily reused to realize another service template. This is because the node type definitions do not expose which requirements they have and which capabilities they offer.

Because management plans are out of scope for the prototype implementations of this thesis, it does not matter that there are no plans included in the generated Sugar CSAR. The prototypes are focused on the Sugar service deployment. Management tasks that are performed after

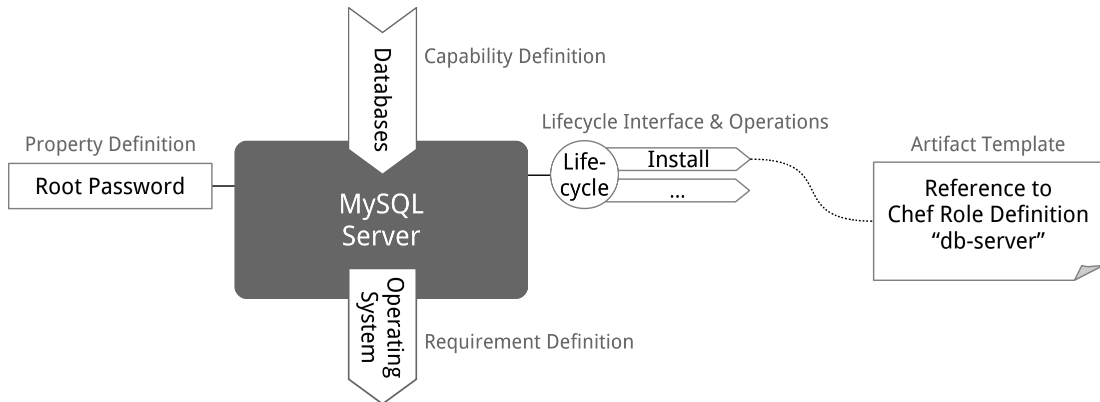

Figure 4.1: Sample Node Type Definition: MySQL Server in the Sugar Service Template

the service has been deployed are not explicitly supported. The deployment itself is realized by following a declarative approach. Thus, the deployment is not performed by executing a "build plan." It is performed by viewing the topology template as a declarative topology graph. This graph gets traversed by following the relationships, beginning at the bottom level of virtual machines. For all nodes that own a "lifecycle" interface, the lifecycle operations (1) "create," (2) "configure," and (3) "start" are called in this order. Some relationships own a specific operation that corresponds to their type. For instance, a "hosted on" relationship may own a "hostOn" operation. This operation is executed when the particular relationship gets established. Those specific lifecycle operations are originating in best practices established at IBM. They are currently implemented by the IBM Common Cloud Stack. TOSCA will define standardized lifecycle operations in the future [Org12a].

The following Section 4.1 is focused on how to include Chef artifacts into the Sugar CSAR. Those artifacts implement the lifecycle operations mentioned before. The corresponding concepts for integrating configuration management artifacts into a CSAR were outlined in Subsection 3.3.2. Then, Section 4.2 presents a prototype that extends the IBM Common Cloud Stack to enable it to process Chef artifacts.

4.1 Integration of Chef Artifacts into the Sugar Cloud Service Archive

As stated before, some of the node types and relationship types in the Sugar service template own lifecycle operations that are called when the service gets deployed. This chapter outlines how to attach Chef artifacts to implement those operations. The corresponding concepts were outlined in Subsection 3.3.2. Figure 4.1 shows a Chef-specific artifact template that is attached to the "install" operation of the "MySQL Server" node type.

For the node types in the Sugar CSAR, the "install" operation is the only lifecycle operation that gets a real implementation attached. The "configure" operation as well as the "start" operation get empty implementation artifacts attached only. This is because for Chef it is usual

that the installation, configuration, and the start of a software component is realized using a single artifact. Other lifecycle operations such as “stop” and “uninstall” are also implemented using an empty Chef artifact, because the prototype implementation is focused on the service deployment. For all relationship types that own a corresponding operation such as “hostOn” or “connectTo,” a Chef artifact is attached to implement those operations.

The following Subsection 4.1.1 presents how to include Chef artifacts into a CSAR using the natural integration approach described in Subsection 3.3.2. Then, Subsection 4.1.2 is about realizing the transparent integration approach for Chef artifacts. The combination of both integration approaches is realized in Subsection 4.1.3.

4.1.1 Natural Integration

As indicated in Figure 4.1, an artifact template needs to be defined to realize an implementation artifact for an operation. The complete XML schema definition for how to define an artifact template can be obtained from [Org12b]. Listing 4.1 represents the Chef artifact template that implements the “install” operation of the “MySQL Server” node type.

This example outlines the structure of an artifact template including the generic parts that are common for each artifact template as well as the Chef-specific parts. The *ArtifactTemplate* element itself belongs to the generic parts and owns two attributes: the *id* attribute specifies a unique identifier for this particular artifact template inside the service template. The *type* attribute specifies the artifact type. To reference a particular artifact type, it has to be defined inside the service template. Listing 4.2 presents the Chef artifact type definition that is included in the Sugar service template. The *ArtifactReferences* element including all its child elements is a generic mechanism for artifact templates to point to the files included in the CSAR that are necessary to execute the artifact. In case of Chef artifact templates, the cookbooks and role definitions get referenced in this section. The content of the *Properties* element is Chef-specific. The structure of the *ChefArtifactProperties* element including all its child elements is defined in the Chef artifacts XML schema definition (Prototype Impl.: ChefArtifacts.xsd). It consists of the following parts:

Cookbooks: Each cookbook that is needed to realize a particular artifact template is referenced using a *Cookbook* element. This includes cookbooks that are directly referenced inside the *RunList* section as well as dependencies of those.

Roles: Each role definition that is needed to realize a particular artifact template is referenced using a *Role* element. This includes roles that are directly referenced inside the *RunList* section as well as dependencies of those.

Mappings: Values of node type properties can be mapped to cookbook attributes using a *PropertyMapping* element. The *propertyPath* attribute contains an XPath expression [Wor12a] that points to a particular property. To refer to the corresponding cookbook attribute, the *cookbookAttribute* attribute is used. The same can be done for relationship type properties. Due to the fact that a relationship owns a source node and a target node, their property values can be mapped using a *SourcePropertyMapping* or a

