Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 3567

# Integrating Cloud Service Deployment Automation With Software Defined Environments

Darsana Das

| | |
|---|---|
| **Course of Study:** | INFOTECH |
| **Examiner:** | Prof. Dr.-Ing. habil. Bernhard Mitschang |
| **Supervisor:** | Dipl.-Inf. Tim Waizenegger |
| **Commenced:** | September 19. 2013 |
| **Completed:** | March 21. 2014 |
| **CR-Classification:** | D.2.13, K.6 |

# Abstract

The last decade has seen a deluge of technology and services related to cloud -computing. The idea of traditional IT infrastructure has been usurped by a rush to move all services to a cloud infrastructure and to implement them in a cloud-native way.

A cloud infrastructure is unlike a conventional infrastructure in that, it is implemented as a Software-Defined Environment (SDE) which abstracts and virtualises the underlying physical resources. It manages physical resources and provides virtual resources that process, manage, store, create networks and provision services. But while the concept of a virtual data center as played out by an SDE seems ideal, the issues it brings in implementation are quite a few. For one, deployment and management of services on such a large scale is difficult. It is also subject to human efficiency and error. So, new levels of tracking and automation have to be explored in parallel to support and sustain the cloud-computing phenomenon.

Several configuration management and orchestration tools have been developed over years, to fulfill the need of automated deployment and management. TOSCA is one of such that provides a specification which empowers this goal by using a metamodel. It defines implementation artifacts that encapsulate functionality of external services to use them during deployment. An open-source container called OpenTOSCA, is under development at the University of Stuttgart. This framework can process the TOSCA specification.

In this thesis, OpenStack has been employed as the cloud infrastructure provider. To achieve our goal, an implementation artifact for the relevant OpenStack components has to be developed in concept and implementation. In order to create useful implementation artifacts an appropriate level of abstraction has to be introduced between the low-level OpenStack API and the services that the implementation artifact provides. Accordingly, a TOSCA artifact is designed and implemented as a web service and packaged in a container file, called CSAR. This file, when processed by the OpenTOSCA container, causes the automated provisioning of a server on OpenStack. To evaluate the nature of the implementation artifact, the web service is hosted on the local machine and tests are performed to check for correct implementation of service endpoints. The results of these invocations are later presented.

## Acknowledgements

It is with great pleasure that I wish to acknowledge the efforts of those who made this thesis possible. Foremost, I wish to express my sincere gratitude to Prof. Dr.-Ing. habil.Bernhard Mitschang for giving me an opportunity to work in the Department of Application Systems on a topic that interests me.

My deepest gratitude is towards my supervisor, Dipl.-Inf. Tim Waizenegger for the motivation, guidance and the extraordinary patience that he has displayed in helping me to complete this work. Without his encouragement and involvement, this thesis would not have been possible. I was indeed very fortunate to have been mentored by him.

A special thanks to my friends- Sukanya and Patrick- for having been there to help me out with all my doubts and queries while I was finishing up my report. Their timely advice has saved me from quite a few serious issues while writing. And to my other friends who made the stress bearable and work seem so much enjoyable, a big thank you.

Last but not the least, I wish to thank my family for being a constant support in everything I have done in life.

# Contents

# List of Figures

# Chapter 1

# Introduction

The funny thing about technology is that the moment one thinks it has reached the highest rung of a ladder, it amazes one by showing an even higher rung. The ladder is never-ending.

And as companies climb this proverbial ladder today, they find their heads in the cloud. Quite literally, for cloud computing is currently, the best known answer to IT's long standing need for maintenance of resources and reduction of expenses. In a field where initial investments are high and infrastructure is nowhere used to full capacity, the concept of computing as a utility comes as a highly valued, highly desired phenomenon.

If we take the case of an ordinary IT organization, there are several software and hardware components that need to be installed, configured and maintained on a regular basis. Most of these components are proprietary meaning that the admin has to procure licenses from the providers and periodically update them. This is a pricey venture and most small firms often struggle to survive with high costs. The use of electronic processing paves the way for generation of tons of data that need to be securely stored and efficiently handled. For example, a bank conducts millions of transactions, a production environment produces huge amounts of data per day. Processing such bulk data requires an expensive setup, not only in terms of physical infrastructure but also in funding of research and development. Thus arose the need to time-share resources. This is in turn gave birth to the concept of distributed computing and the evolution of the world wide web which have enabled the use of so many online applications today [Sta10].Cloud computing takes resource-sharing to the next level.

With its "on-demand" policy, cloud computing facilitates the provisioning of components-both hardware and software as and when needed. A cloud user has access to a shared pool of resources. So whether it's a remote server with a particular bandwidth or a software suite, multiple users/companies can now easily acquire what they require, all through a well-organized interface that requires little or no administration effort on the part of the customer. Apart from the obvious advantage of reducing costs and maintenance complexity, it also cuts across boundaries of time and geographical space via its multi-user facility.

However, the process of migrating from traditional environments to the cloud was not a simple one. Portability of software is not easy so most companies either developed their own private clouds or customised applications to suit a particular cloud developed by another company. This resulted in cloud applications being dependent on a specific hosting provider's infrastructure. Expenses shot up and problems caused by single vendor lock-in began to arise. Thereafter, sprang the existence of cloud-native application providers which are companies that build their services from the very onset, to run on the cloud. The significance of their presence is that they provide solutions to specific business problems without pegging a customer to accept or use services which are either not required or simply not the best of what is available. Such applications are independent of any hardware and can live and function across multiple clouds. Cloud-native applications, thus, epitomize the key properties of a cloud environment-elasticity and scalability [1].

Unlike conventional infrastructures, a cloud infrastructure is ideally implemented as a "software-defined-environment"(SDE). This SDE abstracts and virtualises the underlying physical compute resources. Different applications and services are then deployed on this infrastructure. Provisioning such an infrastructure and configuring services on it is a long, tedious process which is repeatable and deterministic. Also with growing number of instances, the process of managing them becomes cumbersome. Hence, automation of both deployment and management are highly desired.

TOSCA provides a specification that enables automated deployment and management of resources on the cloud via the use of Templates. These Templates are reusable and provide a lot of flexibility to cloud service consumers and developers in selecting cloud providers. The TOSCA language provides a meta-model for describing the topology and orchestration processes of cloud services in a formal, machine-readable way. A container for that language reads and executes the service definition and deploys a service instance onto a cloud infrastructure. An open-source container for the language TOSCA is currently being developed at the University of Stuttgart. In this thesis, an attempt has been made to use the SDE OpenStack as infrastructure provider for deploying a server automatically via the OpenTOSCA container.

## Scope of Work

The aim of this thesis is to succesfully automate the instantiation of a VM created by OpenStack, within the OpenTOSCA container. A TOSCA Node Type is defined for the same purpose, along with its consituents and their associated functionalities. The management of the whole lifecycle of a VM is attempted to be automated. Apart from the traditional lifecycle implementation, an alternative specification for deploying the Node Type is also provided.

## Outline

This thesis is structured as explained below:

Chapter 1-Introduction

[1]http://venturebeat.com/2012/12/03/cloud-native-app-services/

A brief explanation of the need for cloud computing and a motivation for the use of an automated configuration management tool are provided. Also, the scope of this thesis is outlined.

Chapter 2-Background Concepts

The philosophy and the concepts that are central to the idea of cloud computing are introduced. Additionally, the technologies that have been used to realise the goal of this thesis are delineated.

Chapter 3-Related Work

The evolution of automation in IT through various tools and technologies is elucidated. A brief assessment of some of these past and current mechanisms is also provided.

Chapter 4-Concept and Implementation

The OpenStack API and the OpenTOSCA features that have been explored for the development of a Template, are examined. The basics of the implementation are also highlighted.

Chapter 5-Evaluation

A concise evaluation of the implementation with the help of some test results is presented in this chapter.

Chapter 5-Summary and Future Work

A brief summary of the work in this thesis is provided. The scope for future work is also discussed.

# Chapter 2

# Background

This chapter aims to give a simple explanation of the fundamental concepts and technologies used in the thesis. Cloud computing and its desirable properties-elasticity and scalability are elucidated. A brief insight into the IaaS provider, OpenStack, and the automated deployment offered by TOSCA and OpenTOSCA, are also provided.

## Cloud Computing

Broadly speaking, cloud computing refers to a distributed system of resources and services that communicate across boundaries of geographical and ethernet space. But that said, the cloud is not just a collection of resources because it possesses the capability to manage them as well. A cloud computing platform is responsible for provisioning, deprovisioning and monitoring servers and balancing their workload. Servers in the cloud could be both physical and virtual machines. Other than those, cloud platforms also provide a slew of applications, storage services, network filters etc. These offerings are billed on a consumption basis. Everything is offered as a utility, thus encompassing the traditional pay-as-you-go subscription model for services.

A major characteristic of the cloud is that its services are transparent, meaning, the user need not be concerned about how the service is made available to him. All he needs is to know how to use it. And this is usually simplified by interactive User Interfaces. Most services on the cloud require little or no administrative knowledge on the part of the customer. An access is simply enabled on the request of a customer. The latter has a view of only the abstraction of the service he needs- the details of implementation and accessibility are hidden from him. The cloud is thus a stack of different services which can be classified as Software as a Service(SaaS), Platform as a Service(PaaS) and Infrastructure as a Service(IaaS).A brief overview follows[1].

*IaaS - Infrastructure as a Service*: Cloud providers of IaaS offer infrastructure starting from the simplest and most basic of resources like physical and virtual machines to complicated storage, network and operating systems as utilities. A hypervisor node is responsible for the running of these virtual machines. Clusters of such hypervisors administer the functioning of hundreds of virtual machines, thus serving the purpose of scaling in cloud services. Users

[1]http://blog.cloudclickware.com/2013/06/05/cloud-computing-saas-vs-paas-vs-iaas/

need to buy access rights to infrastructure components they need, without having to buy the actual components. Not only does this cut costs, it also rids them of the pains of infrastructure management. Such a cloud service is ideal for organizations that have low capital or require temporary infrastructure. IaaS can further be divided into private and public clouds–the former refers to resources on a private network while the latter refers to resources on a public network available to external users. Amazon Web Services is the most renowned instance of IaaS.[2]

*PaaS - Platform as a Service*: This offers an integrated environment that would allow application developers or the like to create, deploy and test applications without having to worry about the cost and complexity of underlying hardware and software infrastructure. A computing platform would typically include atleast an operating system, programming language execution environment, database, and web server, mostly with integrated test and deployment environments. Force.com from Saleforce.com is a renowned PaaS system that empowers developers to develop multitenant applications hosted on Salesforce.com servers.

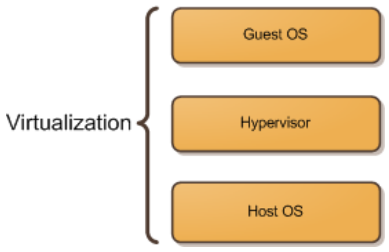
*SaaS - Software as a Service*: This brand of cloud computing offers a single application built on a multi-tenant architecture, over the internet to thousands of customers. The software is provided on-demand, either free of cost or on a pay-per-use basis. All updates and security patches are taken care of by the provider. For the customer, this implies no efforts to support or maintain software, thus reducing significant costs, while for the provider, this means reduced management effort as it is only one app at a central location, needing administration. Google Apps is one of the best known SaaS offerings.

## Virtualization

Virtualization is a mandatory requirement of cloud computing. It is an old concept, but in recent times, it has been explored in great detail and a lot more diversity to fulfill the demands of providing cloud services and resources as a utility.

Virtualization has asserted its importance in three major fields of IT- network virtualization, storage virtualization and server virtualization[3]. A virtual network is a software-based realisation of the hardware and software processes that combine to perform the functionality of a real, physical network. Network hardware such as switches and adapters, media like ethernet cables, elements such as firewalls and networks such as virtual LANs are the key components of a virtual network. Network virtualization intends to improve the efficiency, productivity and scalability of a network by performing many of the network administration tasks automatically, thus making life easier for the administrator. It enables the easy subdivision of an existing network into many smaller, virtual networks as also the combination of separate local systems into a VLAN[4]. Resources such as files, images, folders etc. can be easily managed from a central physical site.

[2]http://blog.cloudclickware.com/2013/06/05/cloud-computing-saas-vs-paas-vs-iaas/
[3]http://searchservervirtualization.techtarget.com/definition/virtualization
[4]http://en.wikipedia.org/wiki/Network_virtualization



[a]

**Figure 2.1:** Virtualization

[a]http://itechthoughts.wordpress.com/2009/11/10/virtualization-basics/

Storage virtualization refers to the pooling of physical storage devices from multiple networks, and creating the impression of a single storage system that can be accessed centrally. There is a way of allocating storage dynamically and of different sizes, by means of a physical-virtual space mapping function. The idea is to abstract physical location of data such that end users can access it without regard to actual physical space. This also helps system administrators to manage, archive and recover storage more flexibly.

Server virtualization refers to partitioning of a single server into multiple, smaller servers with the help of special software. This enables maximising server resources-the number and identity of individual physical servers, processors, and operating systems-which are masked from the end user. The end result is enhanced resource sharing and full capacity utilization.

Three very important concepts need to be explained in the context of server virtualization: the host operating system, the hypervisor, and the guest operating system. The host operating system provides the environment and physical resources to host the resource or instance for virtualization. The guest operating system is the one that is installed inside the host and functions on its own. It could be significantly different from the host operating system shown in Fig 2.1. The hypervisor is also called the Virtual Machine Manager (VMM), it is a software or firmware that runs on all the virtual machine instances, enabling them to function concurrently on the same hardware host, sharing its resources and memory. The hypervisor ensures the right allocation of memory and resources and a smooth working of all virtual machines.

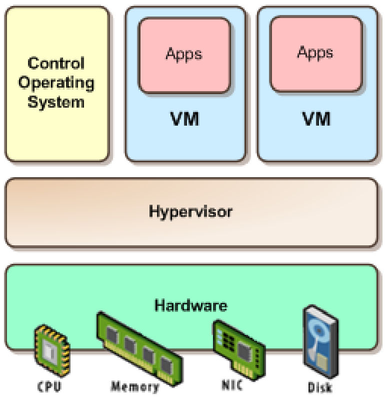
### Hypervisor

Hypervisors are of two types: native or bare metal and hosted[5].

Type 1, or native/bare metal hypervisors are those which run directly on the hardware and monitor guest operating systems. The term "bare-metal virtualization" arises from this concept of the hypervisor being placed directly on the hardware. The lack of any layer in between the hardware and the hypervisor eliminates the overhead of a host OS, hence increasing speed. This is the reason bare-metal virtualization is the preferred choice of technology for data centers, shown in Fig 2.2.

[5]http://en.wikipedia.org/wiki/Hypervisor

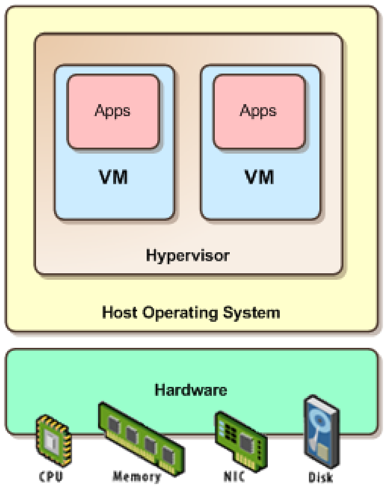
Type 2, or hosted hypervisors are those which run within a conventional operating system, thus forming the second layer, on top of which, the guest operating systems run. This runs in most client-side virtualizations such as Microsoft's Virtual PC, and VMWare's Workstation. It is depicted in Fig 2.3.



[a]

**Figure 2.2:** Hypervisor Type 1

[a]http://itechthoughts.wordpress.com/2009/11/10/virtualization-basics/



[a]

**Figure 2.3:** Hypervisor Type 2

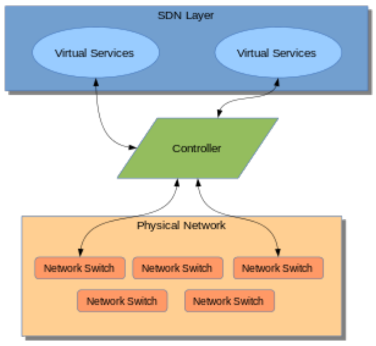
[a]http://itechthoughts.wordpress.com/2009/11/10/virtualization-basics/

## Software-defined computing

When it comes to a constantly evolving technology as the cloud, no new idea can claim to be the ultimate solution. Software-defined infrastructure revolutionises the concept of virtualization by shifting the idea of hardware and hands-controlled infrastructure to one controlled merely by software. In traditional infrastructure systems, resources are assigned manually to workloads. A software defined environment seeks to make everything programmable and automatic. Typically it comprises the entire computing infrastructure-starting with servers, it has been extended to include storage and network facilities.

In an SDE, virtual machines are automatically defined with the desired configuration and capacities and workloads are assigned to these machines by the system. Over and above this basic automatic creation and assignment, the system also takes care of dynamic optimisation and reconfiguration to address issues in infrastructure that might crop up. And all of this is regulated by policy-based compliance checks and enhanced by updates [6].

Software Defined Networking (SDN) is an emerging approach to controlling/managing network traffic and routing by the use of special software without having to physically handle network switches and routers. The programmability of the software abstracts the underlying network infrastructure, thus shifting the traditional handling of data flow and control from switches, to a Controller software. This provides flexibility in changing the topology of the network with much reduced effort, making the life of network administrators quite easy, particularly, in a cloud environment. OpenFlow is the most common network protocol used in SDN. Administrators can now change traffic flow and deploy services through the use of APIs offered by SDN that support all services and applications running over the network[7].



[a]

**Figure 2.4:** Software Defined Networking

[a]http://en.wikipedia.org/wiki/Software-defined_networking

The concept of Software Defined Security (SDSec) arises from a need to make SDN strong and secure. This is a virtualization of traditional network security functions, provided by firewalls, load balancers which are now decoupled from hardware appliances so that they run in software. Adaptive behavior, such as the automatic activation of a firewall on detection of a security threat, is enabled in the software layer that abstracts the security functionality[8].

[6]https://www-304.ibm.com/partnerworld/wps/servlet/ContentHandler/pw_com_sol_software-defined-environment
[7]http://www.sdncentral.com/what-the-definition-of-software-defined-networking-sdn/
[8]http://www.sdncentral.com/security-challenges-sdn-software-defined-networks/

Software Defined Storage is a next-generation form of data storage, in which the programming that controls storage-related tasks has a separate existence from the physical storage hardware. Simply put, storage infrastructure is managed by special software such that at the time of assignment, resources are allocated from the pool according to specific needs, without having to worry about physical storage constraints. This makes the process of allocation and storage space administration efficient and convenient. Moreover, SDS provides functionality like deduplication, replication, thin provisioning, snapshots in addition to backup and restore capabilities across a wide range of server hardware components[9].

Extending the concepts of virtualization, abstraction, pooling and automation to network, security and storage, gives birth to a new concept - Software Defined Data Center. The idea is to have a self-sufficient infrastructure or platform, replete with services - networking, storage, compute etc - that can conveniently and efficiently deliver Information Technology as a service. Still in its nascent stages, the SDDC has been envisioned as a future necessary foundation for efficient, scalable cloud computing[10].

## Scalability

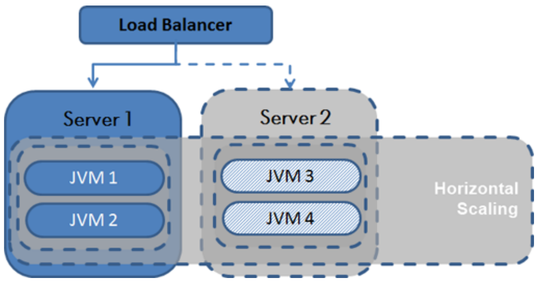
Scalability is defined as that property of a system or process that enables it to handle a specific amount of work and to continue working smoothly and proportionately after enhancements have been made to the existing setup. A system is called scalable if it is capable of functioning evenly or in an improved manner on addition or removal of features. In other words, the overhead incurred on change of configuration should not hamper the running of the existing system.

Scaling is of two types: vertical and horizontal.

Horizontal scalability is the ability to connect multiple hardware or software entities such that they function as a single logical entity. In a distributed environment, this could mean adding a new computer. So, if an application was originally deployed on a single server, then, on horizontal scaling, several servers have the application deployed on them. Scaling out an application horizontally across multiple servers implies that the failure of a single server cannot bring down the system. In high performance computing applications like seismic analysis, the workload which was originally handled by supercomputers has been shifted to a cluster of low-cost computers, whose aggregate computing power is higher than that of a traditional RISC processor[11]. However, managing so many nodes is a complex task. Synchronization problems are common. For e.g., while scaling out large databases, the primary concern is consistency amongst different clusters of data stores. Thus, hand-in-hand with horizontal scaling, arises the need for software that manages multiple nodes as well as shared storage space, amongst other things. Fragmentation is an important concept in horizontal scaling. It happens because companies, in an attempt to scale out, use special-purpose servers rather than

[9]http://www.webopedia.com/TERM/S/software-defined_storage_sds.html
[10]http://en.wikipedia.org/wiki/Software-defined_data_center
[11]http://en.wikipedia.org/wiki/Scalability

all-purpose ones. It means that machines in such an environment, service particular requests, leaving the other requests for other machines to take care of. Say, in a web-based business application, certain servers handle the placing of orders while others handle the transaction processing of customers. The workload however, is not even throughout the day. While one set of servers remains busy during one time of the day, the other set remains idle. The situation reverses during times of the day, depending on the workload. This condition, where the usable capacity of the system is cleaved or segmented, is called fragmentation[12].



[a]

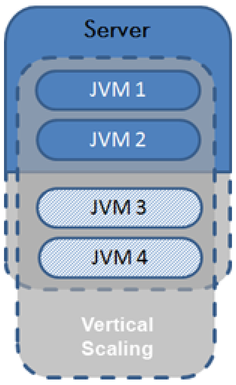
**Figure 2.5:** Horizontal Scaling

[a]http://javabook.compuware.com/content/intro/performance-and-scalability.aspx

Vertical scalability is the ability of increasing the capacity of a single entity in the system, for example a node/computer, by the addition or removal of features. This is also called scaling up or scaling down. The addition of hardware or software to an existing system can be limited by several factors, primary amongst them, the features/capabilities of the current system setup. The existing hardware might not support the addition of more hardware or software. Addition of resources include more or high-performing CPUs, extra memory or shared resources in case of virtualized instances. For e.g., a big piece of software, like a spreadsheet program requires more memory and processing power to function, so we add CPUs to handle it. But to add CPUs, there must be sufficient number of free sockets in the mainboard [Liu13]. Adding more CPUs means that each has to share its resources (disk space, bus, cache etc.) with the other CPUs.

To understand which type of scaling to use in a scenario is very important. Every system is vertically scalable, limited mostly by existing hardware design. However, applications that do not scale up well, should not be made subjects for vertical scaling when they are deployed on the cloud. They end up being disastrously expensive because the demand for resources on the cloud is very high. Horizontal scaling again requires some kind of inherent distribution nature in the system. If a system can function smoothly after extending the number of nodes, then it can be enlarged to accomodate as many nodes as possible. Also, these additional nodes need not possess high-computing abilities, deploying low-cost nodes works fine in a horizontal scalable system. Hence, for large-scale architectures, horizontal scaling is more economical [the12].

[12]http://www.practicingsafetechs.com/TechsV1/Scaling/



[a]

**Figure 2.6:** Vertical Scaling

[a]http://javabook.compuware.com/content/intro/performance-and-scalability.aspx

## Elasticity

Elasticity is a key characteristic of the cloud paradigm-the term, owing its origins to physics and economics. Often it is confused with other cloud computing terms-scalability and efficiency. Various institutes and organizations have tried to define it, leading to fuzziness of the definition. Broadly speaking, elasticity is the ability of a system to dynamically provision and de-provision resources so as to meet the demands of a changing workload. By the definition given by National Institute of Standards and Technology, the resources available for provisioning should appear to be unlimited to the consumer such that they(resources) can be allocated as and when the need rises.

For instance, a service provider wishes to host a website on a cloud[13]. Initially the site is not very popular and a single server might be sufficient to serve site users. Now, owing to an online advertisement, a lot of interest is generated in the website and it is suddenly flooded with visitors. One web server can no longer do the needful. More servers are needed. If the system is elastic, then this requirement is immediately detected and additional servers are provisioned to serve all users. If the host cloud is public, then elasticity is easily achievable because public clouds do not place too many constraints on size or location, thus illusioning consumers into the impression of unlimited resources. However, if the host cloud is private, there is likely to be a fixed capacity, hindering elasticity.[LB12] Hence, elasticity is restrained by the availability of resources as well as the state of infrastructure.

Elasticity aims to correctly match needs with requirements in a dynamically, evolving system. Overprovisioning of resources should be avoided else costs will run high unnecessarily. Underprovisioning of resources, on the other hand, might completely cause the system to stop functioning. But for all its usefulness, the concept of elasticity still lacks appropriate benchmarking and no standard metrics exist to quantify elasticity for comparing systems.

[13]http://en.wikipedia.org/wiki/Elasticity_cloud_computing

Herbst et al. attempt to give a clear definition of elasticity and propose a benchmarking methodology using certain metrics in this paper [ela13].

## OpenStack

OpenStack is a large-scale, open source cloud computing project, predominantly offering Infrastructure-as-a-Service, that aims to build public and private clouds. Founded by Rackspace and NASA, OpenStack is supported by a global collaboration of more than 200 companies-HP, Arista Networks, AMD, Cisco, Dell, EMC, Ericsson etc, to name a few-and managed by the OpenStack Foundation. The project has a six-month release cycle and the latest version(2013.2.1), codenamed Havana, was released on 17 December 2013.

### OpenStack Architecture

OpenStack consists of several interrelated components and each of them has an Application Programming Interface (API). A subset of any of these components provides Infrastructure-as-a-Service (IaaS). These nine components along with their codenames are as follows[14]:

Dashboard(Horizon)-Web-based GUI for users to access, provision and automate cloud services

Compute(Nova)-Provisions and manages virtual machines on demand

Networking(Neutron)-Provides networking facilities to other OpenStack components

Object Storage(Swift)-API-accessible, scale-out storage platform for storing and retrieving files

Block Storage(Cinder)-Provides persistent block storage to Compute instances

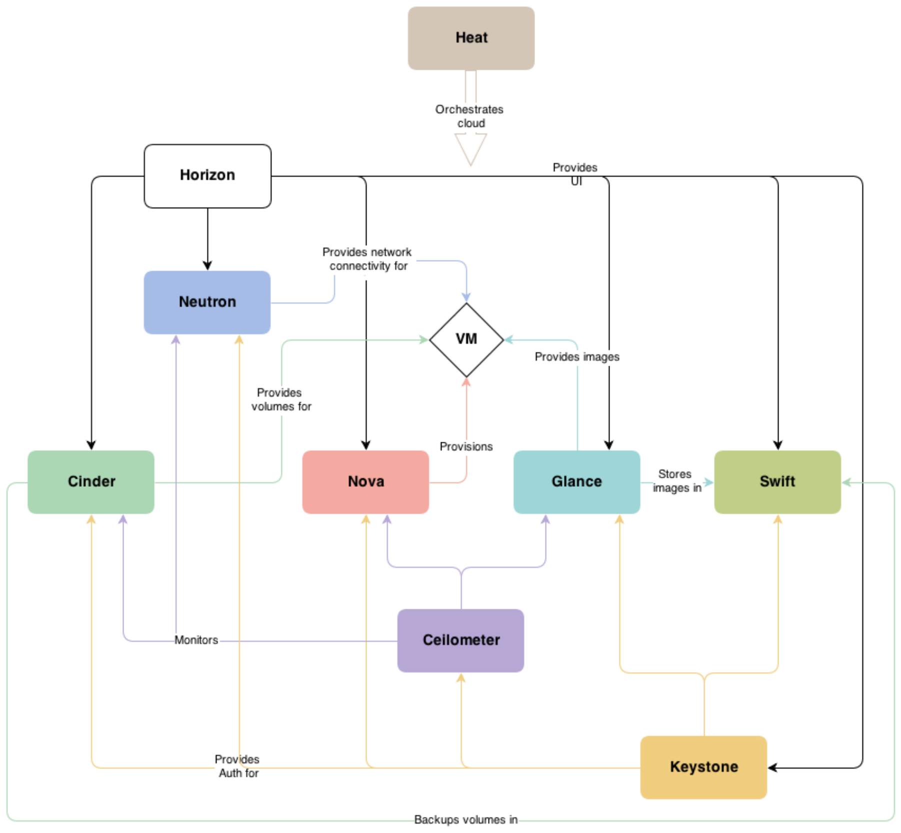
Identity Service(Keystone)-Provides authentication and authorization for other OpenStack services

Image Service(Glance)-Stores and retrieves disk images for provisioning VMs

Telemetry Service(Ceilometer)-Takes care of billing, benchmarking etc by monitoring OpenStack cloud

Orchestration Service(Heat)-Orchestrates multiple composite cloud applications using templates

[14]http://docs.openstack.org/admin-guide-cloud/content/ch_getting-started-with-openstack.html



*a*

**Figure 2.7:** OpenStack Architecture

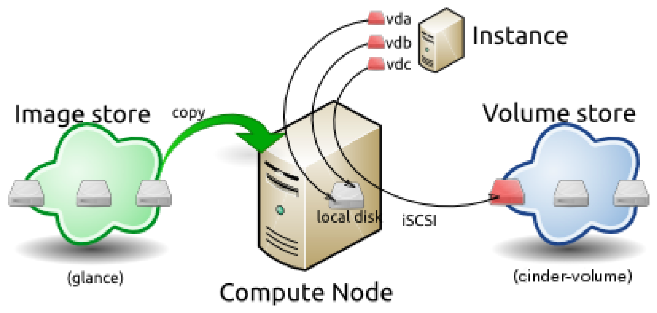
[a]http://docs.openstack.org/admin-guide-cloud/content//ch_getting-started-with-openstack.html

## OpenStack Functionality

Nova is the brain behind the entire system. It is the controller, written in Python, and its architecture enables horizontal scaling with no consideration for proprietary hardware or software requirements. It is easily integrable with external systems, is designed to automate virtual machines and other resources, and can work well with virtualization technologies, bare metal and high-performance computing configurations[15].

Nova communicates with the Identity Service for authentication purposes, downloads images from Glance and launches them into instances, and interacts with the Dashboard for user interfaces. Nova facilitates a multi-tenant environment based on roles that determine access privileges for users and projects. Multiple hypervisors are supported by Openstack Compute, such as KVM, QEMU, Xen, Bare Metal etc, and the choice for the most appropriate one should be taken, based on considerations of budget, resource constraints and other technical requirements. Nova communicates with the hypervisor via API calls.

[15]http://en.wikipedia.org/wiki/OpenStack



[a]

**Figure 2.8:** Launching a VM

[a]http://docs.openstack.org/admin-guide-cloud/content/ch_introduction-to-openstack-compute.html

Glance, the image repository, stores and manages images which are templates for the creation of virtual machine instances. An instance, is thus, a running machine on a physical compute node. Several instances may be created from a single base image, as the creation does not modify the state of the base image. Terminating any of these instances also, does not delete the base image. The current disk state of an instance can be captured in a snapshot and used later, for the creation of further instances. These snapshots are managed by Nova.

Before launching an instance, it is required to choose a particular flavor. A flavor is nothing but a specification of the virtual resources needed to run an instance, i.e, the number of CPUs, the size of RAM and memory. A set of predefined flavors are provided by Nova for a particular cloud and the user needs to choose one.

Openstack provides two kinds of block storage: ephemeral and persistent volumes. Ephemeral storage is the flavor which is associated with a unique virtual machine instance and it exists until the termination of the instance. Persistent volumes on the other hand, are raw blocks of storage without partitioning or a file system. These are created by users depending on availablity and requirements. Upon being attached to an instance, these can be partitioned and used just as an external hard drive on a computer. Persistent volumes can be mounted to several instances but not concurrently.

As shown in figure 2.8, Glance provides a base image that is copied to the local disk of the node where the instance is going to be created. The local disk containing the image copy is called vda. An ephemeral storage, called vdb, is mounted to the node and its lifetime will be limited by the existence of the instance. A third block volume, from Cinder, is also attached to the node. This persistent volume contains data from all the instances that it has been mounted to earlier. Now, the resources specified by the flavor are instantiated and the virtual machine achieves a running state. On deletion of this instance, the ephemeral storage ceases to exist, the base image remains unchanged and the persistent volume has new data changes saved in it. All resources such as vCPU, memory, are released [$os_13$].

Heat is an Openstack project that aims to orchestrate the generation and running of cloud applications using templates. It integrates other core components of Openstack into templates which can be used to create VM instances, floating ips, security groups, nested stacks etc. Templates can specify the relationship between different resources so that provisioning resources for an application will happen in a structured manner. For any change in the infrastructure, one only needs to change the templates and launch the updates. The coupling with all Openstack core components expands their usability. Heat attempts to provide compatibility with AWS CloudFormation template but its own native templates integrate well with software configuration management tools-Puppet and Chef[16].

## TOSCA

TOSCA stands for Topology and Orchestration Specification for Cloud Applications. It is an OASIS standard, that is developed in a collaboration of multiple companies, chief among them being IBM and SAP. TOSCA is an XML-based specification to describe cloud applications and their automated deployment and management. It provides a standardized, modular approach to define the application's components, the relationships that exist amongst them and the operational behavior of these components [BBKL14].

The chief motivation behind the TOSCA project is to develop a standard for portability and interoperability of cloud services. With multiple vendors providing cloud services, proprietary APIs exist to access and manage these services. These APIs provide interfaces that are coupled with the inhouse software or platform. Migrating from one vendor to another requires a complete change of technology, which proves to be an expensive venture. This situation, commonly known as vendor lock-in, has been a major concern in the cloud computing world. TOSCA attempts to eliminate this issue by proposing a standard for the description and exchange of cloud applications. Service Templates, provided by TOSCA, are meant to be successfully operable in any TOSCA-supported environment, irrespective of which organization it was developed for or which provider it was developed by.

### TOSCA architecture

An explanation of the basic concepts behind TOSCA bifurcates into two main topics: Application topologies and management plans. An application topology is a structural description of the application's many services, their inter-relationships and operabilities, modelled in an XML-like language. This represents a Topology Template. A management plan, on the other hand, exposes the management capabilities of the various components, which put together, achieve the objective of automated deployment, configuration and management of the application as a whole.

### Application Topologies

The application topology is a colored graph where vertices/nodes represent the services or components and edges represent the relationships existing between them. Nodes and edges, in

[16]https://wiki.openstack.org/wiki/Heat

addition, contain information about the components, pertaining to their properties, functional and non-functional requirements as well as how to manage them, for e.g. how to deploy the component, how to scale it up in the application etc. Providing this description in a structured, machine-readable format is crucial to automation of application management. A template implements a type, in much the same way as a concrete class implements an abstract class. A topology template is composed of node templates and relationship templates, both realising their respective types, and provides reusable instances.

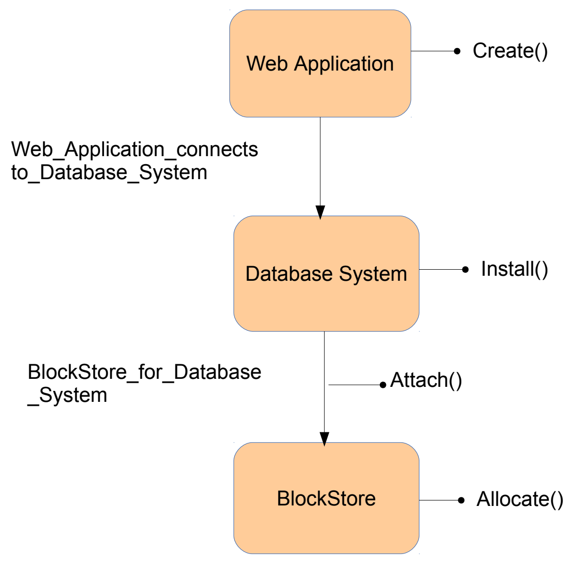
Application models in TOSCA are an abstraction of two layers: templates and types. A Node Type is a specification of the properties and operations of the component it represents. A Node Template simply references this type and adds constraints of its own, such as the number of occurences of the component. For instance, a web service might have three components- a server, a database and a process engine. Each of these is represented by a Node Type. Properties of the server Node Type may include an IP address, the interface may expose operations for starting and shutting down the server etc. The Node Template will provide a concrete implementation of all these properties, in addition to specifying the range of IP addresses for the server [oas12].