Listing 4.1 Chef-Specific Artifact Template to Implement MySQL Server's "Install" Operation

```
<ArtifactTemplate id="at-735bfe66-4e32-4bee-a458-be41b6de49c3" type="chef:ChefArtifact">
  <Properties>
    <chef:ChefArtifactProperties xmlns:chef="http://www.example.com/ChefArtifacts"
        xmlns="http://www.example.com/ChefArtifacts">
      <Cookbooks>
        <Cookbook location="files/chef/cookbooks/build-essential.zip" name="build-essential"/>
        <Cookbook location="files/chef/cookbooks/openssl.zip" name="openssl"/>
        <Cookbook location="files/chef/cookbooks/mysql.zip" name="mysql"/>
      </Cookbooks>
      <Roles>
        <Role location="files/chef/roles/db-server.json" name="db-server"/>
      </Roles>
      <Mappings>
        <PropertyMapping mode="input" propertyPath="/RootPassword"
            cookbookAttribute="mysql/server_root_password"/>
      </Mappings>
      <RunList>
        <Include>
          <RunListEntry roleName="db-server"/>
        </Include>
      </RunList>
    </chef:ChefArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="files/chef/cookbooks">
      <Include pattern="build-essential.zip"/>
      <Include pattern="openssl.zip"/>
      <Include pattern="mysql.zip"/>
    </ArtifactReference>
    <ArtifactReference reference="files/chef/roles">
      <Include pattern="db-server.json"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```

TargetPropertyMapping element. If the particular artifact template implements a script operation, input and output parameters can be mapped using an *InputParameterMapping* or an *OutputParameterMapping* element.

RunList: At its core, a Chef artifact template defines which recipes and roles have to be part of the corresponding run list. The *RunList* element can contain two child elements: the *Include* element consists of at least one *RunListEntry* element that point to recipes and roles. Those have to be part of the run list in the given order. The *RunListEntry* child elements of the *Exclude* element denote recipes and roles that have to be explicitly excluded from the run list. Two examples for *RunListEntry* definitions are presented in Listing 4.1 and Listing 4.3.

As an additional remark regarding the definition of property mappings, the *mode* attribute specifies the mapping direction: "input" means that the property value is mapped to the cookbook attribute before the Chef recipes are executed, "output" maps the cookbook attribute

Listing 4.2 Chef Artifact Type Definition

```
<ArtifactType name="ChefArtifact" targetNamespace="http://www.example.com/ChefArtifacts">
  <PropertiesDefinition element="chef:ChefArtifactProperties"/>
</ArtifactType>
```

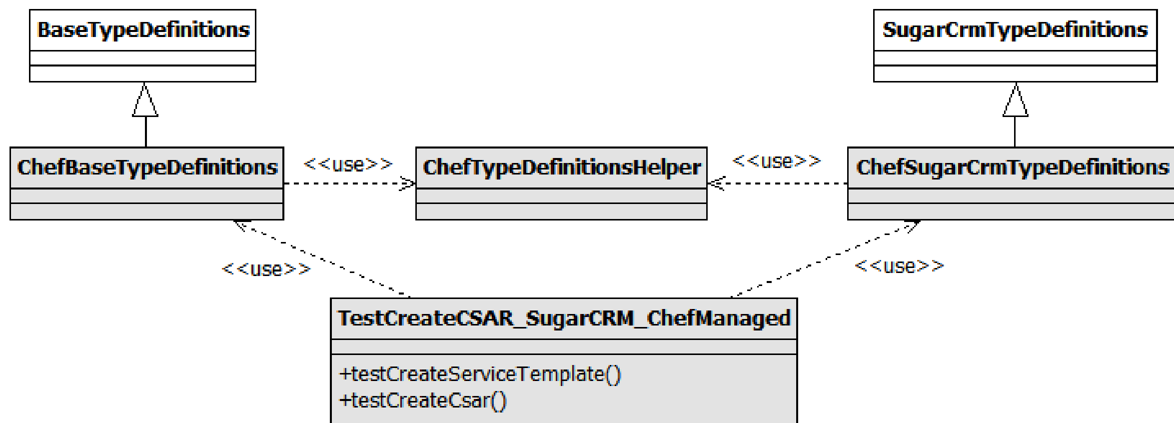

Figure 4.2: Class Diagram for the Modified JUnit Test Case to Generate the Sugar CSAR

to the property after the Chef recipes were executed, and “input-output” combines both kinds of mapping.

This is how Chef artifact templates can be realized using the natural integration approach. The TOSCA container has to understand the content of the *ChefArtifactProperties* element to perform the corresponding actions. However, it is completely up to the TOSCA container implementation how to perform those actions. Two options are: the Chef server’s REST API and Chef’s command line interface.

As part of this thesis, the JUnit test case to generate a Sugar CSAR was modified and extended (Prototype Impl.: Sugar CSAR Generator). The output of this test case is a Sugar CSAR including Chef artifact templates that implement the operations defined in the service template. Those Chef artifact templates point to cookbooks that get included in the CSAR. The Java class “TestCreateCSAR_SugarCRM_ChefManaged” located inside the package “com.ibm.orchestrator.tosca.vapp.test.chef” implements the modified version of the JUnit test case. Figure 4.2 presents the modular design of the test case: the classes shaded in gray were implemented as part of this thesis. The classes “ChefBaseTypeDefinitions” and “ChefSugar-

Listing 4.3 Alternative Run List Definition for Chef-Specific Artifact Template

```
<RunList>
  <Include>
    <RunListEntry cookbookName="mysql" recipeName="install"/>
  </Include>
</RunList>
```

CrmTypeDefinitions" were derived from their original counterparts "BaseTypeDefinitions" and "SugarCrmTypeDefinitions." Methods to create basic node types and basic relationship types such as "createNodeType_MySql" are provided by "BaseTypeDefinitions." Type definitions that are specific to Sugar are created using methods such as "createNodeType_SugarApp" provided by "SugarCrmTypeDefinitions." The Chef-specific classes "ChefBaseTypeDefinitions" and "ChefSugarCrmTypeDefinitions" override all methods of their original counterparts that generate implementation artifacts. This is the foundation for realizing the goal of the modified test case, namely to generate a CSAR including Chef artifacts instead of including script artifacts. "ChefTypeDefinitionsHelper" provides a set of helper methods to keep the implementation of "ChefBaseTypeDefinitions" and "ChefSugarCrmTypeDefinitions" simple and to avoid code duplication. The actual JUnit class "TestCreateCSAR_SugarCRM_ChefManaged" provides two methods: (1) "testCreateServiceTemplate" generates the service template whereas (2) "testCreateCsar" packs the generated service template including all artifacts that are referenced in it such as cookbooks together into a Cloud service archive. As a remark, the classes behind the scenes that represent TOSCA entities such as node types and relationship types were generated using the Java Architecture for XML Binding (JAXB). This makes it easy to serialize and deserialize TOSCA service templates.

4.1.2 Transparent Integration

The transparent integration approach described in Subsection 3.3.2 does not have any additional requirements for the TOSCA container. How this approach can be implemented based on Chef is described in this subsection. Listing 4.4 represents the artifact template that implements the "install" operation of the "MySQL Server" node type using the transparent integration approach. In contrast to the approach described in the previous subsection, the *type* attribute of the *ArtifactTemplate* element indicates that this is a script artifact. It is assumed that this is a standard artifact type in TOSCA, as currently discussed in the TOSCA technical committee [Org12a]. As a result, each TOSCA container can process artifacts of that type. A corresponding artifact type definition for script artifacts has to be included in the service template. This type definition would be very similar to the Chef artifact type definition outlined in Listing 4.2.

The artifact template presented in Listing 4.4 implements the very same operation as the artifact template in Listing 4.1. The content of the *ScriptArtifactProperties* element is limited to some generic meta data regarding the corresponding script. All the Chef-related information such as the mappings and the run list entries are hidden inside the wrapper script "install.sh." As a result, the Chef specifics of the artifact template are completely transparent to the TOSCA container: the container implementation does not have to know anything about Chef. However, all the actions necessary to realize the artifact are hard coded inside the wrapper script. The TOSCA container does not have much flexibility in performing the corresponding actions. Whereas for the natural integration approach the mapping of properties and parameters is explicitly defined inside the artifact template, those mappings are hidden inside the wrapper script for the transparent integration. The TOSCA technical committee did not establish a convention yet how scripts can access properties of nodes and relationships. The current

Listing 4.4 Script Artifact Template to Implement MySQL Server's "Install" Operation

```
<ArtifactTemplate id="at-0de3dfa4-9faa-4bb0-81ca-f5ad8247b432" type="af:ScriptArtifact">
  <Properties>
    <af:ScriptArtifactProperties
        xmlns:af="http://docs.oasis-open.org/tosca/ns/2011/12/Artifacts"
        xmlns="http://docs.oasis-open.org/tosca/ns/2011/12/Artifacts">
      <ScriptLanguage>sh</ScriptLanguage>
      <PrimaryScript>scripts/mysql/install.sh</PrimaryScript>
    </af:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="files/chef/cookbooks">
      <Include pattern="build-essential.zip"/>
      <Include pattern="openssl.zip"/>
      <Include pattern="mysql.zip"/>
    </ArtifactReference>
    <ArtifactReference reference="files/chef/roles">
      <Include pattern="db-server.json"/>
    </ArtifactReference>
    <ArtifactReference reference="scripts/mysql">
      <Include pattern="install.sh"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```

implementation of the Common Cloud Stack exposes all properties as environment variables to the script. As an example, the wrapper script can access the "$RootPassword" variable to get the value of the node type property "RootPassword."

4.1.3 Combined Integration Based on a Preprocessor Component

The natural and the transparent integration approach can be combined as described in Subsection 3.3.2. To stick to the "install" operation of the "MySQL Server" node type the artifact templates presented in both Listing 4.1 and Listing 4.4 can be included in a single service template. They can be both attached as implementation artifacts to the "install" operation.

However, a more mature approach would be to include the Chef-specific artifact templates in the original service template only. Those artifact templates do not contain any hard-wired assumptions in wrapper scripts. Thus, the service template owns a high degree of portability. Any TOSCA container that understands the Chef artifact type can directly process the included Chef artifact templates. For TOSCA container implementations that cannot process those artifact templates, a preprocessor component can be implemented. Such a component automatically generates a corresponding script artifact template (Listing 4.4) for each Chef artifact template (Listing 4.1) and embeds them into the Cloud service archive.

Figure 4.3 outlines a use case for implementing such a preprocessor component. The following Section 4.2 describes the prototype implementation that extends the IBM Common Cloud Stack

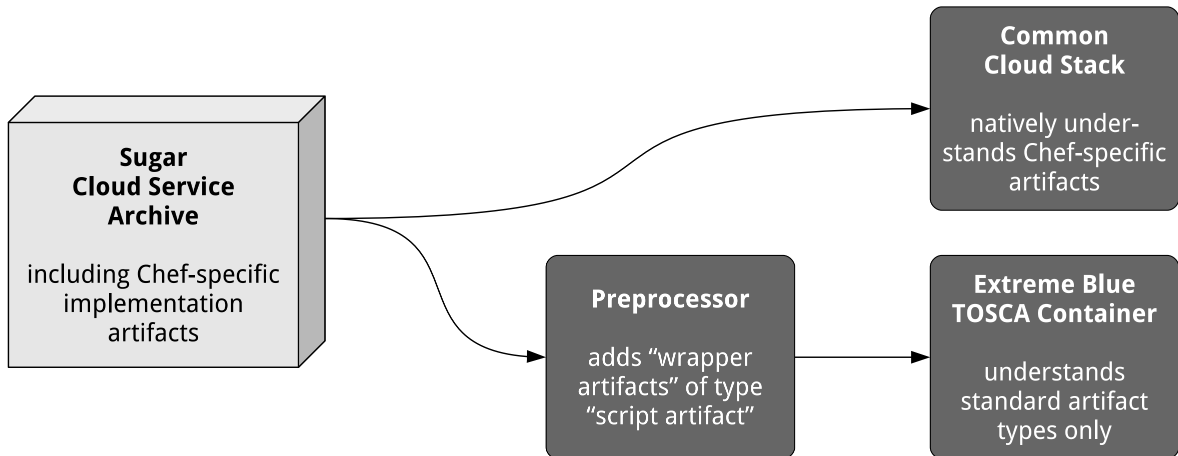

Figure 4.3: Use Case for a CSAR Preprocessor Component for Chef-Specific Artifacts

so that Chef artifact templates can be processed. Thus, the Common Cloud Stack represents a TOSCA container implementation that natively understands Chef artifact templates. As a result, the generated Sugar CSAR including Chef artifact templates can be directly processed by the Common Cloud Stack. In addition, four students at IBM developed a TOSCA container during the “Extreme Blue 2012” summer internship. This container understands standard artifact types only. To enable this container implementation to process the very same generated Sugar CSAR, a preprocessor component was implemented as part of this thesis (Prototype Impl.: CSAR Preprocessor). This preprocessor realizes the combined integration approach for Chef artifact templates: a corresponding script artifact template gets embedded into the given Cloud service archive for each Chef artifact template. The preprocessed CSAR can be processed by the students’ TOSCA container without changing the container implementation.

4.1.4 Evaluation

This section presented prototype implementations to integrate Chef artifacts into a Cloud service archive. Those implementations enabled the attachment of Chef artifacts to the Sugar CSAR. As a result, there are several lessons learned so far:

1. Most of the cookbooks included in the generated CSAR were directly obtained from [Ops12a] without any modification. Only the “sugarcrm” cookbook was created manually. This proves the point that many configuration management artifacts published and shared by the DevOps communities [Ops12a, Pup12d] can be reused to be included in Cloud service archives.
2. For each operation a Chef artifact template and a script artifact template can be included in the CSAR to implement the very same operation.

3. However, the script artifact templates should not be created manually. In addition, they should not be part of the original CSAR because they contain hard-wired assumptions such as platform-specific commands. A CSAR preprocessor component should generate those artifact templates and embed them into the CSAR. The script artifact templates act as wrapper artifacts for the actual Chef artifact templates.
4. Each TOSCA container that understands the Chef artifact type can process the included Chef artifact templates. It is completely up to the container implementation how the artifact template is realized. This leads to a high degree of portability because there are no platform-specific assumptions made inside the Chef artifact templates.
5. Furthermore, any TOSCA container that does not understand the Chef artifact type can use the generated wrapper artifact. Thus, the Cloud service archive as a whole is highly portable.