A Relationship Type defines the semantics of relationships between nodes. A Relationship Template specifies the direction of this relation with the help of a source and a target element, along with any constraints. For e.g., the process engine can have a "hosted on" relationship with the application server.

In Fig 2.9, the nodes Web Application, Database System and BlockStore are connected to one another via relationships Web_Application_connectsto_Database_System and BlockStore_for_Database_System.

An artifact is a representation of an executable. TOSCA defines two types of artifacts: Deployment Artifact and Implementation Artifact. A Deployment Artifact represents the executable of instances of a Node Template or a Relationship Template. For e.g., a web service may have a Java Web Archive as a Deployment Artifact(DA). This DA will contain all installables to get the service running. An Implementation Artifact is one that represents the executables of a Node Type or a Relationship Type. For e.g., an operation of a web server Node Type, say deploy, may be implemented as a Java Web Application and packaged as a WAR file [UB]. This is the Implementation Artifact.

To better explain the difference between DAs and IAs, a pictorial depiction is used in Fig 2.10. As one can see, after importing a service template into the environment, the IAs representing Node Type operations are deployed by the TOSCA container on the host environment. After that, these operations are bound to the artifact just deployed, i.e, they are readied for execution. These operations may now be invoked by implementations mentioned in the management plans. A few of these operations will process the deployment artifacts to instantiate Node Types in the target environment. For e.g., to launch a VM, multiple images(or a reference to those images) may be stored as a deployment artifact. The implementation artifacts will execute operations to launch the VM after processing the particular image that it needs, from the deployment artifact.



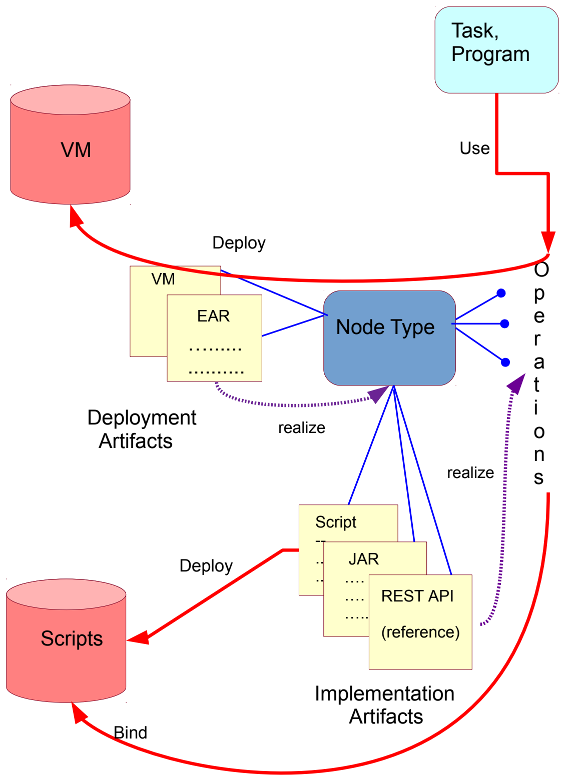
*a*

**Figure 2.9:** Application Topology

[a]http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html#_Toc347391541

### Management Plans

Every Node Type and Relationship type exposes certain management capabilities. For instance, a web application Node Type will contain information about how to configure, upgrade and backup the application, apart from deploying it on a server. Implementation Artifacts realise these operations and expose them via REST or WSDL [GB]. Plans come into the picture now, to interpret this data, and execute instructions in the correct sequence to create a running instance of the application. Apart from operations defined by the Node Types in the Implementation Artifacts, plans can also execute operations exposed by external services, which may or may not be facilitated by user interaction. For e.g., some tasks may require parameters for execution. These parameters could be obtained from users or they could be extracted from property files or values saved in the system.



a

**Figure 2.10:** Processing a service template

[a]https://www.oasis-open.org/committees/download.php/45871/TOSCA Issue 15 - Proposed Description of Artifacts.pdf

TOSCA utilizes existing workflow languages BPMN and BPEL to execute plan workflows. These languages expose their capabilities of parallel execution, monitoring, recovery etc. to TOSCA, thus making management plans very effective indeed. A significant benefit of BPMN or BPEL is the portability they offer, allowing TOSCA plans to be portable across different environments. Apart from this, those tasks which absolutely cannot function without human intervention, such as installing physical infrastructure, are also facilitated by these workflow languages [BBKL14]. Fig 2.11 shows a set of activities aimed at getting storage for an application.

In the absence of TOSCA, management tasks would be mostly manual, with only a few aspects handled by scripts. For composite applications having components from multiple vendors, orchestration becomes highly complicated.

*a*

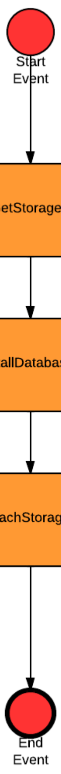


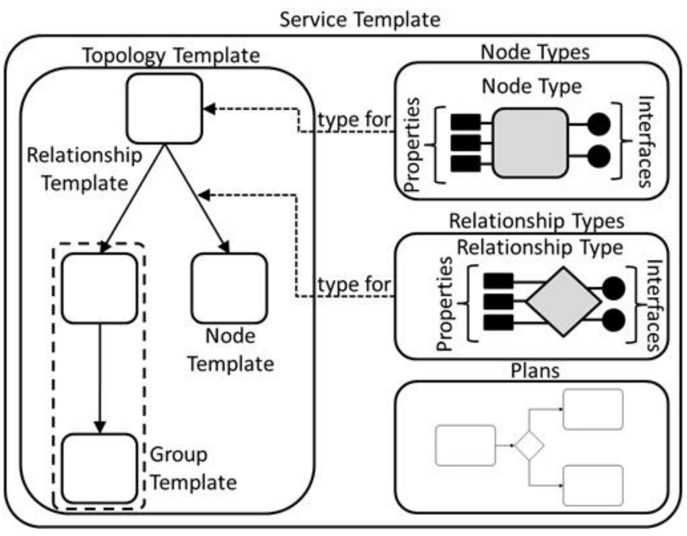
**Figure 2.11:** Management Plan

[a]http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html#_Toc347391541

### Packaging

A TOSCA Service Template is essentially composed of Application Topologies and Management Plans. Some Service Templates could also be composed or decomposed into other Service Templates, to allow further segregation of functions, based on need. For e.g., a Service Template for deploying an application on a server could have two sub-templates-one to deploy on a single node application tier, the other to deploy on a clustered application tier. [tos13b]

To make execution of a cloud application possible, the presence of all artifacts, along with the Service Template is required. Thus, the Implementation and Deployment artifacts and the Service Template are packaged together into an archive format called CSAR. This is not yet a



a

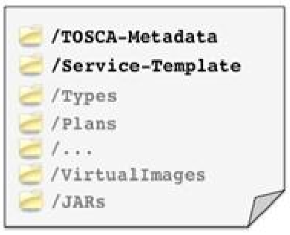
**Figure 2.12:** Service Template

[a]http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.html

standardized format but it serves the purpose of having a single file that contains all ingredients necessary to install the cloud application. This format is also portable across different TOSCA-supported runtime environments. Fig 2.12 shows a TOSCA Service Template.

A CSAR is typically a compressed, ZIP file. Its contents are organized in subdirectories, specific to an application. The presence of a subdirectory called TOSCA-Metadata is mandatory.

The figure 2.13 shows a typical CSAR structure.



a

**Figure 2.13:** Structure of a CSAR

[a]http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

## Imperative and Declarative processing

The Imperative processing model is one in which actions are performed in a sequence, according to instructions given, to achieve a predefined goal. In the case of TOSCA, imperative processing would imply the execution of management tasks-create, deploy, upgrade, terminate etc. in compliance with the workflow outlined in management plans. All management operations defined by Node and Relationship Types are made available via Implementation Artifacts. These operations are abstracted to higher level task definitions to be executed by plans. Thus operators, who are not involved with the development of plans, need not bother about technical details. The details are left to the developers to worry about and the operators need only to deploy the plans.

The Declarative Processing model, on the contrary, seeks to define a goal, based on properties or requirements and capabilities. A model is used to create an implementation or instance of the goal, but an interpreter is required to understand the semantics behind the symbolism used. In case of TOSCA, declarative processing implies that deployment and management logic is handled by the runtime. So the runtime should identify the operations to be deployed, interpret them correctly and be aware of the order in which they should be called. This requires tasks to be simple. For complicated management tasks, declarative processing is a poor idea [BBH+13].

## OpenTOSCA

OpenTOSCA is an open-source browser-based implementation of TOSCA that has been developed at the University of Stuttgart. In this thesis, version 1.1 has been used.

The standard terminologies and definitions of the original TOSCA project have mostly been retained in OpenTOSCA. But while TOSCA supports both declarative and imperative processing, OpenTOSCA presently supports only the latter. As in TOSCA, there exist Types and Templates to describe entities. All necessary assets are packaged in a container file that has a fixed layout with separate folders for each of the deployment artifacts (DEPL-ARTIFACTS), implementation artifacts (IMPL-ARTIFACTS), additional software assets like WARs, images or installers (IMPORTS) and the plan (PLANS). A Service Template exists in the root folder and it is wholly written in XML. The container file is nothing but a zipped archive, originally called THOR, but now renamed to CSAR. The workflow modelling languages used are BPEL and BPMN. While it was originally only possible to create the Build Plan, the current version supports the execution of the Plan as well.

OpenTOSCA allows the processing of CSAR files to set the management plans in flow. Once the plans take over, automatic provisioning and management of instances of an application are achieved.

### OpenTOSCA Architecture

An OSGi Framework enables the execution of the OpenTOSCA container. OSGi stands for Open Source Gateway Initiative; it is a set of specifications that can be used to build a component system in Java. These specifications facilitate a development model where applications composed of many reusable components can be created. Components can communicate with one another through the use of shared objects called services. The implementation details of components are hidden from view by the framework.

Modularity, which is at the core of the OSGi specifications, is the principal idea behind bundles or OSGi components. Bundles are more like JAR files, except for the fact that their contents are hidden to other bundles unless explicitly imported. These can be easily reused and extended in functionality. A bundle can instantiate an object, registering it in the service registry under one or more interfaces. Services are these common, registered objects that facilitate communication and sharing. Bundles for an application, are deployed on the OSGi Framework, which is the bundle runtime environment. There exist APIs that allow to start, stop, uninstall, update and remove bundles- making management possible[17].

There are three parts to the TOSCA architecture- Implementation Artifact Engine (IAE), Plan Engine (PE) and Controller. The Implementation Artifact Engine deploys all the implementation artifacts stored in the CSAR, and provides them to the management plans. The engine is developed on the basis of a plugin system that enables developing the artifacts in different programming languages. The Plan Engine is responsible for executing plans written in BPEL or BPMN. It also has to establish correct bindings with management operations provided by Implementation Artifacts. The third component, the Controller, is responsible for controlling all other components, and provides APIs to manage and install/uninstall CSAR files[18].

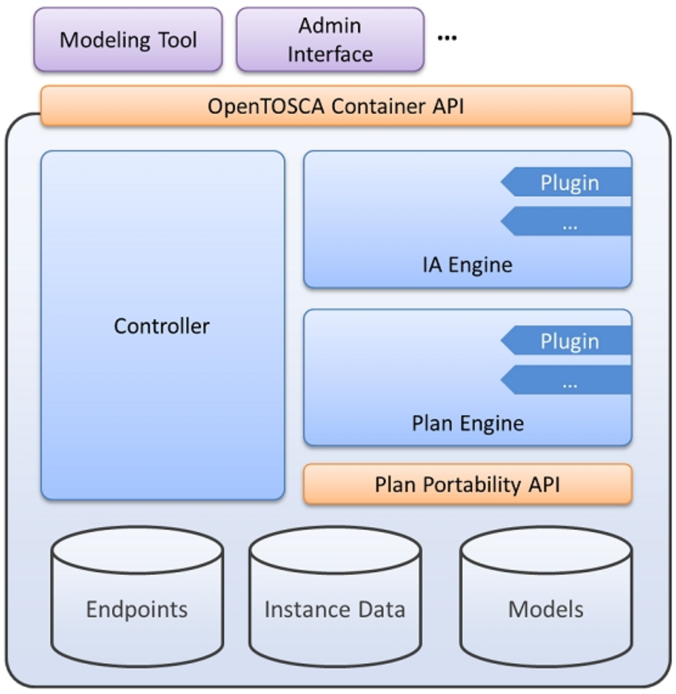
All REST-URIs have their root structure in the Container API. The File API is responsible for the uploading of new CSAR files and the extraction of their component files. The traversal through the CSAR file in the container is done by the TOSCA API. Both these APIs together form the Portability API. The Container API monitors the status of deployment processes and provides them with necessary parameters and methods when requested.

The Core Functionalities component acts as the central information store of the container. It interfaces with the local DB and file system, so it has the latest copy of changes with itself. This enables the published interface to keep functioning even if the internal interface changes. The Core Endpoint Service is in charge of managing the REST and WSDL endpoints. Post deployment of implementation artifacts, the IAE stores the endpoints in the Core Endpoint Service. The stored endpoints are picked up by the Plan Engine to update values in other plans. The Core File Service manages and extracts the CSAR files and saves the extracted files. The Core Model Repository manages application Service Templates, the Core Deployment Service tracks the status of deployment processes and the Core Capability Service is responsible for the IAE/PE plug-ins as well as those of the container. All changes in the data are centrally saved by an invocation by the Container API and they become available for access by the engines.

Implementation Artifacts, located in the CSAR, provide management operations of nodes and relationships. The Implementation Artifact Engine provides different plugins to handle different kinds of implementation artifacts. Once a plugin deploys a particular artifact, the engine returns the endpoint address of the operations invoked. These addresses are now stored in the Endpoints database.

[17]http://www.osgi.org/Technology/WhatIsOSGi

[18]http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php#architecture



[a]

**Figure 2.14:** OpenTOSCA Architecture

[a]http://www.iaas.uni-stuttgart.de/OpenTOSCA/

Plans, contained in the CSAR, are executed by the Plan Engine. The latter provides plugins suited to handle a particular workflow language and runtime. But the Plan Engine is unaware of the endpoints of the services deployed by the Implementation Artifacts. All it knows is the kind of services. Hence, the Plan Engine is responsible for providing the correct binding of the services with their respective endpoints. This information is retrieved from the Endpoints database. This scheme of binding ensures easy portability of management plans into different runtime environments whereas the plugin feature allows extensibility [BBH+13].

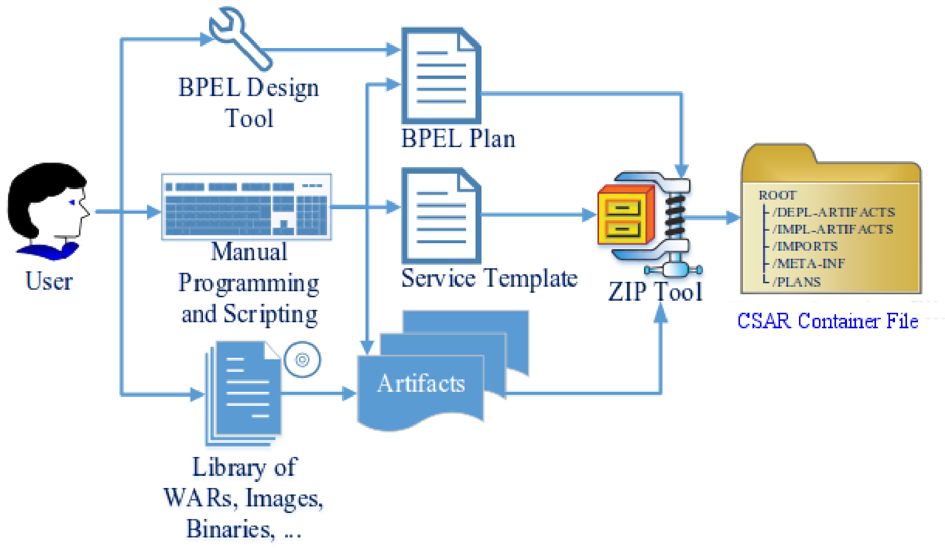
The TOSCA Engine serves the purpose of validating a TOSCA document and processing imports and references. It also holds the references that might be required in future.

The OpenTOSCA Controller sees to it that all needed components are available. It provides control features and is also responsible for the activation of both the engines.

### OpenTOSCA Functionality

A graphical modelling tool called WINERY is used to model application topologies and define nodes and relationships. First,the topology is created, the node and the relationship types are defined, with implementation artifacts realising the operations of these components. The artifacts and all other files are assigned into specific folders. Parameters and properties for the application are also defined. Thereafter, the build plans are modelled, using the Eclipse

BPEL Designer and stored in the Plan folder. In the final step, a Service Template is defined in XML. All these ingredients are now zipped into the archive, CSAR. Fig 2.15 depicts the creation of a CSAR file.