There are two open issues related to the preprocessor component for Cloud service archives:

1. The preprocessor prototype implementation that was realized as part of this thesis is not configurable at all. As an example, the preprocessor assumes that the generated scripts are executed on a Linux machine. However, some of the included commands and assumptions may be different for a machine running a different operating system. This is why it makes sense to introduce configuration parameters for the preprocessor. Those parameters can influence the content of the generated wrapper scripts.
2. The core of the preprocessor component could be implemented as a separate and independent library. Then, this library can be used to implement individual preprocessor components for existing TOSCA container implementations. In addition the library can be the foundation for a component inside a TOSCA container that processes Chef artifact templates.

4.2 Extension of IBM Common Cloud Stack to Process Chef Artifacts

This section describes the prototype implementation that extends the IBM Common Cloud Stack (Prototype Impl.: Common Cloud Stack Extension) to enable it to process Cloud service archives including Chef artifacts. As a result, the extension enables the Common Cloud Stack to deploy the Sugar CSAR generated by the prototype implementation described in Subsection 4.1.1 (Prototype Impl.: Sugar CSAR Generator). The following Subsection 4.2.1 is focused on achieving this goal by integrating an existing Chef infrastructure including a Chef server. Subsection 4.2.2 describes an approach that is based on "Chef solo." This is a very lightweight approach of using Chef where the Chef client directly executes recipes without connecting to a Chef server.

4.2.1 Extension Based on an Existing Chef Server

The current implementation of processing Cloud service archives using the Common Cloud Stack was outlined in Figure 2.16 in Subsection 2.4.2. This implementation was extended as part of this thesis (Prototype Impl.: Common Cloud Stack Extension). Figure 4.4 presents an overview of how a CSAR is deployed using the extended Common Cloud Stack and a Chef server. The TOSCA importer component is extended to understand Chef artifact templates as outlined in Subsection 4.1.1. All the cookbooks that are referenced in at least one artifact template get uploaded to the Chef server by the TOSCA importer. The creation of plug-ins that correspond to node types and relationship types as well as the creation of virtual application patterns does not change. The only difference to the original implementation is that the generated plug-ins do not directly contain the artifacts that implement their operations. Instead, there are references inside each plug-in pointing to cookbooks and recipes that were uploaded to the Chef server before. The actual deployment is performed using the service deployer component after the CSAR was imported.

Figure 4.5 presents the part of the TOSCA importer, which is responsible for generating the plug-ins that correspond to TOSCA node types and relationship types. The classes shaded in gray were implemented as part of this thesis. Starting from the "PluginGenerator," the "PartGenerator" is called that creates the individual parts of the generated plug-in. The input for the "PluginGenerator" is a node type or a relationship type. Originally, the "PartGenerator" calls the "ScriptGenerator" in order to process implementation artifacts that are script artifacts. The "ChefScriptGenerator" was implemented as part of this thesis to enable the TOSCA importer to process Chef artifacts. "ChefScriptGenerator" calls "ChefScript" to generate a script that will realize the property mappings and populate the run list of the corresponding

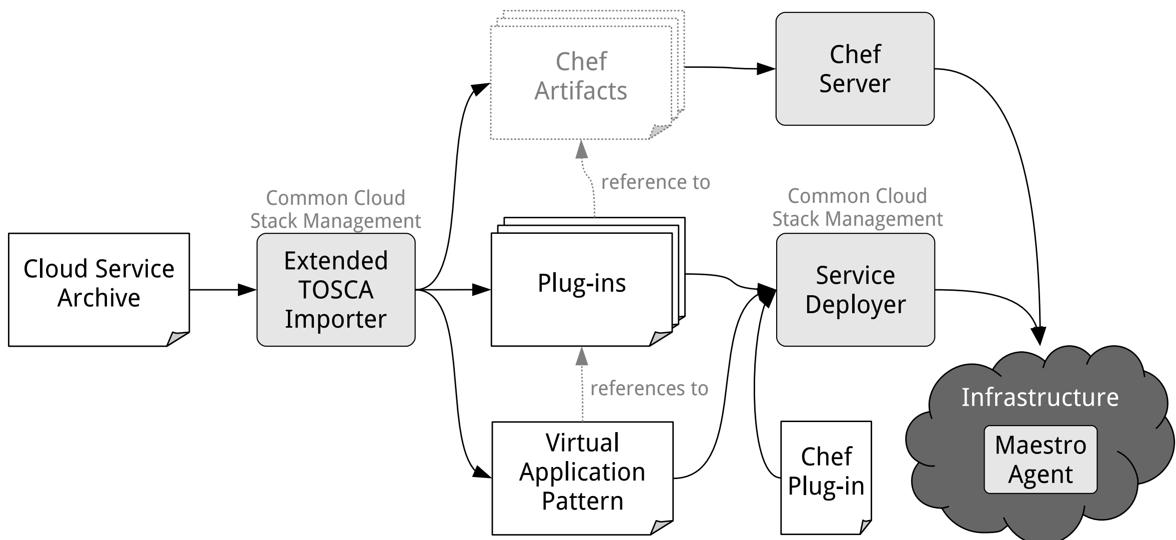

Figure 4.4: CSAR Deployment Using IBM Common Cloud Stack and Chef Server

node during service deployment. In addition, cookbooks and role definitions that are referenced inside those generated scripts get uploaded to the Chef server using "ChefUploader."

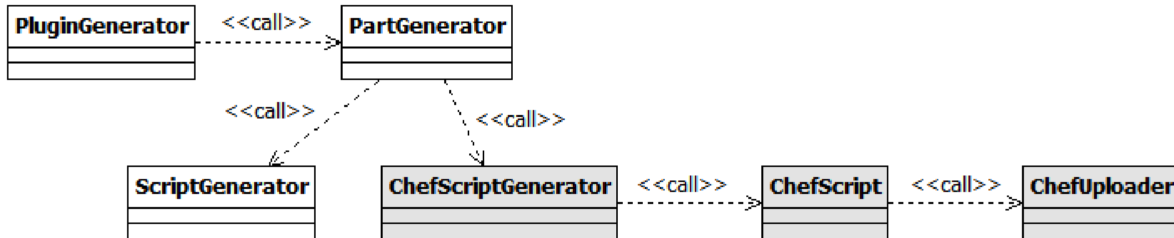

Figure 4.5: Class Diagram for the Extended TOSCA Importer Part that Generates Plug-Ins

Beside the generated plug-ins, the Chef plug-in is implemented separately. It is in charge of installing and configuring the Chef client on the virtual machines that get provisioned inside the target infrastructure where the service is deployed. The Chef plug-in offers several configuration parameters to set up the credentials to enable the Chef clients to communicate with the Chef server. After the Chef client was installed and configured on a particular machine it connects to the Chef server to retrieve the recipes that have to be executed on that machine. All the machines that are registered at the Chef server can be managed using Chef tooling. As an example, the Chef server's REST API as well as its Web-based user interface can be used to perform management tasks.

Figure 4.6 shows a screenshot of the Common Cloud Stack's management user interface presenting the topology of the imported Sugar CSAR. The deployed Sugar service instance is presented in Figure 4.7, whereas Figure 4.8 shows the Chef Server's management user interface presenting the run list of the node where the Sugar application is running.

Figure 4.6: Screenshot of the Topology of the Imported Sugar Cloud Service Archive

Firefox

IBM SmartCloud Orchestrator | Chef Server

https://xvm296/dashboard/runtime/deployments/#d-11e1696e-c921-4bbb-a560-77f25f624415

IBM SmartCloud Orchestrator — Administrator | Help | About | Logout — IBM

Welcome | Instances | Catalog | Reports | Cloud | System | Patterns

Virtual Application Instances

Stop | Delete | Manage | Upgrade | Maintain | Resume

SugarCRM_ChefManaged Service Template

Name:	SugarCRM_ChefManaged Service Template
Created by:	cbadmin
Started on:	Oct 23, 2012 9:14:59 PM
Access granted to:	Administrator [owner]
Virtual application instance ID:	d-11e1696e-c921-4bbb-a560-77f25f624415
Status:	Running
In cloud group:	bld08-06 Group
Pattern type:	TOSCA Foundation Pattern Type 1.0

Middleware perspective (5 in total)

- **PhpModule** (VmApache)
- **SugarCrmApp** (VmApache)
- **ApacheWebServer** (VmApache)
- **MySql** (VmMySql)
- **SugarCrmDb** (VmMySql)

Virtual machine perspective (2 in total)

Name	Public IP	VM Status	Started on	Role Status
VmApache. 11351019699691	9.152.138.119	Running Log	Oct 23, 2012 9:15:28 PM	PhpModule SugarCrmApp ApacheWebServer
VmMySql. 11351019699692	9.152.138.118	Running Log	Oct 23, 2012 9:15:29 PM	MySql SugarCrmDb

© Copyright IBM Corporation 2011. All Rights Reserved. 4.0.0.0-20121012224931 / 20121012-2248-068

Figure 4.7: Screenshot of the Deployed Sugar Service Instance

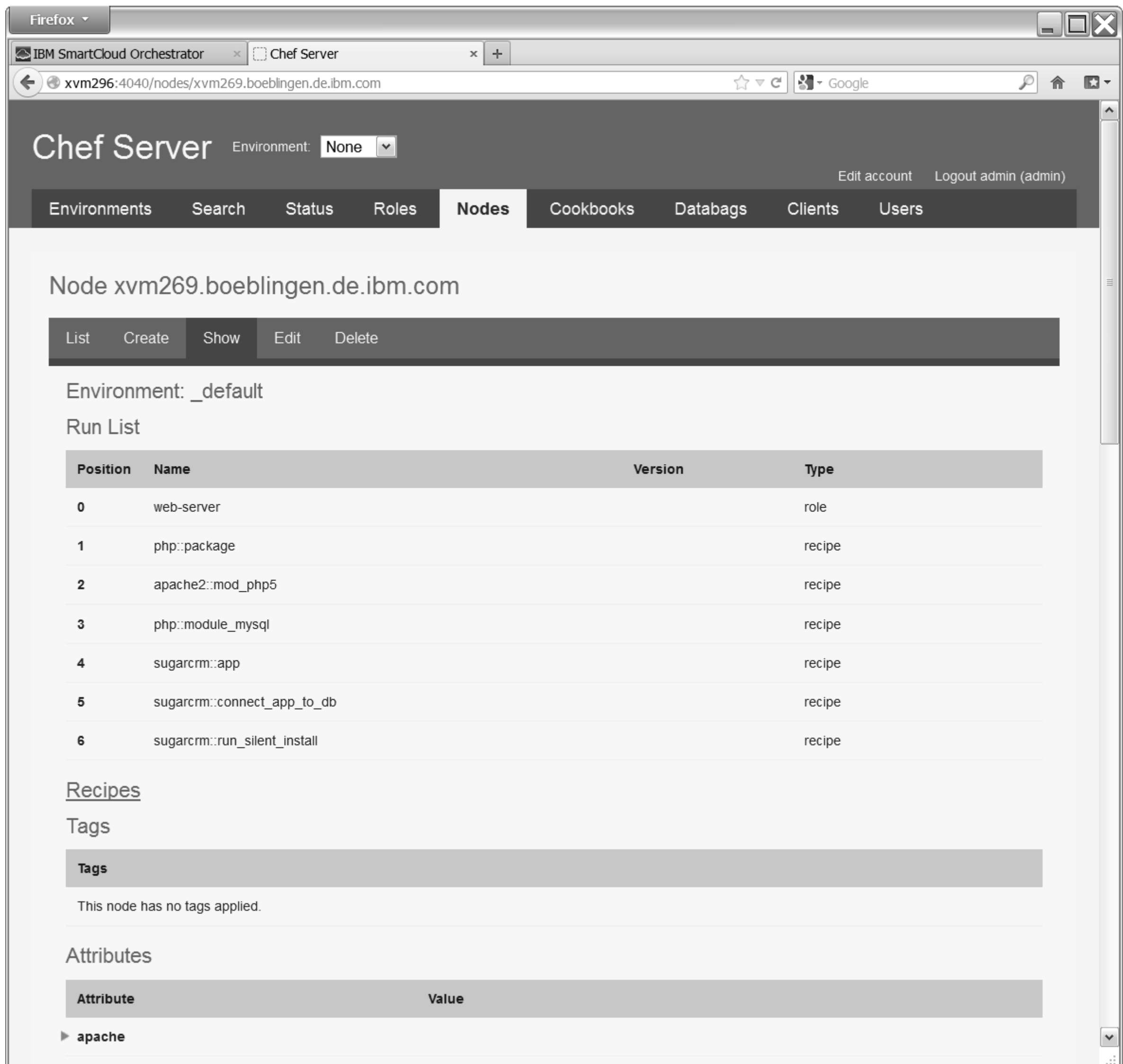

Figure 4.8: Screenshot of the Chef Server's Management User Interface

4.2.2 Extension Based on Chef Solo

Chef cannot only be used in a client/server scenario. The “Chef solo” approach enables the Chef client to execute recipes without any connection to a Chef server. The Common Cloud Stack extension that was implemented as part of this thesis (Prototype Impl.: Common Cloud Stack Extension) can also process Chef artifacts using Chef solo without a Chef server. Figure 4.9 outlines how this happens.