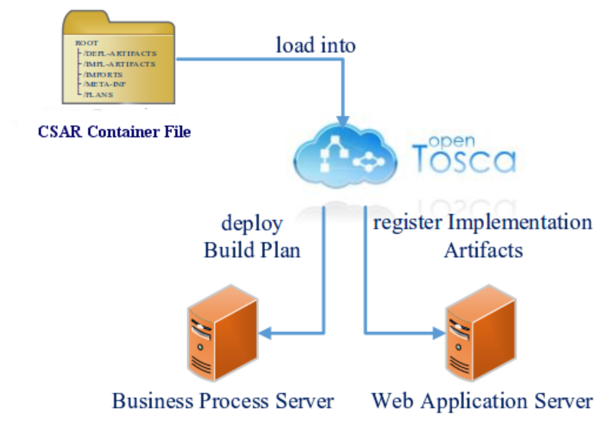


[a]

**Figure 2.15:** Creating a CSAR

[a]ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3428/DIP-3428.pdf

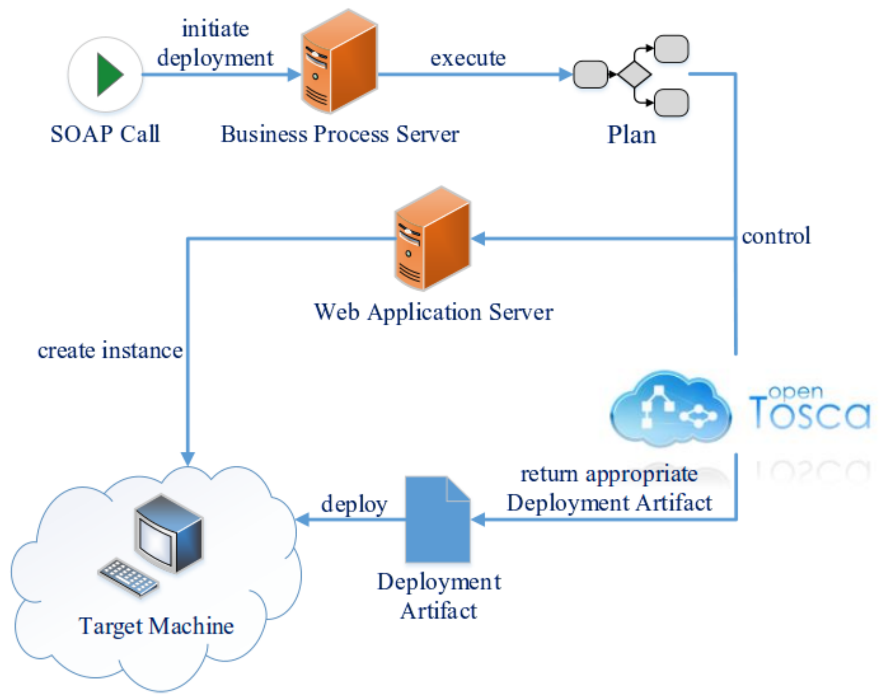
The CSAR file is loaded into OpenTOSCA via a web-interface. The archive is now unzipped to reveal its constituent files and the deployment of the application takes place in three steps. First, the constituent files are stored in the Files store, which is backed up by the local file system or Amazon S3. Next, the Control Component loads the TOSCA XML files, imports and resolves dependencies and references. It then invokes the Implementation Artifact Engine, whose suitable plugin deploys the Implementation Artifacts and makes the operations available by exposing their endpoints and storing them in the Endpoints Database. The Control Component now invokes the Plan Engine which performs the bindings and deploys the management plans. The build plan can be executed to instantiate the application. But prior to that, it might need outside data to function, such as login credentials or other input parameters. Once these data are provided, the build plan can be called either via the UI of the Self-Service Portal or on receipt of a SOAP message. Figures 2.16 and 2.17 give a graphical representation of the steps mentioned above[Liu13].



[a]

**Figure 2.16:** Deploying a CSAR in the OpenTOSCA environment

[a]ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3428/DIP-3428.pdf



[a]

**Figure 2.17:** Deploying application

[a]ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3428/DIP-3428.pdf

# Chapter 3

# Related Work

Cloud computing introduced a paradigm shift in technology a decade ago. It promised lower operating costs and particularly, offered to relieve the burden off the shoulders of enterprise staff. The debate over whether to adopt it or not was settled quite a while ago but some of the issues that came in its wake are still in need of solutions.

While the cloud helped to deploy hundreds and thousands of resources, managing distributed system resources across multiple data centers did not prove to be simple at all. An update to existing software meant applying updates at several hundred machines on which the software was in use. What would earlier take minutes to download an update or apply a security patch in a few machines, now required time multiplied by a factor of hundreds resulting in tasks that could run days and weeks, perhaps even months[1].

Well in time, automation came to the rescue of system administrators. The concept of automation was born in the Manufacturing sector long back when robots took over the long processes involved, reducing human effort considerably. Ever since, it has come to be applied in Healthcare, Security, Transportation, Agriculture and many other areas [aut12]. The agents of automation have however, changed from robots to computers. Computers have not only drastically cut down human labour, they have made processes faster and more accurate. Human involvement has been reduced to merely monitoring and controlling them. When it comes to large enterprise softwares, automation seems to be the only way out to manage processes. With several modules to install, configure and run, the possibility of error is high when relying solely on human efforts, not to mention the wasted time and tedium. Automation not only makes this work easy, it also generates fast, efficient results. Particularly for cloud deployment, automation is necessarily mandated. The efforts to automate processes to handle scalability have resulted in several configuration management tools [Lue11], noted amongst them being CFEngine, Puppet, Ansible and Chef. We examine a few of the automation techniques that have been used so far.

[1]http://www.infoworld.com/d/data-center/puppet-or-chef-the-configuration-management-dilemma-215279

## Scripts

Scripting evolved as a way of carrying out repetitive and deterministic tasks, which significantly reduced human effort and saved time. Scripts are nothing but a structured representation of commands that, when interpreted by a specific scripting engine or a certain program, are capable of executing tasks. Human interaction is not required. In time, scripts became the new robot system admins.

## Puppet

Puppet is a configuration management tool written in Ruby for IT automation. It is available in both enterprise and open source forms. Using declarative language or Ruby-DSL, a user can specify system resources and their state, which are then stored in files called 'Puppet manifests'. Afterwards, these Puppet manifests are read by a utility called Facter to retrieve system information and compiled by Puppet into a system-specific catalog containing resources and information about their inter-dependencies. This catalog is later applied against the target systems for automation and all actions undertaken by the tool are reported. So Puppet not helps in critical deployment of resources, it also helps in their management and reporting. System administrators can manage infrastructure throughout its lifecycle, from provisioning instances/servers, configuring them for the system, maintaining and updating them with necessary changes, whether on-premise or in the cloud.

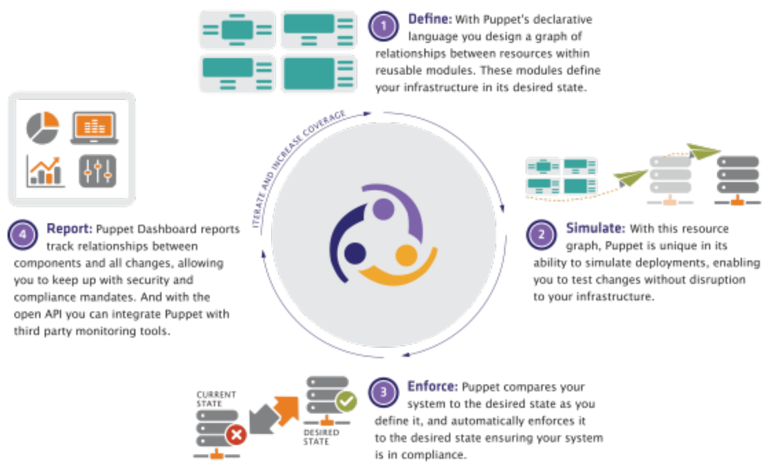
Puppet is generally used in a client/server formation. All clients contact one or more servers periodically to download the latest configuration. Then a check is performed to see if the existing configuration is the same as the one downloaded. The client can send back a report in case its configuration is out of sync.

Puppet offers a model-driven approach to IT automation. A user can define the desired state of the infrastructure's configuration using one of the many pre-built,freely downloadable configuration modules available in Puppet Forge, which is Puppet Labs' online marketplace. Otherwise, the user can build a customised module to suit his needs. Once built, these modules can be reused to run across different physical, virtual and cloud environments. These modules can be combined to form an application stack as shown in figure 3.1. [pup13]

Puppet is a widely popular tool and it is used by the Wikimedia Foundation,ARIN, Reddit,Dell, Rackspace, Zynga, Twitter, the New York Stock Exchange, PayPal, Disney, Citrix Systems, Oracle, Yandex, the University of North Texas, the Los Alamos National Laboratory, Stanford University, Lexmark and Google, among others [2].

http://www.aosabook.org/en/puppet.html

[2]http://en.wikipedia.org/wiki/Puppet_software



1 **Define:** With Puppet's declarative language you design a graph of relationships between resources within reusable modules. These modules define your infrastructure in its desired state.

2 **Simulate:** With this resource graph, Puppet is unique in its ability to simulate deployments, enabling you to test changes without disruption to your infrastructure.

3 **Enforce:** Puppet compares your system to the desired state as you define it, and automatically enforces it to the desired state ensuring your system is in compliance.

4 **Report:** Puppet Dashboard reports track relationships between components and all changes, allowing you to keep up with security and compliance mandates. And with the open API you can integrate Puppet with third party monitoring tools.

[a]

**Figure 3.1:** How Puppet works

[a]http://puppetlabs.com/puppet/what-is-puppet

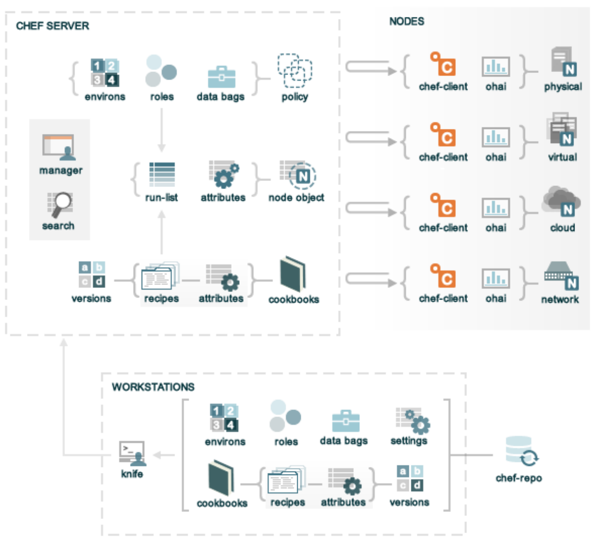
## Chef

Chef is a configuration management tool that can be used to deploy servers and applications on any scale of infrastructure-physical, virtual or cloud. Chef is available in both open-source and enterprise versions. An organisation using Chef must have one server, one or more workstations and the nodes that it needs to be configured. The requirements and conditions for configuration are written in Ruby language in the form of abstract definitions called cookbooks and recipes and these are stored at the server for all nodes. Their definitions describe the content of infrastructure-different resources, and how they should be deployed and managed. The chef-client on each node, queries the server for this information pertinent to its own node and accordingly applies the definitions on its local node. Thus automation of tasks is achieved in individual nodes. This results in a fully automated infrastructure which welcomes the provisioning of new nodes requiring only that their recipes be known [3].

The three main elements of Chef are the server, atleast one workstation and nodes. The server is a central hub that communicates with every node, ensures that every new node is registered and the right cookbooks and policies are applied. A workstation is the location where policies, cookbooks are defined or created and from where data is uploaded to the server. A node is a physical, virtual or cloud machine that is configured and run by a local chef-client. The figure 3.2 depicts these elements and the relationships that exist amongst them.

Noteworthy amongst the customers of Chef are Facebook, LAN, Wharton School and a host of other Universities and companies.

[3]http://docs.opscode.com/chef_overview.html



[a]

**Figure 3.2:** Chef

[a]http://docs.opscode.com/chef_overview.html

## IBM Workload Deployer 3.1

IBM Workload Deployer is a hardware appliance, packaged with software that offers both IaaS and Platform as a Service (PaaS) in a private cloud. Using the concept of patterns to describe the logical and physical configuration of virtual assets of a desired system, the deployer provisions such a system, building one integral component at a time. These patterns are nothing but templates that can be customized and they are of two kinds-virtual system patterns and virtual application patterns. A virtual system pattern is generally an operating system with an IBM software suite such as WebSphere Application Server. A virtual application pattern on the other hand, is a workload template, that enables the user to set up and manage his application environment (such as, IBM Workload Deployer Pattern for DB2 Workgroup Edition), simply by declaring the components, relationships, and policies that define it. The use of a workload template is a higher level of abstraction than a topology pattern, because it allows one to focus on the application, more than the environment [DLH11].

## SmartFrog

SmartFrog stands for Smart Framework for Object Groups, and it was developed in HP Labs as part of a research project. It is a java-based software framework for configuring, deploying and managing a system that is broken into components and distributed over several network hosts. In a distributed system, there are various components running in different environments that communicate with one another to keep the system running as a whole. It is essential that these components are installed and configured correctly for smooth functioning of the system. SmartFrog provides a standard language for defining configurations and system modelling, a runtime to deploy and manage running sytems, and its own set of libraries that implement its functionality. It enables automatic installation of systems and provides orchestration facilities to start and stop systems in the sequence desired. Failure detection and recovery are also supported.

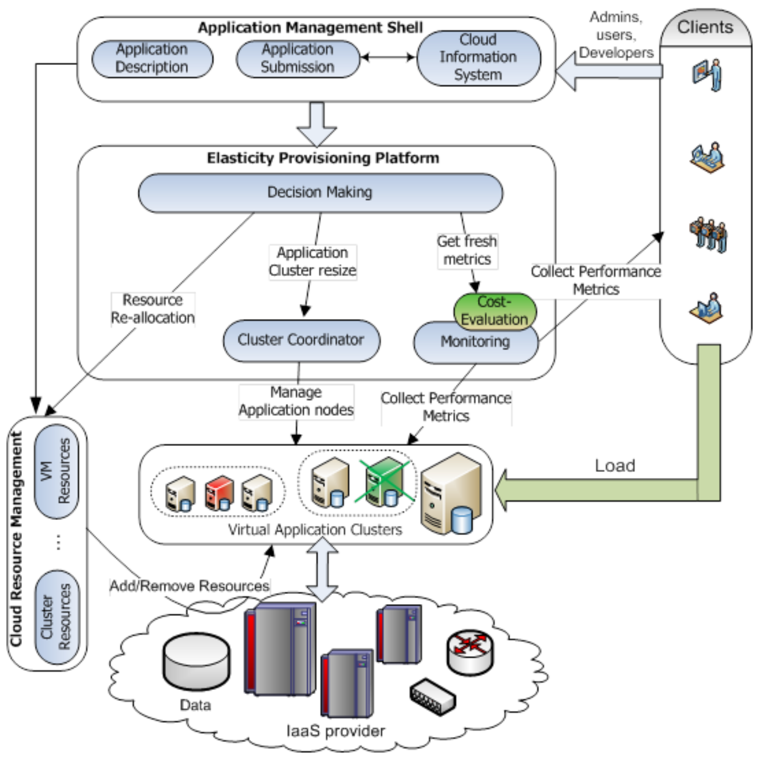
There are three main parts of the SmartFrog architecture. The first is the configuration language which provides powerful features to describe configuration data of the components. It also allows to define templates for higher level abstraction for complex configurations. The second part is a specification of how SmartFrog components fit into the framework. The component model provides features to describe a component's lifecycle as well as track its state at a particular time. Components are the entities that interpret the configuration details to carry out management and other operations. Apart from some standard components, there exists the possibility of adding new ones or extending functionality of the old components. The third part of the SmartFrog framework is the runtime. It interprets system descriptions to control the creation of components and enable the correct deployment of the tasks they embody. It is fully distributed in nature, with no central point of control, so any component can run it to manage its tasks and sub-tasks. The runtime also facilitates inter-component communication.

Although primarily open-source software, SmartFrog is also available in commercial versions [sma].

## Celar

Celar is a collaborative research project involving several European Universities. It is aimed at providing an automatic, multi-grained resource allocation for cloud applications which simply put, translates to allocating only the required amount of resources depending on the demand, thus cutting down both administrative costs and complexity. While scalability is a mandatory requirement of cloud computing, automatic scaling is not always achievable optimally. As the requirement for network, compute storage resources increases, over- or under-provisioning of resources poses a significant problem. Celar provides a solution to this by enabling an intelligent decision-making process that is based on two metrics: a) Making a cost analysis of collected cloud and application metrics by using a sort of scalability meter and presenting it to the user b) Modelling elastic properties of the application and measuring them both qualitatively and quantitatively.

Fig 3.3 depicts the three different layers covered by CELAR. In the bottom-most layer, the infrastructure is provided by two IaaS providers. The middle layer enables automatic elastic provisioning of resources on the cloud as well as their monitoring. The topmost layer is the one that provides a programming development environment to users, administrators and developers for describing applications, monitoring them etc. The outcome is deemed to be an open-source, modular software package that empowers an accurate allocation of resources,



[a]

**Figure 3.3:** Celar

[a]http://www.celarcloud.eu/

based on application demand and performance, thus optimising the utility of resources and reducing administrative overhead [cel12].

## Comparison and evaluation

### Scripts

Scripts hold the distinction of being the first agents of IT automation. The simplicity of their execution and the non-requirement of a framework, makes them the easiest automation tools. Further, their property of extensibility allows them to be used with other automation technologies such as BPEL, for executing steps in a workflow. The range of scripting languages has steadily grown from small and highly domain-specific languages to general purpose programming languages. Most languages are designed for specific environments, such as, bash, for Unix or Unix-like operating systems; JavaScript, for web browsers; and Visual Basic for Applications(VBA), for Microsoft applications[4].

[4]http://en.wikipedia.org/wiki/Script_computing

But while scripts have their utility in several automation tasks, in large-scale deployment systems however, they prove to be painfully inadequate. Particularly in a cloud environment, the complexities of role-based access privileges, the communication with various components etc. that are required to set up the infrastructure, are too much for scripts to handle. The simplicity of their usage fails them because, complicated workarounds with the aid of software libraries need to be implemented. Hence, scripts are best suited for an environment that has an infrastructure in place and merely requires deployment of applications. For other scenarios, scripts do not provide much utility.