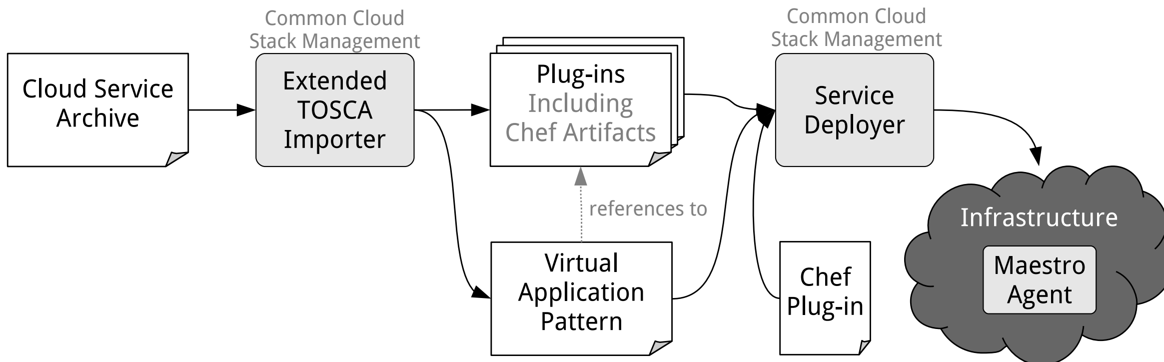

Figure 4.9: CSAR Deployment Using IBM Common Cloud Stack and Chef Solo

In contrast to the approach described in the previous subsection, the Chef cookbooks do not get uploaded to a Chef server because there is no Chef server in a Chef solo scenario. Instead, the Chef cookbooks are bundled with the generated plug-ins. Thus, each generated plug-in contains all recipes that are needed to perform the corresponding operations. The credentials configured using the Chef plug-in do not matter for Chef solo.

4.2.3 Evaluation

The extension of the Common Cloud Stack described in this section showed the elements and steps necessary to process Chef artifacts inside a TOSCA container. Lessons learned from this prototype implementation are:

1. Both the Chef solo approach and the Chef client/server approach make sense to process Chef artifacts that are included in a Cloud service archive.
2. The client/server approach implies more overhead because a separate Chef server is needed. However, the Chef server can be used to perform simple management tasks. The Chef server’s REST API as well as its Web-based user interface can be used to modify the run list of the machines where the Chef client is running.
3. There is a command line tool called “knife” [Ops12c] that acts as a REST client for the Chef server. “Knife” is part of the Chef client and thus can be used to perform management actions in conjunction with the Chef server. As a result, the Chef server’s REST API can be accessed using “knife” without implementing a REST client separately.

4. If there is no Chef server already available, Chef solo is a lightweight approach to process Chef artifacts inside a TOSCA container. However, in this case the Chef infrastructure cannot be used to perform management tasks.

The prototype implementation also revealed the following open issues:

1. Chef does not allow to define namespaces when it comes to cookbook names, role names, and attributes. As a result, naming conflicts can occur easily. As an example, there may be two different Cloud service archives that both include a cookbook called “mysql.” The content of those two cookbooks differs. When the first CSAR is imported using the Chef client/server approach, the “mysql” cookbook gets uploaded to the Chef server. The import of the second CSAR causes an upload of the “mysql” cookbook again. Thus, the original “mysql” cookbook gets replaced and some implementation artifacts for operations of the first CSAR may be missing or invalid. A possible solution for this issue could be to prefix the names to simulate different namespaces for different CSARs. Another approach could be to use an isolated Chef server instance for each deployed service. This approach would increase the management infrastructure’s security because several service deployments do not share a single Chef server. The current Chef server implementation is not multi-tenant aware.
2. The current prototype implementation is focused on deploying a Cloud service archive. However, the deployed service instance can be modified, for instance, by modifying run lists of provisioned machines. An open issue in this context is how to consider those modifications in case of exporting a Cloud service archive based on a deployed service instance. Somehow the implementation artifacts have to be updated to represent those modifications.
3. When an imported Cloud service archive is removed, the Chef server has to be cleaned up as well. The corresponding clean-up actions include the removal of cookbooks and roles that have been uploaded to the Chef server when the CSAR was imported.

As an overall result, the prototype implementation showed that it is relatively easy to implement a component to process Chef artifacts included in a Cloud service archive. The benefits and drawbacks of the two different approaches to use Chef, namely Chef client/server and Chef solo, are confirmed by the implementation. The open issues outlined before can be addressed in a refinement of the current prototype implementation.

5 Conclusion and Future Work

This thesis extended the scope of model-driven Cloud management by integrating DevOps methodologies into it. The goal was to bring together the strengths of both worlds and thereby making concepts, practices, and artifacts originating in the DevOps world reusable for scenarios based on model-driven Cloud management.

The starting point of achieving this goal was Chapter 2 that outlined the motivation for establishing DevOps methodologies originating in IT infrastructure and service management. Central aspects of DevOps are implemented using configuration management products such as Chef or Puppet. Those products follow the paradigm of Infrastructure as Code. This paradigm plays an important role when it comes to realizing DevOps methodologies. Based on the deficits of pure DevOps methodologies, the motivation for model-driven Cloud management was given. Chapter 3 motivated the integration of configuration management with model-driven Cloud management. The following integration concepts and prototype implementations described in Chapter 3 and Chapter 4 are the key contributions of this thesis:

1. Approach to realize service deployment for multiple environments such as development, test, and production using TOSCA (Subsection 3.3.1)
2. Natural, transparent, and combined integration of configuration management artifacts into TOSCA-based Cloud service archives (Subsection 3.3.2, Section 4.1)
3. Reusing configuration management artifacts in model-driven Cloud management because many of those are published and shared by the DevOps communities (Subsection 3.3.2)
4. Comparison of managing Cloud services using management plans and using configuration management tooling as well as combining both approaches (Subsection 3.3.3)
5. Approaches to create TOSCA-based service models for existing services that are managed using configuration management tooling (Subsection 3.4.1, Subsection 3.4.2)
6. Extension of an existing TOSCA container implementation to enable it to process configuration management artifacts based on Chef (Subsection 4.2)
7. Architectural approaches to realize a TOSCA container fitting DevOps methodologies (Subsection 3.5.1)
8. Proposal for realizing model transformations to generate reusable TOSCA artifacts such as node types based on existing configuration management artifacts (Subsection 3.5.2)

Those contributions clearly present the additional value that is provided by combining DevOps methodologies with model-driven Cloud management based on TOSCA. The prototype implementations described in Chapter 4 prove that the integration of both worlds is not only valuable from a conceptual point of view. In fact, the underlying concepts and approaches can be implemented in practice.

Currently, the TOSCA technical committee [Org12a] is discussing the structure of the TOSCA primer. The key concepts for integrating configuration management artifacts into a Cloud service archive as outlined in Subsection 3.3.2 can be proposed to be included in the primer. In addition, Chef-specific examples can be attached based on the work described in Section 4.1.

Furthermore, the TOSCA technical committee plans to define a set of artifact types that have to be understood by each TOSCA container implementation. The Chef artifact type that was introduced in Subsection 4.1.1 can be proposed to be part of this standardized set of artifact types. In addition, a Puppet artifact type can be defined and proposed because the Puppet community [Pup12d] is actively publishing reusable artifacts, too. This would lower the barrier to using TOSCA for people originating from the DevOps world.

When it comes to the implementation part, there is a lot of potential for future work. The concepts outlined in Section 3.4 to create a TOSCA-based model for an existing service are not covered by the implementation part of this thesis. The same is true for the concepts described in Section 3.5 that are focused on realizing a TOSCA container fitting DevOps methodologies. Prototype implementations can be created to realize those concepts. Then, the concepts itself can be refined based on the lessons learned of those implementations. In addition, the existing prototype implementations can be extended to support Puppet artifacts, too.

A Pseudo Code to Generate a Topology Template

```
/*
 * An instance of the type "ChefNode" represents a node that is registered at the
 * Chef server. An instance of the type "HostedOnStack" is a stack of node templates
 * inside the TOSCA topology template. Each node template in the stack is logically
 * connected to its underlying neighbor by a "hosted on" relationship template.
 * Example:
 *
 * TOPOLOGY TEMPLATE VIEW                      |                   "HOSTED ON" STACK
 * ===================================================================================
 * "operating system" node template            |
 *         |                                   |
 *         | "hosted on" relationship template |
 *         |                                   |   "operating system" node template
 *         V                                   |
 * "virtual machine" node template             |   "virtual machine" node template
 */



// List of all "hosted on" stacks
List<HostedOnStack> hostedOnStacks = new List();

// List of all nodes registered at the Chef server
List<ChefNode> chefNodes = getAllChefNodes();



// Iterate over all nodes
for(ChefNode node : chefNodes) {

  // Generate a new stack for the given node consisting of a "virtual machine" node
  // template and an "operating system" node template on top of it
  HostedOnStack stack = generateDefaultStack(node);

  // Iterate over the entries of the run list
  for (RunListEntry entry : node.getRunList()) {
    NodeType type = getNodeTypeForCookbook(entry.getCookbook());

    // If the stack does not contain a node template of the given node type, create
    // a node template and add it to the stack
    if (!stack.containsTemplateOfNodeType(type)) {
      NodeTemplate template = new NodeTemplate(type);

      stack.add(template);
    }
  }
```

```
  // Add the newly created stack to the list of "hosted on" stacks
  hostedOnStacks.add(stack);

}



// Create a TOSCA topology template skeleton based on the list of "hosted on" stacks,
// that is, the node templates get linked to one another by including "hosted on"
// relationship templates
TopologyTemplate template = createTopologyTemplate(hostedOnStacks);
```

Bibliography

[Ama12] Amazon Web Services LLC. Amazon Web Services, 2012. URL http://aws.amazon.com. (Cited on pages 11 and 33)

[APM12] APM Group Ltd. ITIL Official Site, 2012. URL http://www.itil-officialsite.com. (Cited on page 18)

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. Portable Cloud Services Using TOSCA. *Internet Computing, IEEE*, 16(3):80–85, 2012. (Cited on pages 8, 28, 30, 31 and 44)

[BCGG10] L. Baresi, M. Caporuscio, C. Ghezzi, S. Guinea. Model-Driven Management of Services. In *2010 IEEE 8th European Conference on Web Services*, pp. 147–154. IEEE, 2010. (Cited on page 27)

[BLS11] T. Binz, F. Leymann, D. Schumm. CMotion: A Framework for Migration of Applications into and between Clouds. In *2011 IEEE International Conference on Service-Oriented Computing and Applications*, pp. 1–4. IEEE, 2011. (Cited on page 28)

[Bur05] M. Burgess. A Tiny Overview of CFEngine: Convergent Maintenance Agent. In *Proceedings of the 1st International Workshop on Multi-Agent and Robotic Systems, MARS/ICINCO*. 2005. (Cited on page 20)

[Can12a] Canonical Ltd. Juju Charm Browser, 2012. URL http://jujucharms.com. (Cited on pages 8, 33 and 34)

[Can12b] Canonical Ltd. Juju Documentation, 2012. URL https://juju.ubuntu.com/docs. (Cited on pages 33, 34 and 36)

[Can12c] Canonical Ltd. Juju Official Site, 2012. URL https://juju.ubuntu.com. (Cited on pages 26, 33, 55 and 56)

[Can12d] Canonical Ltd. Ubuntu Official Site, 2012. URL http://www.ubuntu.com. (Cited on page 33)

[Cat10] D. Catteddu. Cloud Computing: Benefits, Risks and Recommendations for Information Security. *Web Application Security*, pp. 17–17, 2010. (Cited on page 15)

[CFE12a] CFEngine, Inc. CFEngine Customers, 2012. URL http://cfengine.com/use_cases. (Cited on page 19)

[CFE12b] CFEngine, Inc. CFEngine Manuals, 2012. URL http://cfengine.com/manuals. (Cited on page 21)

[CFE12c] CFEngine, Inc. CFEngine Official Site, 2012. URL http://cfengine.com. (Cited on pages 19 and 21)

[Cit12] Citrix Systems, Inc. CloudStack Official Site, 2012. URL http://www.cloudstack.org. (Cited on page 27)

[Clo12] Cloudways Ltd. Cloudways Official Site, 2012. URL http://www.cloudways.com. (Cited on page 27)

[Dix12] G. Dixon. IBM SmartCloud Evolution: SmartCloud Provisioning and SmartCloud Orchestrator (IBM Confidential), 2012. (Cited on pages 26 and 31)

[DJV10] T. Delaet, W. Joosen, B. Vanbrabant. A Survey of System Configuration Tools. In *Proceedings of the 24th Large Installations Systems Administration (LISA) conference (San Jose, CA, USA, 11/2010 2010), Usenix Association*. 2010. (Cited on pages 19 and 20)

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005. (Cited on page 18)

[GDQC09] S. Galup, R. Dattero, J. Quan, S. Conger. An Overview of IT Service Management. *Communications of the ACM*, 52(5):124–127, 2009. (Cited on page 18)

[GHS10] S. Günther, M. Haupt, M. Splieth. Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Technical report, Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010. (Cited on pages 8, 18, 19, 20, 21, 23, 24, 40, 41 and 46)

[Goo12] Google Inc. Google App Engine, 2012. URL https://developers.google.com/appengine. (Cited on page 11)

[Hew12] Hewlett-Packard Development Company, L.P. Hewlett-Packard Official Site, 2012. URL http://www.hp.com. (Cited on page 27)

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. (Cited on pages 16, 17 and 53)

[HKJ+09] H. Han, S. Kim, H. Jung, H. Yeom, C. Yoon, J. Park, Y. Lee. A RESTful Approach to the Management of Cloud Infrastructure. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pp. 139–142. 2009. (Cited on page 27)

[HM11] J. Humble, J. Molesky. Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IT Journal*, 24(8):6, 2011. (Cited on pages 16 and 17)

[IBM11] IBM. Getting Cloud Computing Right. 2011. (Cited on pages 15 and 16)

[IBM12a] IBM Corporation. IBM Business Process Manager, 2012. URL http://www.ibm.com/software/integration/business-process-manager. (Cited on page 32)

[IBM12b] IBM Corporation. IBM Official Site, 2012. URL http://www.ibm.com. (Cited on page 27)

[IBM12c] IBM Corporation. IBM Workload Deployer, 2012. URL http://www.ibm.com/software/webservers/workload-deployer. (Cited on page 32)