### Puppet vs. Chef

Where data center orchestration and configuration management tools come into play, both Puppet and Chef take chief slices of the cake. Apart from enterprise offerings, both of them have open source versions that enable plug-ins to extend functionality for particular types of hardware and software. With the services they provide, the life of infrastucture admins has undeniably become simpler.

Puppet and Chef differ chiefly in their implementation and functional details. Puppet offers a Puppet-specific language, bundled with all of the installation requirements into packages. It is based on Ruby and even allows to write specifications in Ruby. Chef, on the other hand, lacks a Domain Specific Language and uses Pure Ruby, which lacks a model-driven approach, making it difficult to learn and use. The Web UI for Enterprise Chef does not provide some essential features that the Web UI for Enterprise Puppet does, for eg., Reporting. Chef is procedural in that it follows the sequence of instructions given. Puppet essentially requires one to list dependencies of packages which it takes care to install in the order required. Both Puppet and Chef have a client/server architecture that compiles instructions and keeps track of the actions performed by various machines(clients). Puppet uses data stores like PuppetDB and Heira to store these instructions while Chef stores them using a mixture of Postgres, Solr, and its own internal structure[5].

The decision to choose Puppet or Chef is hard to make and depends on individual preferences. But at the end of the day, neither of them proves to be of much help when required to provision or terminate an entire cloud infrastructure. The complexities are too many for either to handle. Nevertheless, the property of automated management of running VMs, offered by both configuration tools, expands the scope of integrating them into solutions like TOSCA [Liu13].

### IBM Workload Deployer 3.1

The IWD is the first hardware of its kind. It comes shipped with the Web Application Server Hypervisor Edition and provides access to WebSphere Application Server virtual images and

[5]http://www.infoworld.com/d/data-center/puppet-or-chef-the-configuration-management-dilemma-215279?page=0,1

patterns. To create and deploy an application environment using these images and patterns is simple and, if multiple systems need to be instantiated with the same requirements, the process is fast. Also, the ability to customize these patterns to one's need, makes it flexible for use. Infact, the very segregation of virtual system patterns and virtual application patterns allows users to create the environment after careful consideration of all the choices.

However, the IWD is an expensive technology when one needs to provision a cheap cloud service. When it comes to using an IaaS offering, the IWD loses out to OpenStack, which is open-source. The OpenTOSCA implementation, which again is open-source, also strives to provide all other XaaS offerings of the IWD to users [Liu13].

# Chapter 4

# Concept and Implementation

OpenTOSCA was used to deploy a virtual machine on the OpenStack cloud. This chapter begins with a briefing of the OpenStack Compute Service, the Compute API and the OpenTOSCA interface, explains the development of the virtual machine, and once deployed, tests and analyzes results.

## OpenStack and OpenTOSCA integration

Before OpenStack came to be used as the IaaS provider, Amazon EC2 API was used by OpenTOSCA, to deploy instances and applications. The EC2 API is quite powerful and the functionalities offered by it as a cloud provider are manifold. But AWS is a proprietary framework and the need for an open-source solution prompted the use of OpenStack, which offers many of the required functionalities and is fast expanding to incorporate new features.

The migration from AWS to OpenStack meant that all Node Types that were previously developed for the EC2 API would now have to be emulated on OpenStack. Obvious changes in URL composition, use of input parameters, composition of request messages would have to be taken care of. The first Node Type on OpenStack to be deployed by OpenTOSCA was chosen such as to create a virtual machine. Considering this as a first step, future Node Types and complicated workflows will be developed after successful deployment of this Type by a Build Plan.

Before venturing into the OpenStack API, the following section recapitulates a few key concepts related to the OpenStack Compute Service.

## OpenStack Compute - Concepts, Features and Requirements

Server is a virtual machine created in the compute system, after mandatorily specifying flavor and image.

*Reboot* refers to either a soft or a hard reboot. The former dispatches a restart call to the Operating System which is followed by a graceful shutdown of all running processes while a hard reboot powers down the VM. In the latter case, the virtualization platform must ensure that the reboot has been accomplished successfully even if the concerned VM was in the halted stage.

*Rebuild* function enables an existing server to change its image, after removing old data. Server ID and IP addresses are however, retained.

*Resize* is a function that facilitates scaling up or scaling down a server, i.e, changing its flavor from the current one. The original server is retained for a period of 24 hours to allow rollback should the conversion not yield the desired results. Once, the resize has been confirmed, the original is removed from the system.

The OpenStack Compute service is responsible for allocating server space in the cloud. Compute servers come in a variety of flavors, operating systems, disk space, RAM etc. They can be provisioned in a matter of minutes. Compute provides an API to facilitate interactions with these servers.

The Compute service allows control to access cloud entities through different levels of privilege, classified as users and projects. To enable shared access, Compute assigns a project-user pair, a particular role that determines what services they are allowed to access. Whilst not running any virtualization software itself, it interacts with virtualization mechanisms of the host and exposes this functionality over an API. Compute controls the hypervisor via an API.

The organizational structure of Compute is made up of resource containers called projects or accounts. They are also called tenants. They consist of volumes, images, keys, users and an individual VLAN. Users can be assigned to one or more projects. While making an API request, the *tenant_id* or *project_id* needs to be specified, else Compute attempts to use the *UserId* for it.

There is a per-tenant quota, set to limit the following:

1) Number of volumes that may be launched

2) Number of CPUs that may be allocated

3) Number of floating and fixed IP addresses for any instance provisioned

A tenant can also limit user access to particular images only. All information about role-based access privileges are set in the policy.json file.

Images provide templates to create Virtual Machine file systems. Glance is responsible for the storage and management of these images. Any number of instances can be launched from a single image. That is because, everytime an instance is provisioned, a fresh copy of the image is created. It is this copy that gets modified or even destroyed. The base image remains untouched. Most providers offer predefined OS-images for the client to select from, but there also exist means to create custom images from cloud servers launched earlier. These are called snapshots and come in handy when a particular server configuration needs to be launched frequently.

Flavors provide virtual hardware templates to create Virtual Machines. A flavor embodies a particular memory size, disk space and priority for CPU time. A set of predefined flavors is made available by the cloud provider. The user must select from one of these. Additionally, the user can customise a flavor by deleting the existing one and creating a new one with the same name.

Every instance in a project is associated with a security rule. Rules are filters of IP-addresses that are defined by users while launching an instance. In case a security rule is not specified, the default rule gets applied.

As explained earlier, Compute provides two block storage services- ephemeral and persistent volumes. Ephemeral volumes are defined by the flavor of the instance and they are managed by Compute. Persistent volumes are defined by the users and can be attached to a single instance at a time, but can be detached and reattached to another. They are however, not managed by Compute but by the OpenStack Block Storage (Cinder).

The work in this thesis has been developed on the HAVANA release of OpenStack.

## OpenStack Compute API

The OpenStack Compute API is a RESTful HTTP service that utilizes all aspects of the HTTP protocol. It employs the same parameters of HTTP, such as methods, URIs, response types etc, as well as features like caching and persistent connections. HTTP response codes can be used for facilitating further communication between client and provider. Additionally, the API can be extended to allow conditional GET requests using ETags. All the API endpoints are HTTP web services with their respective interfaces and they handle functions such as authorization, authentication, basic control and commands [api].

### Features and Requirements

The OpenStack Compute API accepts request and response messages in both XML and JSON encoding. The request format needs to be specified in the Content-Type in the header and the response format is specified in the Accept header. The default format is JSON.

The API requests can be sent by using one of the following:

cURL- A command-line tool that sends and receives HTTP requests.

OpenStack command-line clients- Every OpenStack service offers a command-line client to enable communication with the respective service API.

REST clients- Both Mozilla and Chrome web-browsers offer graphical REST clients.

### Authentication

Every HTTP request to the Compute API undergoes an authentication check. Multiple authentication schemes such as OAuth, Basic Auth, Token- are supported and they are provider-specific. For my deployment, authentication is done via tokens. To request for a service, an authentication request needs to be first sent to the Identity Service (Keystone). The request is composed of a payload of credentials and the desired response is a token. Credentials are a combination of *username, password* and optionally the *tenant_id.* Once a *token* has been obtained, future requests to any OpenStack API can be made by using the *token*, rather than supplying *username* and *password.* The *token* value is set in the *X-Auth-Token* header field of an HTTP GET request. A *token* is valid for 24 hours after which a new one needs to be generated. [api]

### Images

The Compute Service API may define an endpoint that hosts an Image Service. Three functions are served by this endpoint-*List Images, Get Image Details, Delete Image.* To retrieve a list of the available images, an HTTP GET request, replete with user credentials and tenant_id or merely the token will suffice. A *200 OK* response will return the list of images, visible by the tenant. The images are identified by their IDs, names and URLs. To filter the list of images, one can specify parameters in the URL, such as ID, name, type etc. By specifying "detail" in the URL, it is possible to view detailed information of images. Also, details of one particular image can be listed by using the ID of the desired image as a query parameter. An image can also be deleted by making an HTTP DELETE request for a particular *image ID.*

### Flavors

As in the case of images, the Compute Service API also defines an endpoint to access available flavors for that tenant. All flavors can be listed simply by making an HTTP GET request and providing the necessary authentication via a token. The list of flavors includes their IDs, names and URLs. There are filters which can be used to narrow down the choice of flavors. Two of them are most helpful-*minDisk*, which specifies the minimun amount of disk storage in gigabytes, and *minRAM*, that specifies the minimun amount of RAM in megabytes. Another functionality supported by the API is detailing all information about the flavor, on specifying its ID.

### Security Groups

Security Groups are sets of rules to filter IP addresses, that are applied to an instance's networking. These groups are specifically set for a project and a default security group is provided to every project. The default group is applied to any instance of the project, if no other security group is defined. The rules in the default security group, ban all incoming traffic. It is possible to edit the default group as well as to add fresh filters. [1]

### Servers

Possibly one of the most useful functionalities that can be achieved via the Compute API is to create a server. But a pre-requisite for this action is to have an image ID/URL and a flavor ID/URL to specify as query parameters. These are obtained earlier with the help of the API operations mentioned in the previous sections. Now the creation of an entity is achieved only by an HTTP POST request. Once again, the *token* is provided for authentication in the header, but this time with new fields, *Content-Type* and *Content-Length*, filled with the appropriate values, to facilitate the POST request. A name for the server needs to be specified. Other parameters such as *metadata, networks* and *personality*, are optional. Once the POST request has been sent along with the required parameters, a new server is provisioned. The progress of this provisioning depends on factors such as location of the image, network i/o and chosen flavor amongst others. The progress can be tracked on a scale of 0 to 100 %, by sending a GET request to the API endpoint alongwith the ID of the server. Once the server is created successfully, a *200 OK* response is sent along with the *server ID* and *URL.* [2]

Apart from creating a server, a list of available servers can also be queried using an HTTP GET. The list contains the names, IDs and URLs of all servers of the tenant. The list can be filtered by ID, server name or even image ID/URL and flavor ID/URL. It is also possible to update some attributes of a server. This is achieved by sending an HTTP PUT request with a *token* and the ID of the concerned server along with desired values of those attributes. Similarly, a DELETE request specified with a *server ID* will delete the server.

### Server Attributes

A few concepts related to server operations are worth mentioning.

1)Server Metadata- These are key-value pairs containing data about the server, such as server *name, tenant_id* etc.

2)Server Networks- These are the networks that the server connects to at launch time. The user can specify more than one network and also, a specific port or a fixed IP address in that network.

3)Server Personality- This is meant for server customization by injecting data into the file system. One can insert ssh keys, store configuration files and other data that one wants to access inside the server instance. For heavier customization, one would need to create a custom image.

[1]http://docs.openstack.org/trunk/openstack-ops/content/security_groups.html
[2]http://docs.openstack.org/api/openstack-compute/2/content/CreateServers.html

### Keypairs

Keypairs are the means provided by OpenStack for ssh access into virtual machine instances. They are composed of public and private keys and the Compute API facilitates their creation for a tenant. To create a keypair, one simply needs to send an HTTP POST request, specifying token in the header and a keypair name in the body. Multiple keypairs can also be created for a tenant. Again, a single keypair can be associated with multiple instances of a single tenant. In case of keypairs, names need to be unique. While creating a new keypair for a tenant, if another keypair by the same name exists, then the creation fails. It is also possible to import keypairs created by an external tool. A *200 OK* response will return the keypair name, public key string as well as the fingerprint.

Other functionalities offered by the API are enlisting existing keypairs, deleting a keypair and showing detailed information about a keypair that consists of the *public RSA key* and the *fingerprint*. They are accomplished by the standard HTTP operations.

### Fixed Ips

Every virtual machine, on booting, is assigned an IP address. This address remains associated with the instance until the latter is terminated. Rebooting does not cause the IP to change. This fixed IP gets allocated by the nova-network component. If the VM gets terminated accidentally and is later restored from its snapshot, there is no way to retain the old fixed IP. [3]

By using an HTTP GET request, it is possible to show detailed information about a specific fixed IP. To reserve or release a specific fixed IP, the appropriate action is called using an HTTP POST request.

### Floating Ips

By default, server instances have a default IP assigned to them. But it is available only internally. A floating IP needs to be assigned for outside access. The Compute API provides several functionalities to access and use floating IPs. The IP is created in the network where the server is launched. The API selects the first available floating IP from a pool of IPs and associates it with the instance. If an instance dies, its floating IP is available for reuse.

An HTTP POST request can be used to create a floating IP, allocate it to an instance or remove it from the instance. The IP can also be completely deallocated from the pool by using an HTTP DELETE request. In addition to that, all available floating IPs in the network pool can be listed using a GET request.

[3]http://www.mirantis.com/blog/configuring-floating-ip-addresses-networking-openstack-public-private-clouds/

## Integration with OpenTOSCA

As explained earlier, OpenTOSCA exposes management services of a node/component via an Implementation Artifact. The IA invokes a webservice whose methods handle the HTTP connection requests to the OpenStack API. One might argue that the Plan could directly call the API instead of deploying an IA. However, the number and the complexity of the connection requests, call for a wrapper object that can not only access different service endpoints of the API but also allow a flexible definition of operations' layout in the form of interfaces. Apart from a standard lifecycle interface, OpenTOSCA makes it possible to define more than one custom interface to carry out requisite operations. In the following sections, we take a look at how this development and integration have been achieved.

The development was done iteratively. Eclipse was chosen as the editor for building the web service to be used for IA. The web service was created in Java and it made use of JAVAX-WS libraries. The Implementation Artifact called PutUpServerLifecycle in this thesis, employs this web service that defines different endpoints. Each of these endpoints exposes an operation. We examine below the different functionalities offered by the endpoints.

### Development of requisite functions

The PutUpServerLifecycle webservice calls several web methods to perform different web operations. Since the Compute API is REST-based, so the standard HTTP calls can be serviced. In each of these methods are functions that make HTTP requests to separate OpenStack API endpoints to access different services.

The idea behind the working of the webservice is based on that of the REST client provided by Mozilla. While defining the Java operations to access the API, the functionality was first experimented with the REST client. Different endpoints were accessed, header values, request and response messages were initially tested on the client. Positive results of these invocations paved the way towards defining the functions of the web-service.

The format for the request and the response messages in this work, was specified to be JSON. Every user is assigned a *username* and *password* to access the OpenStack API, either programmatically or to login via the Dashboard. The admin assigns these credentials along with the tenant/project that the user will access. To make use of any services that the API provides, the user first sends an HTTP request with the login credentials in a JSON message body to the Identity Service. On receipt of correct credentials, a *token* is generated and returned in a JSON response message that is packaged with other details like timestamp of creation, time of expiry etc. In case the *tenant_id* is not known and needs to be obtained, the Identity Service has an endpoint for tenants. An HTTP GET request to this endpoint, with the token in its header, returns the correct *tenant_id.* The authentication is handled in a separate function, *getToken()*, that is called by other methods, and it returns both *tenant_id* and *token* as output parameters.

With this *token* value in *X-Auth-Token* field of the header, all further communication with the API is authenticated. Also, quotas of hardware resources are set for each user in order to control consumption across projects.

Since the aim of the development is automated deployment of a virtual machine, all operations leading to the creation of the server are first executed. A server requires keypairs for ssh access, an image to boot up its OS and file system, a flavor to define its dimensions. The image and the flavor are mandatory ingredients to create the server. Separate functions deal with these requirements.

The function *getFlavorBySize()*, specifies input parameters to define the minimum disk size in GB and the minimum RAM/memory in MB. These values are added as query parameters in the request and the result is the URL of a flavor that meets the minimum specifications. An alternate possibility will be to not specify the minimum requirements, but to select a random flavor URL and use it to build the server.

Two methods to obtain the image URL have been explored. One accepts the URL as an input parameter while another asks for keywords to look up the required image. In the first case, the desired URL is provided by the user or by the Plan. In the second case, a pattern-matching routine iterates through the collection of images stored at the image access endpoint of the API, trying to find the perfect fit based on Operating System name and version. Yet another option could be to randomly select an image URL from the list of URLs, and supply it for access.

Keypairs granting access to an instance are enabled for each user. To create a server, an existing *keypair* can be selected merely by specifying the name. If no *keypair* exists, a new *keypair* can be created and later provided for the server creation.

The *createServer()* function consumes the *flavor URL, image URL and keypair* name as input parameters. An HTTP POST request is sent to the API endpoint for servers, with a payload of the aforementioned parameters in the request body and with the *token* specified as a value for the *X-Auth-Token* header field. A successful JSON response message contains the *server_id*, the *location URL, timestamp* of creation and other values like the *name, status, fixed IP* etc.

After server creation, the *getFreeFloatingIp()* function searches the network pool for a *floating IP* that is not yet associated with some instance. If such an IP does not exist, then a new one is created. The free *floating IP* is associated with the newly created server for outside access, by the *allocateFreeFloatingIP()* function.

### Specification of Lifecycle Interface

In OpenTOSCA, the components to be used as the basis for building cloud applications are defined as Node Types. For example, if one wants to offer a web server as a component, then it is necessary to define a web server Node Type. In this thesis, a PutUpServerLifecycle Node Type attempts to provide a virtual server for deployment on the OpenStack cloud. The Node Type is defined under a unique namespace, and has its properties and operations encapsulated by a lifecycle interface. The lifecycle interface defines five operations: *install* - to instantiate the Node Type component, *configure* - to achieve configuration of the instance after installation, *start* - to start the server, *stop* - to stop the server, and *uninstall* - to unprovision the virtual machine. Each of these operations might require input parameters and produce different output values.

The Implementation Artifact is nothing but the concrete executable of the operations of a Node Type. While implementing an interface, there could be variations in the input and output parameter values, depending on the requirements of the Node Type. For e.g., the *configure* operation of a DBMS Node Type will require altogether different parameters from those of the *configure* operation of a web server Node Type [tos13a]. In this work, the operations to instantiate the server, to start and stop it, are categorised into standard lifecycle methods. While doing so, the input and the output parameters for each operation have been specified, and care has been taken to ensure that the semantics of the lifecycle methods are preserved, i.e, the server is started only in the *start* method and not in the *stop* method.

A specification of the lifecycle interface implementation used in the Node Type in this thesis is outlined below.

1. ***install***

Input params: *username*, *password*, *minDisk*, *minRAM*, *imageUrl*, *keypair*, *name*

Output params: *token*, *serverId*, *static*

The aim of this web method is to instantiate a server. But necessary parameters need to be input before this can be achieved. So the *install* method makes a host of calls to the OpenStack API to gather these parameter values. These calls are encapsulated within function bodies to make dealing with separate service endpoints simpler. The first operation has to be necessarily a call to obtain the *token* for authentication purposes. Retrieving the *tenant_id* is also accomplished by sending a request to the Identity Service.

Once the *token* value is in hand, a request is sent to the flavor endpoint of the API, specifying the *minimum RAM* and *minimum disk* size desired for creation of the server. The matching *flavor URL* is returned and used later as an input parameter, together with a couple of predefined values–*image URL* and *keypair*. The use of the *keypair* value is not compulsorily mandated but it is encouraged to enable future SSH login into the instance.

With the input parameters in tow, a call to the server endpoint of the API creates an instance on OpenStack that goes from state BUILD to ACTIVE in a matter of seconds. This running instance has the desired hardware and file specifications and has an internal IP associated with it. The *install* operation returns the *server_id* and the value of *fixed IP* as output parameters.

2. ***configure***

Input params: *username*, *password*, *serverId*

Output params: *floatingIp*

The *configure* web method attempts to offer functionalities to add or change settings of the virtual server created in the *install* operation. Firstly however, a *token* for the purpose of

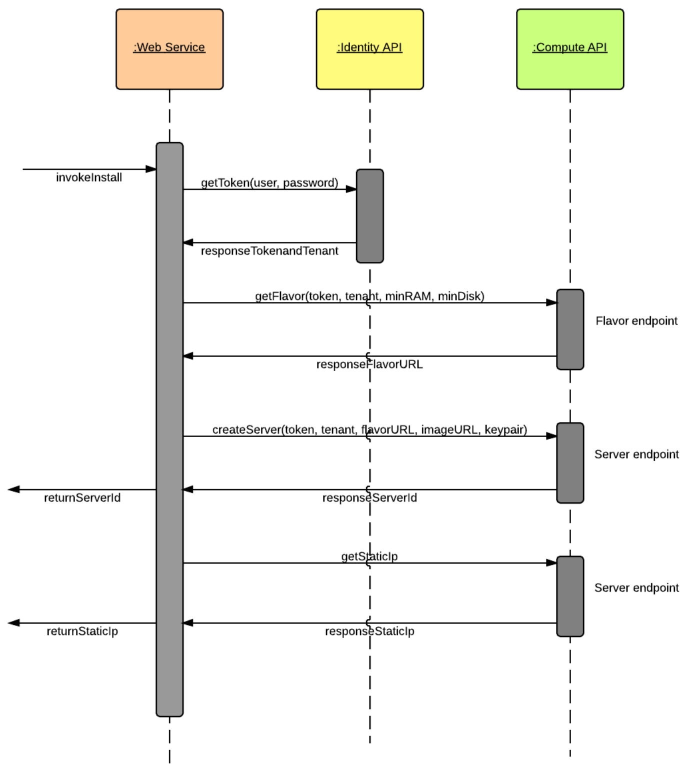


**Figure 4.1:** Sequence of install API calls

authentication needs to be obtained. One might question the redundancy of making the same call in this step when the *token* from the *install* step could be passed down. But in defence of defining the same function, the *token* obtained in the previous call will expire after 24 hours and will no longer be valid if the *configure* operation is invoked 24 hours later. Hence, to be on the safe side, the authentication call is made separately at the beginning of each lifecycle method.

In this thesis, the *configure* operation is used specifically to attach a *floating IP* to the newly created server. A check is first conducted in the pool, for a *floating IP* that is not yet associated with any instance. In case the check returns no IP, a new one is created by sending appropriate requests to the *floating IP* endpoint. After the *floating IP* is selected, another check is performed, this time to determine if the server state is set to 'ACTIVE'. As long as it is not, the server endpoint is pinged to check for the *status* value. Once it assumes the value 'ACTIVE', the *floating IP* is assigned. These multiple connection requests to the API are encapsulated in separate function calls residing inside the configure operation.

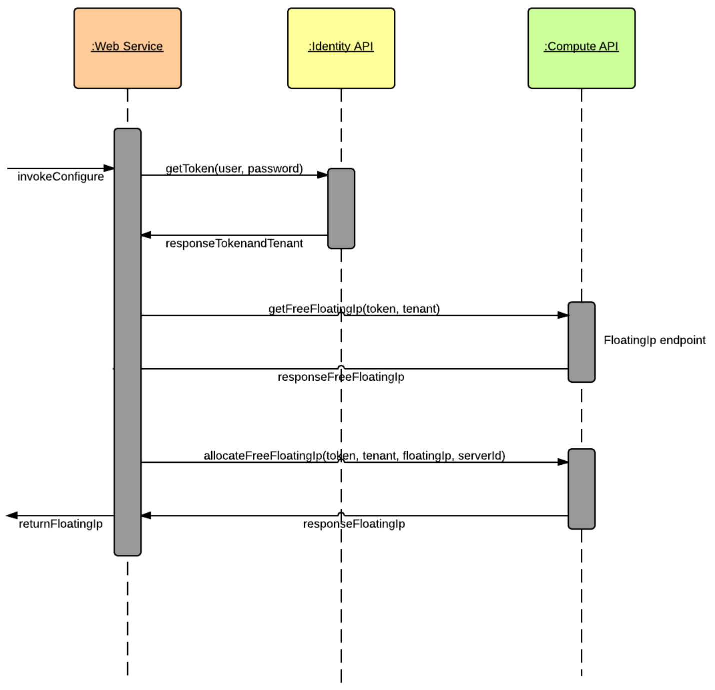


**Figure 4.2:** Sequence of configure API calls

3. ***start***

Input params: *serverId*, *username*, *password*

Output params: *status*

This method is simply meant to start the server. In our case, the server has already been started on instantiation in the install step. Hence, this method performs a necessary check of the *server state* and only if it is set to 'SUSPENDED', is a function invoked to resume running the server. It makes more sense to invoke the *start* method after the *stop* method has been invoked. For the same argument as explained in the previous step, a *token* is first generated and used for authentication. The *status* of the server is returned as an output value of the start method.

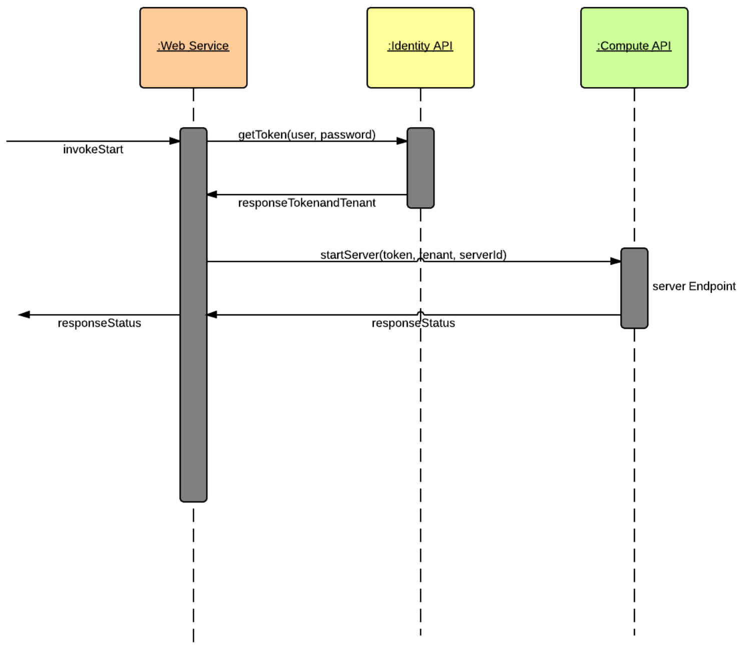


**Figure 4.3:** Sequence of start API calls

4. ***stop***

Input params: *serverId*, *username*, *password*

Output params: *status*

As the name suggests, this method suspends the running instance. An initial check is carried out to determine if the *server state* is 'ACTIVE'. If the check is affirmative, then an HTTP POST request to the server endpoint of the API suspends the instance. As usual, a *token* needs to be obtained before carrying out any functionality. The server *status* is returned as the final output parameter.

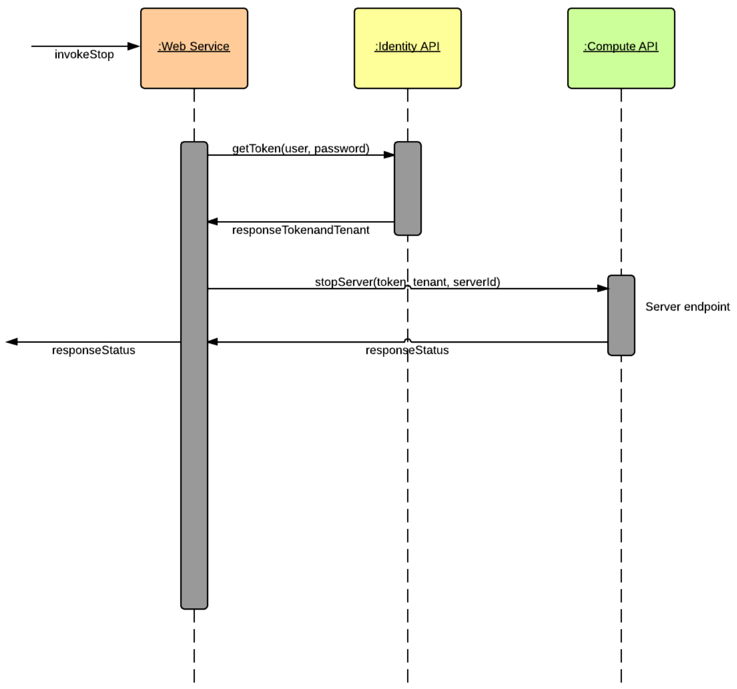


**Figure 4.4:** Sequence of stop API calls

5. ***uninstall***

This method is not currently used in our implementation, but it could be employed in future to deprovision the instance.

Fig 4.5 is a flowchart depicting the sequence of lifecycle methods. Fig 4.6 is a pictorial representation of the WSDL generated by the lifecycle web service.

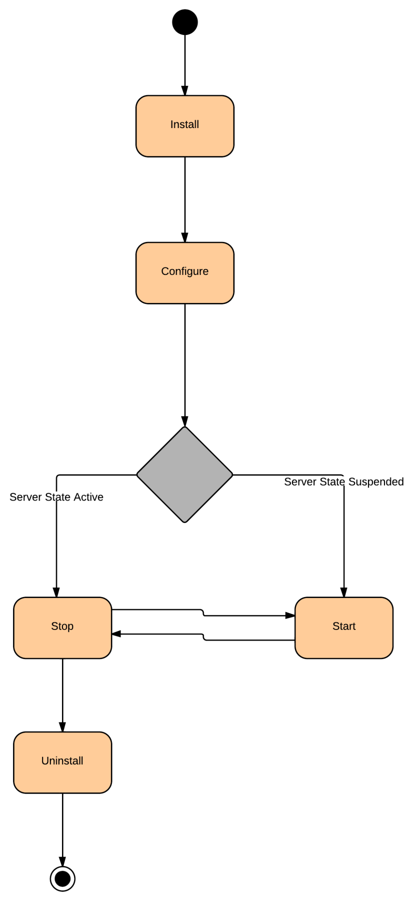


**Figure 4.5:** Sequence of lifecycle operations

## Specification of Alternate Interface

As can be seen, all necessary functions to facilitate server creation are well within the purview of the lifecycle interface. But while this meets our immediate need to deploy a server of a desired

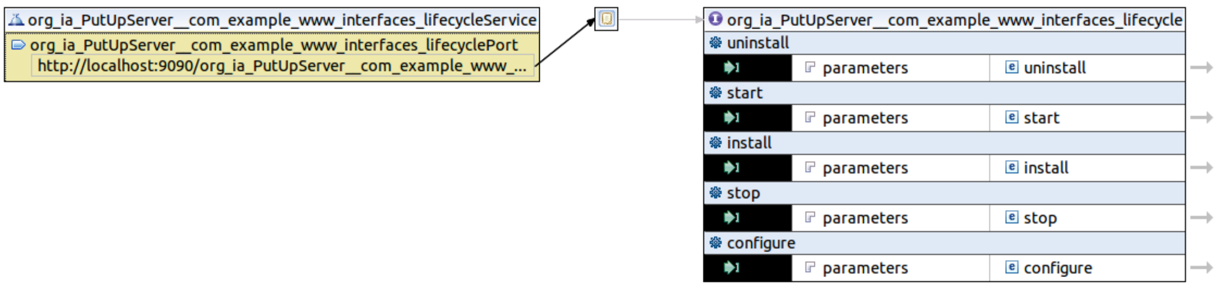

org_ia_PutUpServer__com_example_www_interfaces_lifecycleService
org_ia_PutUpServer__com_example_www_interfaces_lifecyclePort
http://localhost:9090/org_ia_PutUpServer__com_example_www_...
org_ia_PutUpServer__com_example_www_interfaces_lifecycle
uninstall
parameters
uninstall
start
parameters
start
install
parameters
install
stop
parameters
stop
configure
parameters
configure

**Figure 4.6:** WSDL of lifecycle

configuration, one has to admit that the interface provides very basic function definitions. If one wishes to extend functionalities beyond just creating a server with a predefined file system, the lifecycle interface might impose some constraints. Node Types may define more than one interface, that explore capabilities beyond those of the traditional lifecycle interface. For eg: A DBMS Node Type may define an interface that has operations to take backups, restore tables of the database [tos13a]. In this thesis, an alternate interface for the Node Type is proposed. Below are some of the function definitions.

1. Login

Input parameters:*user_id, password*

Output parameters: *token, tenant_id*

This does the job of generating a token apart from retrieving the *tenant_id*. A *token*, being the only means of authentication, makes this function primary. It is included as a necessary first step before making any API calls whatsoever.

2. GetFlavor

Input parameters: *token, tenant_id*

Output parameters: *flavor_URL*

To select a random flavor, one needs to just send a request to the flavor endpoint. Post-authentication, an arbitrary URL is returned as a parameter. To select a specific flavor, *minimum disk* and *minimum RAM* need to be mentioned in the request. The configuration closest to the desired specification is offered.

3. GetImageByName

Input parameters: *token, tenant_id, searchPattern*

Output parameters: *image_URL*

Given the list of available images, there could be two ways to select a URL. One would be to choose a random URL and return it as output, as in the case of flavor. But this gives no freedom of choice for the user. In order to choose an image, the user is enabled to specify some comma-separated keywords, say,*operating system name, version* etc. An iterative search routine looks for the image name which contains atleast one of these keywords. The first hit in the search is returned as output URL.

4. CreateSSHKeypair

Input parameters: *token, tenant_id, keypair_name*

Output parameters: *keypair*

A *keypair* is necessary to ssh into an instance. If while creating a VM, there exists no *keypair* for the tenant or if no *keypair* is specified, later, one cannot access the instance from outside. To create an sshkeypair, a name has to be input. If a *keypair* by the same name already exists, then conflict occurs. To avoid this, a combination of the *username* and a random number should be used for the *keypair* name. Once the *keypair* is successfully generated, it is specified as a parameter while creating a VM. If this interface is implemented, then every new instance created by a tenant will bear a unique *keypair* name.

5. CreateServer

Input parameters: *token, tenant_id, keypair, flavor_URL, image_URL*

Output parameters: *server_id*

As in the lifecycle interface, this method creates a virtual machine with all the input parameters obtained in the previous function calls. A server of desired configuration and file system is deployed on OpenStack, it bears its own unique keypair, but needs a floating IP for ssh access.

6. UpdateServer

Input parameters: *token, tenant_id, server_id*

Output parameters: None

This function can be used to update certain editable attributes of a server by sending an HTTP PUT request. Values of *server name, accessIPv4 and accessIPv6* can be updated.

7. AllocateFloatingIP

Input parameters: *token, tenant_id, server_id, floating_IP*

Output parameters: None

This is used in the same manner as in the lifecycle interface. A check is performed to see if the server status value is 'ACTIVE' and once it is set, a free floating IP is allocated. But prior to that, if a floating IP doesn't exist, a new one is created.

8. SetupSecurityGroup

Input parameters: *token, tenant_id, ip_protocol, from_port, to_port, cidr*

Output parameters: None

This can be categorised into separate function bodies for listing and creating rules in the default security groups. The *ip_protocol* could be TCP, UDP or ICMP. Each of them has its own port requirements. For TCP and UDP, the port range needs to be specified, in which case, the *from_port, to_port* fields will require values. Field *cidr* requires an IP address block. With the help of standard HTTP connection requests, these rules can be listed, viewed in detail and set. The default security group is associated with every instance unless otherwise specified. Instances with the same security group can communicate with one another.

9. SSHintoServer

Input parameters: *user_id, password, floating_IP, keypair*

Output parameters: None

This is used to SSH into the created server via a terminal, after specifying the username and the floating IP associated with the concerned server. Of utmost importance in this activity is to specify the values of the associated keypair and user password. Once the access to the server has been programmatically secured, various kinds of utility and application software could be installed for deriving maximum utility out of the virtual machine.