[JHMO07] M. Johnson, A. Hately, B. Miller, R. Orr. Evolving Standards for IT Service Management. *IBM Systems Journal*, 46(3):583–597, 2007. (Cited on page 18)

[JSO12] JSON.org. JavaScript Object Notation, 2012. URL http://www.json.org. (Cited on page 25)

[JUn12] JUnit Community. JUnit Official Site, 2012. URL http://www.junit.org. (Cited on page 59)

[Ley09] F. Leymann. Cloud Computing: The Next Revolution in IT. In *Photogrammetric Week '09*, pp. 3–12. Wichmann Verlag, 2009. (Cited on page 15)

[LFM+11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, S. Dustdar. Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. *International Journal of Cooperative Information Systems*, 20(3):307, 2011. (Cited on page 28)

[LKN+09] A. Lenk, M. Klems, J. Nimis, S. Tai, T. Sandholm. What's inside the Cloud? An Architectural Map of the Cloud Landscape. In *Software Engineering Challenges of Cloud Computing, 2009. CLOUD'09. ICSE Workshop on*, pp. 23–31. IEEE, 2009. (Cited on page 15)

[Loo11] J. Loope. *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011. (Cited on pages 9 and 22)

[MG11] P. Mell, T. Grance. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*, 2011. (Cited on pages 8, 15 and 16)

[MLM11] D. Milojičić, I. Llorente, R. Montero. OpenNebula: A Cloud Management Tool. *Internet Computing, IEEE*, 15(2):11–14, 2011. (Cited on page 27)

[NS11] S. Nelson-Smith. *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc., 2011. (Cited on pages 18, 19 and 25)

[Obj12] Object Management Group, Inc. (OMG). Business Process Model And Notation (BPMN) Version 2.0, 2012. (Cited on pages 31, 44 and 56)

[Ope12] OpenStack Foundation. OpenStack Official Site, 2012. URL http://www.openstack.org. (Cited on pages 27 and 33)

[Ops12a] Opscode, Inc. Chef Community, 2012. URL http://community.opscode.com. (Cited on pages 23, 24, 42, 56 and 66)

[Ops12b] Opscode, Inc. Chef Customers, 2012. URL http://www.opscode.com/customers. (Cited on pages 19, 23 and 46)

[Ops12c] Opscode, Inc. Chef Documentation, 2012. URL http://wiki.opscode.com/display/chef/Home. (Cited on pages 20, 23, 24, 36, 46 and 73)

[Ops12d] Opscode, Inc. Chef Official Site, 2012. URL http://www.opscode.com/chef. (Cited on pages 19, 23, 24, 44 and 55)

[Ops12e] Opscode, Inc. Spiceweasel, 2012. URL http://wiki.opscode.com/display/chef/Spiceweasel. (Cited on page 25)

[Ora12] Oracle Corporation. MySQL Official Site, 2012. URL http://www.mysql.com. (Cited on page 20)

[Org12a] Organization for the Advancement of Structured Information Standards (OASIS). Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee, 2012. URL https://www.oasis-open.org/committees/tosca. (Cited on pages 26, 28, 45, 60, 64 and 76)

[Org12b] Organization for the Advancement of Structured Information Standards (OASIS). Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification Draft 04, 2012. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.html. (Cited on pages 8, 11, 26, 27, 28, 29, 30, 40, 44 and 61)

[Org12c] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language (BPEL) Version 2.0, 2012. (Cited on pages 31, 44 and 56)

[Pup12a] Puppet Labs, Inc. Marionette Collective, 2012. URL http://docs.puppetlabs.com/mcollective. (Cited on pages 23 and 25)

[Pup12b] Puppet Labs, Inc. Puppet Customers, 2012. URL http://puppetlabs.com/customers/companies. (Cited on pages 19 and 46)

[Pup12c] Puppet Labs, Inc. Puppet Documentation, 2012. URL http://docs.puppetlabs.com. (Cited on page 22)

[Pup12d] Puppet Labs, Inc. Puppet Forge, 2012. URL http://forge.puppetlabs.com. (Cited on pages 23, 42, 56, 66 and 76)

[Pup12e] Puppet Labs, Inc. Puppet Official Site, 2012. URL http://puppetlabs.com/puppet/what-is-puppet. (Cited on pages 19, 22, 44 and 55)

[Rub12] Ruby Community. Ruby Programming Language, 2012. URL http://www.ruby-lang.org. (Cited on page 25)

[Sal04] M. Sallé. IT Service Management and IT Governance: Review, Comparative Analysis and their Impact on Utility Computing. *Hewlett-Packard Company*, 2004. (Cited on pages 8, 17, 18 and 19)

[SAP12] SAP AG. SAP Official Site, 2012. URL http://www.sap.com. (Cited on page 27)

[Sha11] E. Shamow. Devops at Advance Internet: How We Got in the Door. *IT Journal*, p. 14, 2011. (Cited on page 16)

[Smi11] D. Smith. Hype Cycle for Cloud Computing, 2011, 2011. (Cited on page 17)

[SNS07] S. Sengupta, H. Nellitheertha, S. Sundarrajan. Service Oriented Infrastructure. *Managing the Hype*, pp. 21–28, 2007. (Cited on page 18)

[Sug12] SugarCRM Inc. SugarCRM Official Site, 2012. URL http://www.sugarcrm.com. (Cited on page 36)

[TCST09] W. Tan, A. Cater-Steel, M. Toleman. Implementing IT Service Management: A Case Study Focussing on Critical Success Factors. *Journal of Computer Information Systems*, 50(2):1, 2009. (Cited on page 18)

[The12a] The Apache Software Foundation. Apache HTTP Server Project, 2012. URL http://httpd.apache.org. (Cited on page 37)

[The12b] The PHP Group. PHP: Hypertext Preprocessor, 2012. URL http://www.php.net. (Cited on page 36)

[VMw12] VMware, Inc. Virtualization and Cloud Management, 2012. URL http://www.vmware.com/solutions/virtualization-management/index.html. (Cited on page 27)

[VRMCL08] L. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008. (Cited on page 15)

[WCEH09] D. Winniford, S. Conger, L. Erickson-Harris. Confusion in the Ranks: IT Service Management Practice and Terminology. *Information Systems Management*, 26(2):153–163, 2009. (Cited on page 18)

[Wor12a] World Wide Web Consortium (W3C). XML Path Language (XPath), 2012. (Cited on page 61)

[Wor12b] World Wide Web Consortium (W3C). XML Schema, 2012. (Cited on page 42)

[YAM12] YAML.org. YAML Ain't Markup Language, 2012. URL http://www.yaml.org. (Cited on page 25)

[Zam12] D. Zamboni. *Learning CFEngine 3: Automated System Administration for Sites of Any Size*. O'Reilly Media, Inc., 2012. (Cited on pages 9, 20 and 21)

All links were last followed on October 21, 2012.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Johannes Wettinger)

IBM Confidential