# Chapter 5

# Evaluation

In this chapter, a qualitative analysis of the Node Type that has been developed, is presented. First, the concept and the tools that went into creating the particular Node are assessed and then, some of the testing scenarios are presented and analyzed.

## Implementation Analysis

The need for an open-source cloud service prompted the development of an OpenStack Node Type. The first step taken in this direction has been to provision a virtual machine on OpenStack via a Build Plan processed by the OpenTOSCA container. As mentioned earlier, the processing inside the OpenTOSCA container is imperative, so the Build Plan formulates and executes the logic of how artifacts should be deployed or handled. As OpenTOSCA is still undergoing development, there is a certain rigidity in the structures of the artifacts and the execution of the Build Plan. A declarative approach to process the artifacts is yet to be defined.

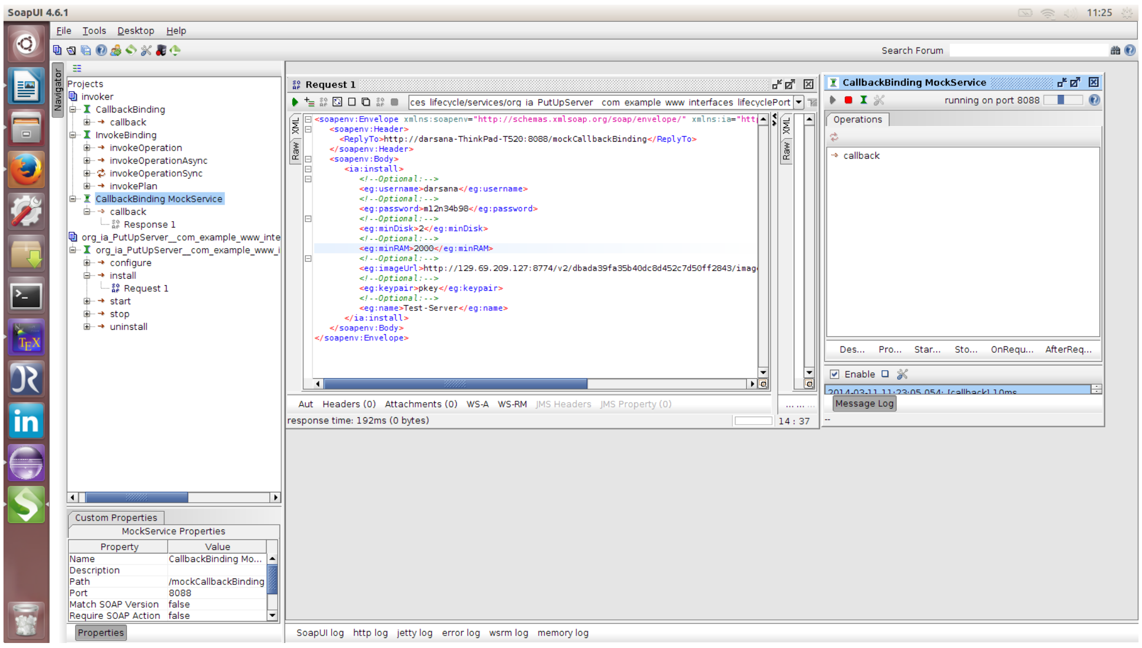
OpenTOSCA has been spawned by several projects and led to several other ones aimed at improving some aspect of the complete system. Notable amongst them are the CloudCycle, Vino4TOSCA and Valesca4TOSCA projects which are still in progress. Vino4TOSCA offers a visual notation of components, Valesca4TOSCA provides a graphical tool to design Types and Templates. The need for a graphical tool to describe components and their relationships and also model application topologies, was long felt. Respite came in the form of WINERY, a modelling tool for TOSCA projects. It provides a web-based environment to create and modify elements under the TOSCA specification. The components and artifacts created in this manner are stored in a repository that makes it convenient to import and export using CSAR format. WINERY allows to upload workflows and also provides a BPMN modeller to model topologies and Service Templates. The defined Types are distinguished by namespaces and multiple Node Types with the same name and definition can be deployed under different namespaces.

This thesis dealt with creating the Implementation Artifact and the tasks of defining the Node Type, packaging the IA into a CSAR and importing it were achieved with the help of WINERY. For the development of the IA, a local Tomcat server and a runtime provided by Apache CXF were used. For the testing of the endpoints, the free version of SOAP UI tool proved adequate. The testing consisted of sending calls with the required soap messages to the web service endpoints and analyzing the response messages.

The development of the Build Plan was undertaken separately in a student project and it was later integrated into OpenTOSCA.

## Results Analysis

Since, creating the Build Plan and deploying via the Plan was not part of this work, some kind of testing was carried out to check for the feasibility of invoking the web service via SOAP calls. Soap UI(free version) was the tool of choice for this purpose. The WAR file of the Implementation Artifact web service was deployed on tomcat 7 via the tomcat manager. The wsdl generated by the service was then used to create a SOAP project on SOAP UI. Necessary input parameters were set in the request messages and the web service was invoked. At another port on the local machine, a mock web service was deployed for the purpose of acting like a receiver service. On invocation of the first web service, the response was observed in the mock service - successful creation of the virtual server on OpenStack generated a response message consisting of all output parameters including *serverId* and *staticIp*, while failure was indicated by blank output parameters.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ia="htt
   <soapenv:Header>
     <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBinding</ReplyTo>
   </soapenv:Header>
   <soapenv:Body>
      <ia:install>
         <!--Optional:-->
         <eg:username>darsana</eg:username>
         <!--Optional:-->
         <eg:password>m12n34b98</eg:password>
         <!--Optional:-->
         <eg:minDisk>2</eg:minDisk>
         <!--Optional:-->
         <eg:minRAM>2000</eg:minRAM>
         <!--Optional:-->
         <eg:imageUrl>http://129.69.209.127:8774/v2/dbada39fa35b40dc8d452c7d50ff2843/imag
         <!--Optional:-->
         <eg:keypair>pkey</eg:keypair>
         <!--Optional:-->
         <eg:name>Test-Server</eg:name>
      </ia:install>
   </soapenv:Body>
</soapenv:Envelope>
```

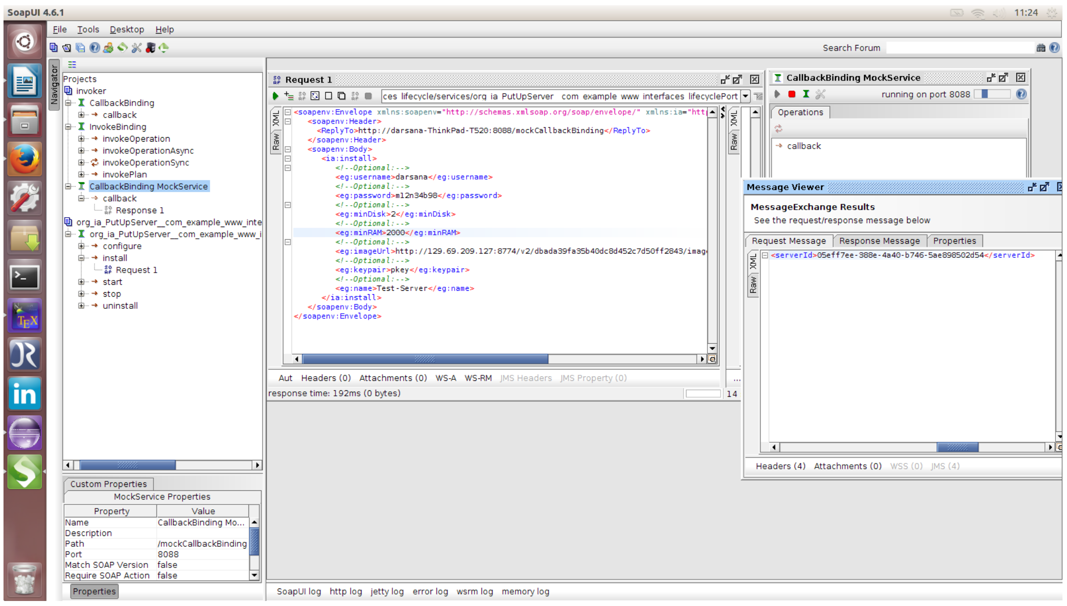
response time: 192ms (0 bytes)

| Property | Value |
|---|---|
| Name | CallbackBinding Mo... |
| Description | |
| Path | /mockCallbackBinding |
| Port | 8088 |
| Match SOAP Version | false |
| Require SOAP Action | false |

**Figure 5.1:** SOAP request for install web method

As can be seen in Fig 5.1, a SOAP request has been sent to the IA web service being hosted at port 8080 on localhost. Each of the web methods is invoked in separate requests. Here, the *install* method is being invoked. The request message contains a payload of parameters. The response is received by the CallbackBinding MockService hosted at port 8088 on localhost. Fig

5.2 shows the output response. The web method has performed successfully, so the response message contains the desired return parameters.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ia="htt
   <soapenv:Header>
     <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBinding</ReplyTo>
   </soapenv:Header>
   <soapenv:Body>
      <ia:install>
         <!--Optional:-->
         <eg:username>darsana</eg:username>
         <!--Optional:-->
         <eg:password>m12n34b98</eg:password>
         <!--Optional:-->
         <eg:minDisk>2</eg:minDisk>
         <!--Optional:-->
         <eg:minRAM>2000</eg:minRAM>
         <!--Optional:-->
         <eg:imageUrl>http://129.69.209.127:8774/v2/dbada39fa35b40dc8d452c7d50ff2843/imag
         <!--Optional:-->
         <eg:keypair>pkey</eg:keypair>
         <!--Optional:-->
         <eg:name>Test-Server</eg:name>
      </ia:install>
   </soapenv:Body>
</soapenv:Envelope>
```

response time: 192ms (0 bytes)

MessageExchange Results
See the request/response message below

```
<serverId>05eff7ee-388e-4a40-b746-5ae898502d54</serverId>
```

**Figure 5.2:** SOAP response for install web method

Once, the server has been created, the configure operation is invoked by another SOAP call. This method allocates a floating Ip to the server, so the *serverId* is passed along with the user credentials. The *floatingIp* value is returned in the response message on successful allocation as can be seen in Fig 5.3.

The *start* method aims to instantiate the server, set the server *status* to 'ACTIVE'. On invocation, if it succeeds in its task, the response message will contain the status 'Running'. Fig 5.4 depicts this operation.

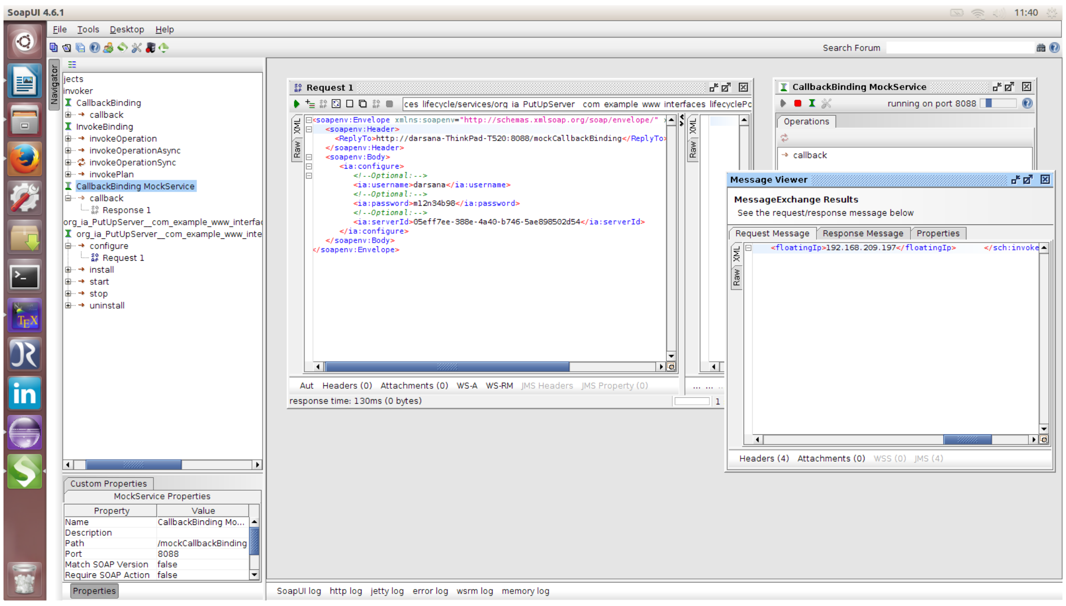
As in the *start* method, the *stop* method attempts to suspend a running server. On invocation and successful accomplishment, a *status* of 'Shutdown' should be seen in the response message, Fig 5.5.

If any of the required parameter values is not input, or say a parameter value is input wrongly, an error situation occurs. In Fig 5.6, a wrong value of *keypair* is specified in the *install* method resulting in no creation of the virtual machine, as indicated by the blank output values of *serverId* and *staticIp*.

Some of the results of the web service invocation using SOAP UI tool are presented in the figures that follow. Fig 5.7 shows a virtual server just being provisioned. The server has been created with desired specifications of size and file system and it is being scheduled to run.

Fig 5.8 shows the virtual machine with a floating IP assigned to it.

During the test using Soap UI, deployment was achieved in a matter of milliseconds but this was because all tests were conducted on the local host. Later by using the OpenTOSCA



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" x
  <soapenv:Header>
    <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBinding</ReplyTo>
  </soapenv:Header>
  <soapenv:Body>
    <ia:configure>
      <!--Optional:-->
      <ia:username>darsana</ia:username>
      <!--Optional:-->
      <ia:password>m12n34b98</ia:password>
      <!--Optional:-->
      <ia:serverId>05eff7ee-388e-4a40-b746-5ae898502d54</ia:serverId>
    </ia:configure>
  </soapenv:Body>
</soapenv:Envelope>
```

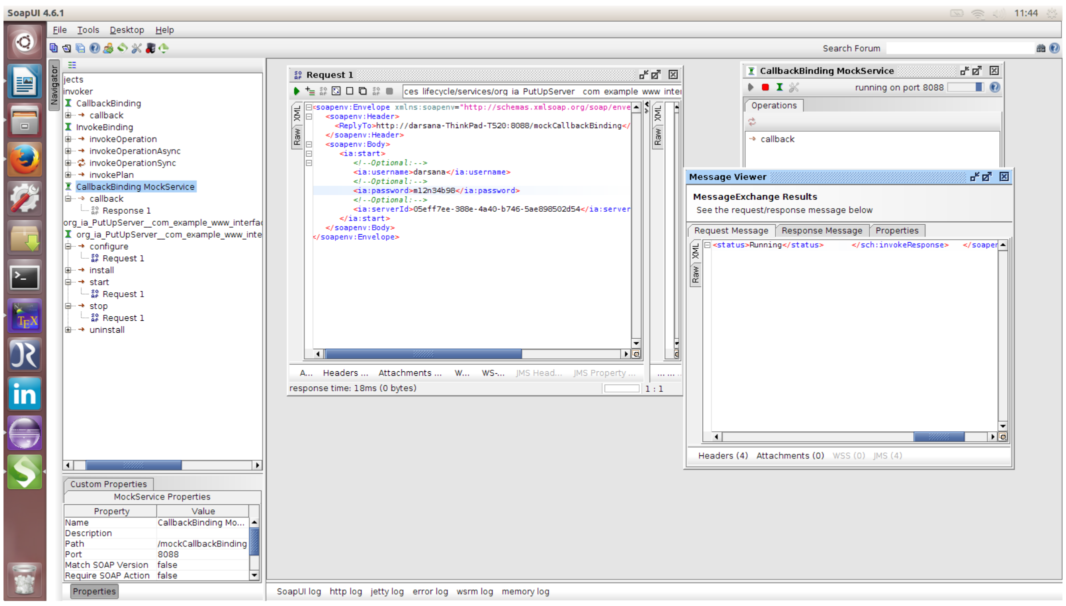
response time: 130ms (0 bytes)

Message Viewer — MessageExchange Results: See the request/response message below

```
<floatingIp>192.168.209.197</floatingIp>    </sch:invoke
```

| Property | Value |
|---|---|
| Name | CallbackBinding Mo... |
| Description | |
| Path | /mockCallbackBinding |
| Port | 8088 |
| Match SOAP Version | false |
| Require SOAP Action | false |

**Figure 5.3:** SOAP request and response for configure web method



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/enve
  <soapenv:Header>
    <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBinding</
  </soapenv:Header>
  <soapenv:Body>
    <ia:start>
      <!--Optional:-->
      <ia:username>darsana</ia:username>
      <!--Optional:-->
      <ia:password>m12n34b98</ia:password>
      <!--Optional:-->
      <ia:serverId>05eff7ee-388e-4a40-b746-5ae898502d54</ia:server
    </ia:start>
  </soapenv:Body>
</soapenv:Envelope>
```

response time: 18ms (0 bytes)

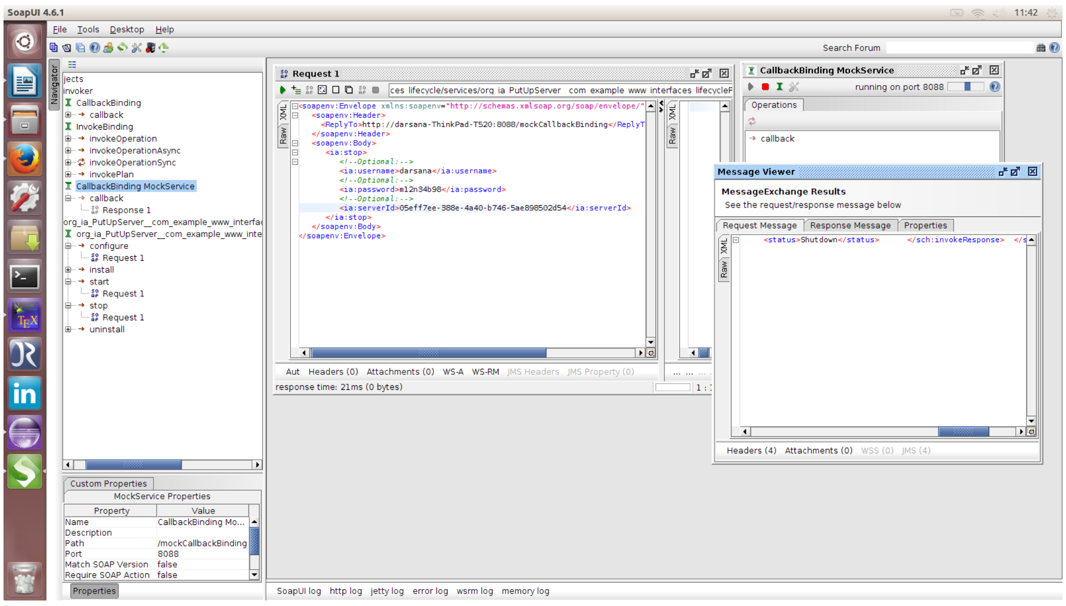
Message Viewer — MessageExchange Results: See the request/response message below

```
<status>Running</status>    </sch:invokeResponse>    </soaper
```

| Property | Value |
|---|---|
| Name | CallbackBinding Mo... |
| Description | |
| Path | /mockCallbackBinding |
| Port | 8088 |
| Match SOAP Version | false |
| Require SOAP Action | false |

**Figure 5.4:** SOAP request and response for start web method

container, the actual deployment of the Node Type by the Build Plan was also successfully achieved. It marked the success of automated deployment on OpenStack for the first time after migration from AWS. Also, the provisioning of the virtual server was achieved fairly fast as compared to previous performance of AWS. This paves the way for future Node Types to be deployed and executed on OpenStack.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  <soapenv:Header>
    <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBinding</ReplyT
  </soapenv:Header>
  <soapenv:Body>
     <ia:stop>
        <!--Optional:-->
        <ia:username>darsana</ia:username>
        <!--Optional:-->
        <ia:password>m12n34b98</ia:password>
        <!--Optional:-->
        <ia:serverId>05eff7ee-388e-4a40-b746-5ae898502d54</ia:serverId>
     </ia:stop>
  </soapenv:Body>
</soapenv:Envelope>
```

response time: 21ms (0 bytes)

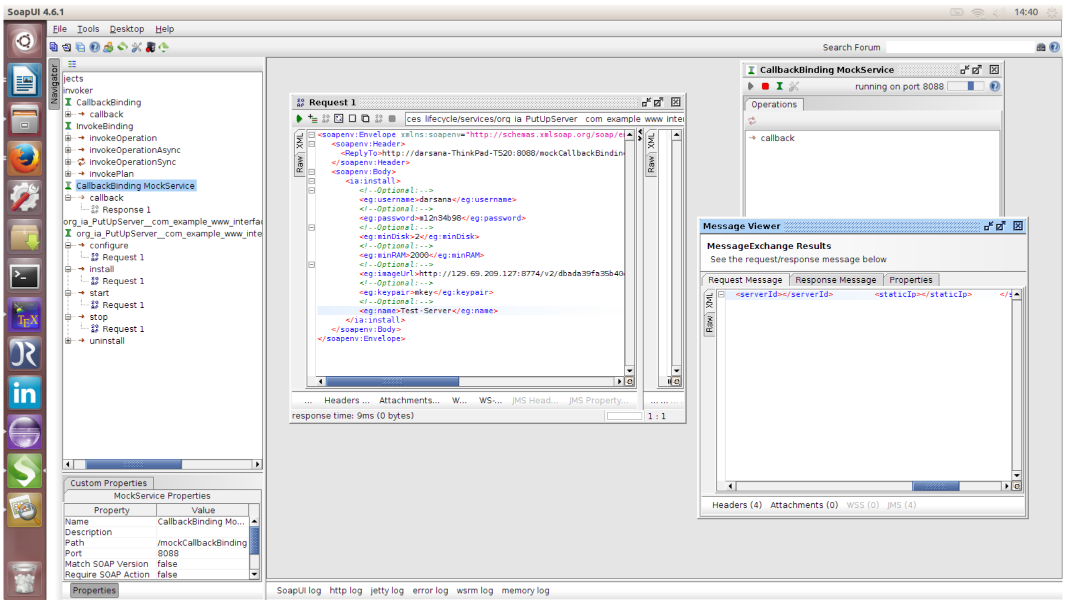
CallbackBinding MockService running on port 8088

MessageExchange Results
See the request/response message below

```
<status>Shutdown</status>      </sch:invokeResponse>  </s
```

| Property | Value |
|---|---|
| Name | CallbackBinding Mo... |
| Description | |
| Path | /mockCallbackBinding |
| Port | 8088 |
| Match SOAP Version | false |
| Require SOAP Action | false |

**Figure 5.5:** SOAP request and response for stop web method



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/e
  <soapenv:Header>
    <ReplyTo>http://darsana-ThinkPad-T520:8088/mockCallbackBindin
  </soapenv:Header>
  <soapenv:Body>
     <ia:install>
        <!--Optional:-->
        <eg:username>darsana</eg:username>
        <!--Optional:-->
        <eg:password>m12n34b98</eg:password>
        <!--Optional:-->
        <eg:minDisk>2</eg:minDisk>
        <!--Optional:-->
        <eg:minRAM>2000</eg:minRAM>
        <!--Optional:-->
        <eg:imageUrl>http://129.69.209.127:8774/v2/dbada39fa35b40
        <!--Optional:-->
        <eg:keypair>mkey</eg:keypair>
        <!--Optional:-->
        <eg:name>Test-Server</eg:name>
     </ia:install>
  </soapenv:Body>
</soapenv:Envelope>
```

response time: 9ms (0 bytes)

CallbackBinding MockService running on port 8088

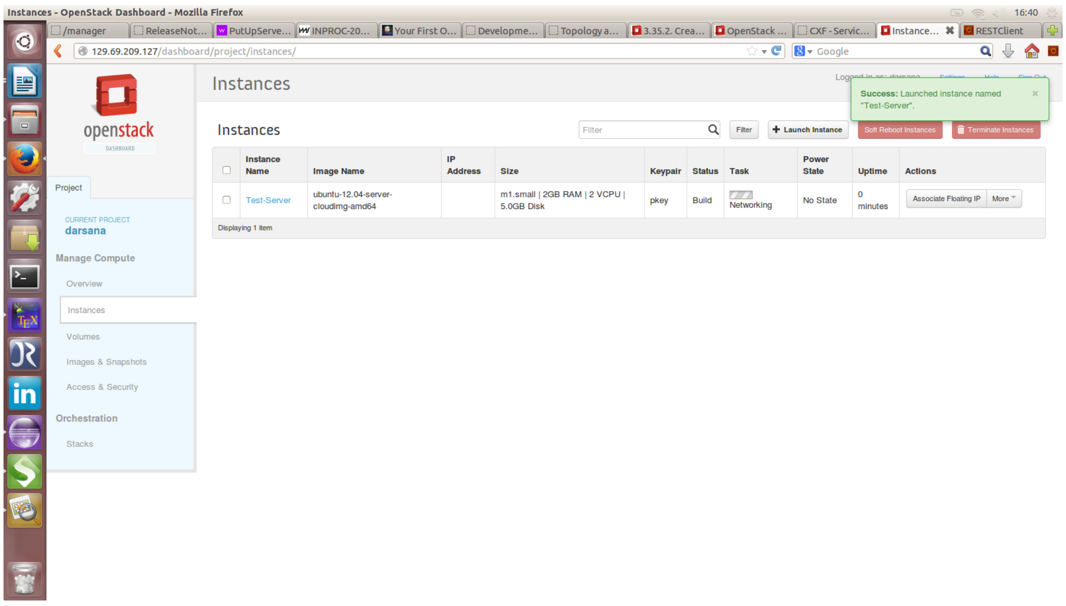
MessageExchange Results
See the request/response message below

```
<serverId></serverId>      <staticIp></staticIp>     </s
```

| Property | Value |
|---|---|
| Name | CallbackBinding Mo... |
| Description | |
| Path | /mockCallbackBinding |
| Port | 8088 |
| Match SOAP Version | false |
| Require SOAP Action | false |

**Figure 5.6:** Soap Call Error

Implementing the proposed interface other than the lifecycle, is also possible, albeit with a change in parameters which could possibly be supplied by the Plan. Service Templates are written in XML, they preserve the same structure and hence, modifications and extensions can be managed with mere copy-paste adaptations. Also, the manual effort of introducing changes in XML has been reduced to a huge extent by the WINERY tool - version 1.1, the use of which facilitates creation and modification of the templates more easily.



Instances

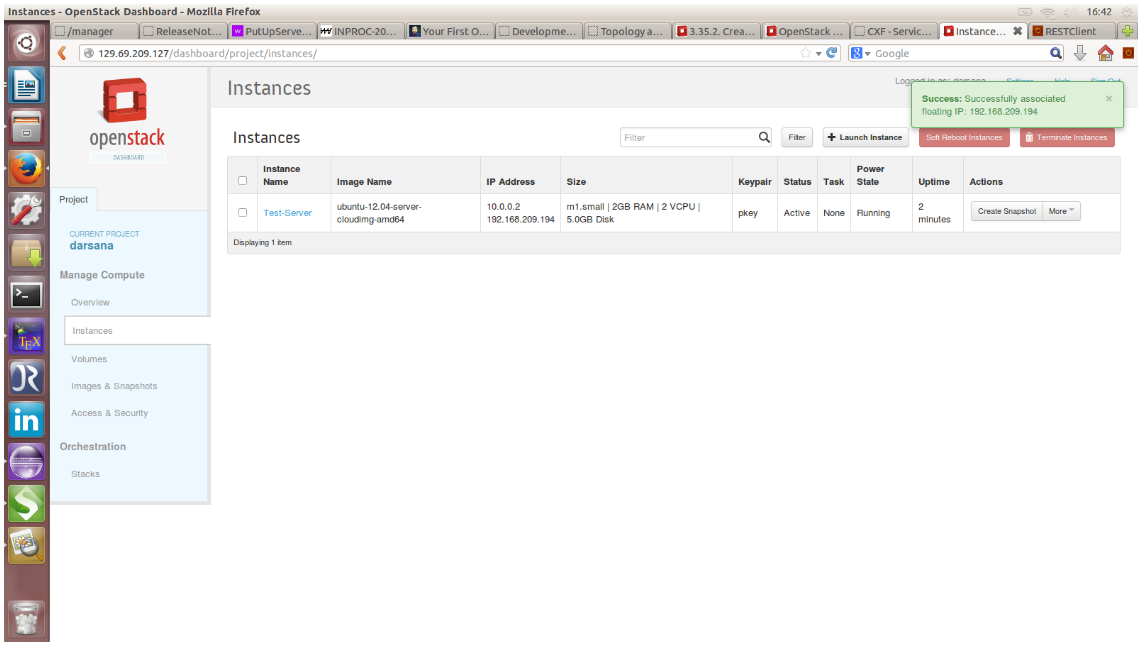
**Success:** Launched instance named "Test-Server".

Instances

| | Instance Name | Image Name | IP Address | Size | Keypair | Status | Task | Power State | Uptime | Actions |
|---|---|---|---|---|---|---|---|---|---|---|
| ☐ | Test-Server | ubuntu-12.04-server-cloudimg-amd64 | | m1.small \| 2GB RAM \| 2 VCPU \| 5.0GB Disk | pkey | Build | Networking | No State | 0 minutes | Associate Floating IP More |

Displaying 1 item

**Figure 5.7:** Create VM



Instances

**Success:** Successfully associated floating IP: 192.168.209.194

Instances

| | Instance Name | Image Name | IP Address | Size | Keypair | Status | Task | Power State | Uptime | Actions |
|---|---|---|---|---|---|---|---|---|---|---|
| ☐ | Test-Server | ubuntu-12.04-server-cloudimg-amd64 | 10.0.0.2 192.168.209.194 | m1.small \| 2GB RAM \| 2 VCPU \| 5.0GB Disk | pkey | Active | None | Running | 2 minutes | Create Snapshot More |

Displaying 1 item

**Figure 5.8:** Allocate Floating Ip

# Chapter 6

# Summary and Future Work

TOSCA provides a generic standard to describe how components interact with one another in a cloud environment. However, the lack of concrete directives leaves a lot of ambiguity in defining these components and their relationships.

OpenTOSCA is an open-source implementation of TOSCA working draft 5 (WD5) specification that was developed by the University of Stuttgart. Currently, the facilities it provides are limited to deployment of services and work is still going on to update features to enable management and termination.

A virtual server was required to be provisioned in an OpenStack environment by means of a Service Template. This was achieved with the help of a web service whose functionality is to create the server, packaged along with other requisites into a CSAR file that is processed by the OpenTOSCA container.

While designing the web service, different possibilities were considered for obtaining the parameters required to create a virtual server. A predefined lifecycle interface was chosen for implementation- the speciality of the interface being that, it provided methods to install, configure, start, stop and even terminate the server, thereby defining a proper flow to get the server up and running. Alternative interfaces may be used, some of which could facilitate adding more customizations to the virtual server, both pre- and post-creation.

Creation of a Build Plan and Service Template was not part of this thesis and it was undertaken separately by other students of the University. The Build Plan successfully invoked the web service, thus deploying a virtual server in OpenStack for the first time in an automated manner. The success of this endeavour gives a green signal to the plan of migration of cloud services from AWS to OpenStack. More complicated cloud services can be undertaken to be deployed in the future.

## Future Work

While the successful provisioning of the virtual machine via OpenTOSCA certainly promises greener pastures in automated deployment, it still leaves a lot to be desired. For one, the container can currently process a single Build Plan. It is possible to have more than one Build Plan in the archive but at the moment, only one can be handled. The idea of management plans is still a theoretical concept and work is being done to ensure that both management and termination plans shall be processed by the container in future. Creation of the Build Plan is another concern. As of now, this is done with some aid of the BPMN plugin in eclipse, but a major part of workflow design is still done by hand. As the complexity of the Plan grows, this will become more and more difficult. Tooling support can fork in two directions-one way could be to generate the Plan from the Application Topology, the other could be to have some sort of graphical designer for defining the workflow. In the former case, there need not be any provision to upload a Plan into the archive, but it should be possible to simply derive the workflow from the defined Topology, thus making automation more concrete. In the latter case, a graphical modelling tool, possibly integrated in WINERY, will reduce manual effort in creating the Plan.

The capabilities of OpenStack as a cloud service provider, too, need to be explored in greater detail. Discussions are presently on to define a host of standard Node Types for processing in OpenTOSCA. Standard Node Types could necessitate the use of compatible Relationship Types, so any Node Type that is inherited or extended from a standard Type would have to interact/communicate with another via a predefined set of Relationship Types. This would not only standardize the handling of different artifacts but would also allow more ease in developing future Node Types with predefined interfaces while also keeping the possiblity for defining custom interfaces, open.

# Bibliography

[api] OpenStack API Quick Start. URL http://docs.openstack.org/api/quick-start/content/. (Cited on pages 37 and 38)

[aut12] What Is Automation? 2012. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6104197. (Cited on page 27)

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*, LNCS. Springer, 2013. (Cited on pages 22 and 24)

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, chapter TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer, New York, 2014. doi:10.1007/978-1-4614-7535-4_22. (Cited on pages 16 and 19)

[cel12] Automatic, Multi-Grained elasticity-provisioning for the cloud. 2012. URL http://www.celarcloud.eu/. (Cited on page 32)

[DLH11] M. M. P. P. S. M. C. S. Deni Lukmanul Hakim, Alexander Hay. *Virtualization with IBM Workload Deployer: Designing and Deploying Virtual Systems*, chapter Abstract. 2011. URL http://www.redbooks.ibm.com/abstracts/sg247967.html?Open. (Cited on page 30)

[ela13] 2013. URL http://0b4af6cdc2f0c5998459-c0245c5c937c5dedcca3f1764ecc9b2f.r43.cf2.rackcdn.com/11719-icac13_herbst.pdf. (Cited on page 13)

[GB] T. S. Gerd Breiter, Frank Leymann. TOSCA Issue 15: On Implementation Artifacts and Deployment Artifacts. URL https://www.oasis-open.org/committees/download.php/45871. (Cited on page 18)

[LB12] R. P.-C. J. V. Lee Badger, Tim Grance. Cloud Computing Synopsis and Recommendations. 2012. URL http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf. (Cited on page 12)

[Liu13] K. Liu. *Development of TOSCA Service Templates for provisioning portable IT Services*. Master's thesis, Institute of Parallel and Distributed Systems, University of Stuttgart, 2012-2013. (Cited on pages 11, 25, 33 and 34)

[Lue11] C. Lueninghoener. Getting Started with Configuration Management. 2011. URL https://www.usenix.org/system/files/login/articles/105457-Lueninghoener.pdf. (Cited on page 27) [oas12] Topology and Orchestration Specification for Cloud Applications Version 1.0. 2012. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.html. (Cited on page 17)

[os$_1$3] Compute. 2013. URL http://docs.openstack.org/admin-guide-cloud/content/ch_introduction-to-openstack-compute.html. (Cited on page 15)

[pup13] What is Puppet? 2013. URL http://puppetlabs.com/puppet/what-is-puppet. (Cited on page 28)

[sma] Overview. URL www.smartfrog.org. (Cited on page 31)

[Sta10] K. E. Stavinoha. What is Cloud Computing and Why Do We Need It? 2010. URL http://isacahouston.org/documents/WhatisCloudComputingandWhyDoWeNeedIt.pdf. (Cited on page 1)

[the12] Concurrent Programming for Scalable Web Architectures. 2012. URL http://berb.github.io/diploma-thesis/original/024_scalability.html. (Cited on page 11)

[tos13a] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. 2013. URL http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html#_Toc347920704. (Cited on pages 43 and 49)

[tos13b] Topology and Orchestration Specification for Cloud Applications Version 1.0. 2013. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html. (Cited on page 20)

[UB] K. K.-O. K. F. L. J. W. Uwe Breitenbucher, Tobias Binz. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. (Cited on page 17)

All links were last followed on March 17, 2014.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature