MSc thesis

Computer Science

Automated Software Configuration for Cloud Deployment

Antero Vainio

October 28, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Sasu Tarkoma, Dr. Ashwin Rao, MSc Lirim Osmani, BSc Kalle Happonen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Antero Vainio

Työn nimi — Arbetets titel — Title

Automated Software Configuration for Cloud Deployment

Ohjaajat — Handledare — Supervisors

Prof. Sasu Tarkoma, Dr. Ashwin Rao, MSc Lirim Osmani, BSc Kalle Happonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	October 28, 2020	54 pages, 15 appendix pages

Tiivistelmä — Referat — Abstract

Nowadays the Internet is being used as a platform for providing a wide variety of different services. That has created challenges related to scaling IT infrastructure management. Cloud computing is a popular solution for scaling infrastructure, either by building a self-hosted cloud or by using cloud platform provided by external organizations. This way some the challenges related to large scale can be transferred to the cloud administrators.

OpenStack is a group of open-source software projects for running cloud platforms. It is currently the most commonly used software for building private clouds. Since initially published by NASA and Rackspace, it has been used by various organizations such as Walmart, China Mobile and Cern nuclear research institute. The largest production deployments of OpenStack clouds consist of thousands of physical server computers located in multiple datacenters.

The OpenStack community has created many deployment methods that take advantage of automated software configuration management. The deployment methods are built with state of the art software for automating different administrative tasks. They take different approaches to automating infrastructure management for OpenStack.

This thesis compares some of the automated deployment methods for OpenStack and examines the benefits of using automation for configuration management. We present comparisons based on technical documentations as well as reference literature. Additionally, we conducted a questionnaire for OpenStack administrators about the use of automation. Lastly, we tested one of the deployment methods in a virtualized environment.

ACM Computing Classification System (CCS)
Networks → Network services → Cloud computing
Software and its engineering → Software creation and management → Software post-development issues → System administration

Avainsanat — Nyckelord — Keywords

Networking, Virtualization, Distributed Systems

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Cloud Computing, IaaS, Software Automation, System Administration, DevOps, IaC

Contents

1 Introduction

As providing services over the Internet has become a common practice in the modern society, maintenance of information systems has encountered many challenges as well. While web-based applications have become more diverse, resource-intensive, and complicated, they face higher expectations with respect to usability, performance and security. Artificial intelligence and machine learning are being utilized in developed software [4], more ways to improve page load time are still being developed [32], and in the year 2018 Google reported 81 of the top 100 web sites using HTTPS by default [1]. Often times industrial software development has strict deadlines to follow. Many IT organizations aim to be agile and build their development practices around the idea of constant service improvement. It means that a software product is rarely considered finished but instead new features are being added to it and flaws are being fixed in a priority order. DevOps is a development philosophy that advocates small and frequent updates [43]. As a result, programs and sometimes entire system architectures may be updated more or less regularly. Similarly to IT services themselves automating repetitive tasks in hopes of achieving reliability and cost-effectiveness, software development processes aim to utilize automation whenever feasible.

Cloud enviroments offer on-demand computing resources for cloud consumers [6]. When deploying to a cloud, a user can expect customized servers being provisioned within seconds. It can be achieved with a combination of hypervisors and preinstalled software in virtual machines for instance. For further configurations, additional software automation tools can be used. Cloud environments are formed by physical host machines that lease virtualized devices such as virtual CPUs (vCPU), logical volumes, or virtual network devices. Users get typically charged for the time that they use some of these resources. Resources in a cloud environment can quickly be set up and down, preventing the users from having to pay for under-utilized servers. In terms of server provisioning, the cloud providers do not typically have the same luxuries as cloud users, since using virtualization is not optimal due to the added processing overhead. However when it comes to configuring hundreds or even thousands of physical server computers, automation is once again seen as a potential solution. Yet many traditional cloud administrators are still relying on manual maintenance operations.

This thesis explores different popular methods for deploying and administering cloud environments with the help of software automation. The deployed software is a combination OpenStack services. OpenStack [36] is a collective of open-source software projects for running cloud servers. It is currently the most widely used project for creating private clouds. It includes official repositories for various automated deployment methods. Most of the methods provide a basis on which to build a solution customized for own working environment.

We organize this thesis followingly. In Chapter 2 we present the motivation to the topic and introduce some core concepts and technologies. Sections 2.1 and 2.2 will provide short introductions to some of the popular software solutions. In Chapter 3 we will present some of the official deployment methods for OpenStack. In these two chapters we will provide answers to research questions RQ1 and RQ2 shown on Figure 1.1. We will provide reasoning to the answers based on technical documentation and reference literature. In Chapter 4 we present results from a small-scale survey we conducted for OpenStack administrators. Questionnaire discussed the use of automation for administering OpenStack deployments. In Chapter 5 we present a test deployment conducted with one of the automated deployment methods. These two Chapters will provide answers to research question RQ3 about the benefits of automated configuration management. Answers will be based on custom experience gained in method evaluation as well as experience of professional OpenStack administrators. In order to avoid bias, we will compare results from both the questionnaire and evaluation to larger datasets such as OpenStack user surveys [26]. Assets used in the deployment evaluation can be found in GitHub repository *https://github.com/AnteronGitHub/openstack-deployment*. Raw data of the quetionnaire results as well as the sources of this thesis paper can be found in GitHub repository *https://github.com/AnteronGitHub/thesis-paper*.

This thesis provides two major contributions. Firstly we present a view of the contemporary methods for using automation in managing a deployment of a complex distributed system, more particularly an OpenStack cloud. Secondly we will outline benefits of using and developing automation tools based on experiences gained from real-world operations as well as empirical study. We will see that with a wide variety of different options available, some of the deployment methods may be more suited for specific use purposes than others. Awareness of the differences in the design of the deployment methods can help to make a decision about the tool that will be adapted for a particular project. Some of the tools provide more features than others. Using a sophisticated deployment method which

RQ1 What are the key factors affecting the design of different deployment methods, and how do the deployment method differentiate from each other?

RQ2 What kind of features do the different deployment methods offer? Which components are used in deployments?

RQ3 Who will benefit from the development of the software automation tools, and why?

Figure 1.1: Research Questions

requires high maintenance might not be the best solution in case most of the provided features go unused. As both the use and the development of the automated solutions requires constant effort, the benefit of automation is not always obvious either. The primary benefit of using automation is the reliability it provides for repating administrative tasks. Repeatable operations enable the use of frequent updates to IT services with decreased risk of human error resulting from manual operations.

2 State Of The Art

This thesis is motivated by the current trends and contemporary practices used for administering IT infrastructures. We will shortly introduce some of these trends next. Cloud computing has reached nearly a de facto status for large scale infrastructure provisioning [6]. While cloud services automate many administrative tasks, another increasingly utilized method in IT service management is the use of software automation for system configurations. Usually whenever there is a trend or a commonly followed practice in the field of computer science, there are various approaches taken to achieve it. Different approaches may share similarities in some aspects while at the same time they can be fundamentally different in other aspects. As an example, virtualizing software runtime environment can be achieved by using fully virtualized operating systems with hypervisor like VMware ESX, paravirtualized operating systems with hypervisor such as Xen [7], container-based virtualization [41], or an application-level virtual machine such as Java Virtual Machine. All of these solutions share the same end goal: to provide runtime isolation for applications running on the same physical host machine. One of the reasons for the existence of various alternative solutions can be commercial competition, another can be different preferences in terms of software architectures.

There is a lot of diversity when it comes to the use of software solutions for IT service management. However for the most parts IT services themselves have similar requirements related to the quality of service. Services have requirements such as highly availability, reliability, efficiency, usability, and security. At the same time service management needs to be cost-efficient. When it comes to providing high quality with low cost, proper use of automated solutions may prove to be valuable. With successful Internet based services, scale makes a big difference. Spikes in the number of active users can affect the quality of service significantly, even to the point where the service is unusable. While optimized software has an impact on service level, there is a point after which the only way to keep the quality of service high against heavy loads is to increase the computing capacity. At the same time, having under-utilized servers wastes resources and thus increases the cost of providing service, resulting in competitive disadvantage in the market. This must be considered both in the design of the application software and in the infrastructure architecture. Number of users for a web-based application can change a lot during the course of a regular business day. Many services have repeating patterns in their level of

usage and the number of users for a particular time of day can be predicted with high accuracy. If usage level has drastic changes and follows a repeating pattern, provisioned computing resources can be scaled correspondinly resulting in cost savings for optimized resource utilization. With the help of detailed analysis and the use of cloud platforms, this can be achieved.

When maintaining an IT service, many administrative tasks become frequent. When scaling system up, deployed applications need to be installed, configured and integrated to the environment. When scaling down, it has to be ensured that no applications are requesting inexistent application instances. At basic level, these operations include software package installations, configuration file modifications, and potentially secret management. When done manually these tasks have a high risk of failure due to human error. According to a study, more than 21% of IT service interruptions were caused by human error [38]. Another study stated that most of the incidents investigated were related to change planning [35]. These studies suggest that even when infrastructure is not scaled rapidly, there is a high pressure towards IT administrators for keeping up the quality of service. Partial automation has been suggested as one of the approaches for avoiding human error it IT service management [38]. Still many IT administrators are relying on manual operations.

Another recent trend in IT service management is the DevOps movement [43]. The term DevOps is often used to describe a principle of speeding up development by simplifying the process of publishing new software versions while assuring high quality. By using efficient development processes and automated tools, the time it takes to deploy a new software version to runtime environment or package distribution can be cut, lowering the overall cost of IT development. DevOps movement was motivated by agile software development principles and the realization that communication between software developers and operators often times ended up being an unnecessary bottleneck for project speed [12]. Some of the goals of DevOps movement, such as fast and automated software release process, can be achieved with a proper use of cloud platforms and software configuration management tools. They allow application developers to define the deployment process either completely or partially with input files for computer programs. Deployment is executed by running the programs as opposed to manually following application specific instructions. Use of automation can potentially avoid some of the errors resulting from miscommunication between developers and operators [12].

Cloud computing can be used for provisioning computing resources rapidly. For customized applications, additional configuration is often necessary. Software configuration

management is a potential solution for avoiding human error in repetitive configuration tasks as it enables these configurations to be applied programmatically. While providing operational reliability, it has additional benefits such as fast execution, and documentation value provided formally defined configuration tasks. This chapter provides a short introduction to the concepts behind cloud computing and software confuration management. We will introduce some contemporary software solutions for these concepts and provide answers to research question RQ1 (Figure 1.1) about the design factors and differences of the methods. Key design principles and differences of the tools are presented as they will affect any deployment methods presented in Chapter 3.

2.1 Cloud Computing

For the last decade, cloud computing has been a common paradigm for IT infrastructure management. It provides benefits such as high server utilization and dynamic scalability with a pay-per-use price model for outsourced server infrastructure. Cloud computing is being widely used in various industrial fields, including telecommunication [45], retail, finance [17], and scientific research [8]. As a result, there is also an inreasing number of public cloud platforms available, most notably Amazon Web Services (AWS) [3], Microsoft Azure [22], and Google Cloud Platform (GCP) [14]. As with many practices in the field of computer science, cloud computing lacks a universally agreed formal definition [6]. However common to most of the contemporary cloud platforms is the ability for the end user to provision computing resources over HTTP interfaces, mostly by using Restful APIs. This makes it possible to provision infrastructure programmatically. Cloud consumer avoids the need to request infrastructure operations from administrators, while administrators do not have to spend time for repetitive tasks related to infrastructure provisioning.

Due to its on-demand nature, a term often used to describe cloud resource provisioning model is *Infrastructure-as-a-Service* (IaaS). It emphasizes the fact that infrastructure can be provisioned through a well-defined interface. When service provider offers more high-level assets on top of which to build applications, the service model is called *Platform-as-a-Service* (PaaS). In *Software-as-a-Service* (SaaS) model, in addition to service provider provisioning infrastructure, it takes care of software licencing and configuration [36]. These models can be generalized to *Anything-as-a-Service* (XaaS), where anything stands for any resource that is provided through programmable interfaces. Reason for varying service models offered by cloud providers, is the fact that clients have varying needs in terms of

outsourced software solutions. Some clients may want an easily adaptable software configuration with no need for additional configurations and expertice related to the software; in case of issues with the software they are willing to pay for customer support. At the same time there are clients who use cloud providers only for leasing infrastructure in order to avoid having to maintain own datacenters. For the former type of client, SaaS may be the right service model, and the latter might prefer IaaS or PaaS.

When it comes to providing XaaS model services, cloud service providers have advantages for making new products. As they know their cloud platform in detail, they are able to produce optimized solutions for particular use cases. They also have full access to their platform, making it possible to avoid any workaround solutions that may arise from having to tailor solutions to interfaces provided by other services. Reasons why some clients may still prefer to configure software being hosted on cloud platforms themselves, are saves in costs, and the ability to tailor solutions to their particular needs. XaaS products have to be generalized enough so that they can be used by a large customer base. The more a product is generalized, the less it can be directly adapted to a particular use case. There are different approaches to dealing with this dilemma, such as configuration options or large catalogs of tailored services.

Another characteristic in cloud environments is seemingly limitless resources available on-demand, enabling the infrastructure to be scaled rapidly based on current usage [6]. As cloud platforms can be used for various computational tasks, investing in large data centers is more reliable for cloud providers than for any particular industrial field practicioner. With more computing resources available, and the ability to provision them for an execution of a particular computational task with no extra cost, some cloud consumers may use the platform to lease many vCPU hours for a short period of time, instead of using less vCPU hours for a longer period of time. This way of using cloud platform is more typical with computational applications such as with business analytics, and scientific research.

Ability to think of a cloud platform as having limitless computing resources makes it possible to consider cloud platforms alternatives for *High Performance Computing* (HPC) or *High Throughput Computing* (HTC). In general this would be achieved by scaling computations horizontally. If a heavy computational task can be split to subtasks that can be run in parallel, they can be executed on different hardware, shortening the overall time of computation. Similarly, if a large data stream can be split to smaller streams to be combined at receiver, the overall time of transmission can be shortened by using different network links. MapReduce [11] is an application developed to help parallelize computa-

tional tasks by defining them as map and reduce fuctions. This model makes it possible to parallelize and thus scale computations horizontally by the runtime system.

Although cloud platforms primarily provide scalability horizontally with a number of provisioned instances, some of the larger cloud platforms provide also instance flavors with highly performant computing hardware. These flavors may be provided for bare-metal services to reduce computational overhead of virtualization. While vertical scaling is not a feature distinct to cloud platforms, they benefit from the high user base. This way while the platform grows, as more varying computing flavors are being provided, computers with high-quality CPUs and GPUs can be aquired in hopes that they will not go unutilized.

Cloud environments may be public or private. While public and private cloud services do not functionally differ from one another, private cloud services are restricted to particular end users and public cloud services are offered more or less without restrictions. As a result, generally public cloud environments are larger in size, known examples of such environments include AWS [3], Azure [22] and GCP [14]. Some cloud platforms are built on top of a number of other cloud platforms. Such architectures are called *multi clouds*. Multi clouds may use a particular cloud provider for certain types of resources, such as virtual networks to be able to use quality wide area network infrastructure. They may also use other cloud platforms for provisioning resources that they do not have available at the moment.

Motivation and defining characteristics of the cloud computing paradigm affect various software solutions built around it. In order to provide an exhaustive description of the features and the components related to the deployment methods presented later, having fundamental aspects of cloud computing outlined is crucial. The topics about cloud computing that we have presented make the foundation for the software solutions we present in this thesis. We will next introduce software essential to the topic of this thesis work. Firstly we present OpenStack, which is the software that will be deployed with the methods presented in Chapter 3. We will then present Docker and Kubernetes which have recently gained popularity among application developers and operators. They can be used for managing applications running in containers, thus making easily portable to a cloud environment. Due to their recent popularity they have been adapted by the OpenStack community and are being used in some of the OpenStack deployment methods.

2.1.1 OpenStack

OpenStack (OS) [36] is a family of open-source software projects for building private cloud environments. Compared to other alternatives, such as Eucalyptus [13] or OpenNebula [25], OpenStack has become the most widely used open-source software used for private clouds. It was originally developed by NASA and Rackspace, and published openly in the year 2010. Since then it has been developed by various organizations, including Cern nuclear research institute as well as individual contributors. It has been battle-proven for a reliable cloud operation by its large user base [23]. While OpenStack can be used for public clouds, it is most commonly used for private or multi cloud environments. It is currently being also developed to provide support for edge computing infrastructures. OpenStack services are not alternatives to hypervisors, logical volumes or SDN controllers, but rather they use them for providing computing resources through RESTful APIs in a scalable manner.

At its core OpenStack is an example of an IaaS platform. It provides functionality for infrastructure provisioning through various services. Infrastructure provisioning is also the primary use case of OpenStack for most of its users. Many of the OpenStack services can be extended for providing XaaS, such as *VPN-as-a-Service* or *Database-as-a-Service*. Some of these extensions can be installed as plugins for the appropriate OpenStack services. Otherwise any extensions have to be implemented. In addition to installing OpenStack with low-level open-source projects, OpenStack community has an official marketplace for commercial high-level OpenStack-based software products. Commercial OpenStack products may provide official integration to other platforms, or simplify and provide support for the installation of OpenStack.

OpenStack consists of different services, not all of which are required for an operational cloud. OpenStack services may also be used as a part of other distributed systems without deploying a full OpenStack cloud. At basic level, OpenStack services are Python application servers and related worker agents. They use other infrastructure services, such as message queues, databases, caches and proxies.

In order to grant access to resources, OpenStack services need to be able to authenticate and authorize requestors, which may be end users or other services. Amongst OpenStack services, Keystone provides Identity and Access Management in cloud. Since OpenStack services use it for access control when communicating with other services it is the first service to be installed in an OpenStack deployment.

An essential resource in any computing environment are the actual computing units. Traditionally OpenStack has provided computing resources by using hypervisors, such as ESX or KVM/QEMU. OpenStack compute service is called Nova and it was one of the first services developed. Nova receives requests that describe the specifications for needed virtual machines. Nova then schedules the creation of the virtual machine from the hypervisor. Virtual machine images used by Nova are provided by Glance. Typically guest operating system images are built by cloud administrator by using tools such as Diskimage-builder developed by the OpenStack community. Additionally users can create virtual machine images from their active Nova server instances.

In order to access virtual machines created by Nova, network access is required. Virtual network devices are used for on-demand network provisioning. Originally virtual networks were provided by Nova service, but currenlty there is a specific OpenStack service, called Neutron, that is responsible for network provisioning. Even though nova-network is considered legacy, older OpenStack deployments may have been using it long after Neutron was published. As an example, Cern which has been a long-time OpenStack user, reported still using nova-network in its deployment in the year 2015 [8].

OpenStack has multiple alternatives for persisting guest data. Service providing block storage is Cinder. It provides mountable logical volumes for guest instances. When instance is terminated, data stored in the volume will persist. Another available service for persistent data is Swift object storage. It processes stored data using object model. Lastly Manila is an OpenStack service that provides shared file system.

There are some OpenStack services that are not required for a functional cloud, but are almost always included in a cloud deployment. One of such services is Horizon, a web UI for managing cloud tenants. Users can login to it with their user credentials and do most of the operations available through other OpenStack services. Another optional but commonly deployed OpenStack service is Heat orchestration engine. Heat takes a stack template file as an input and creates the resources described in the file. This makes instantiating complex cloud stacks quick and repeatable. In addition to creation, Heat has functionality for updating deployed stacks based on changes made to its template file. Heat provides a service similar to that of CloudFormation on AWS and it supports its syntax. Ability to create system infrastructures with input files that are parsed by computer programs is sometimes referred to as *Infrastructure-as-Code* (IaC).

Figure 2.1 illustrates interrelations of different core OpenStack services in an example logical architecture. Services communicate with HTTP requests through RESTful API

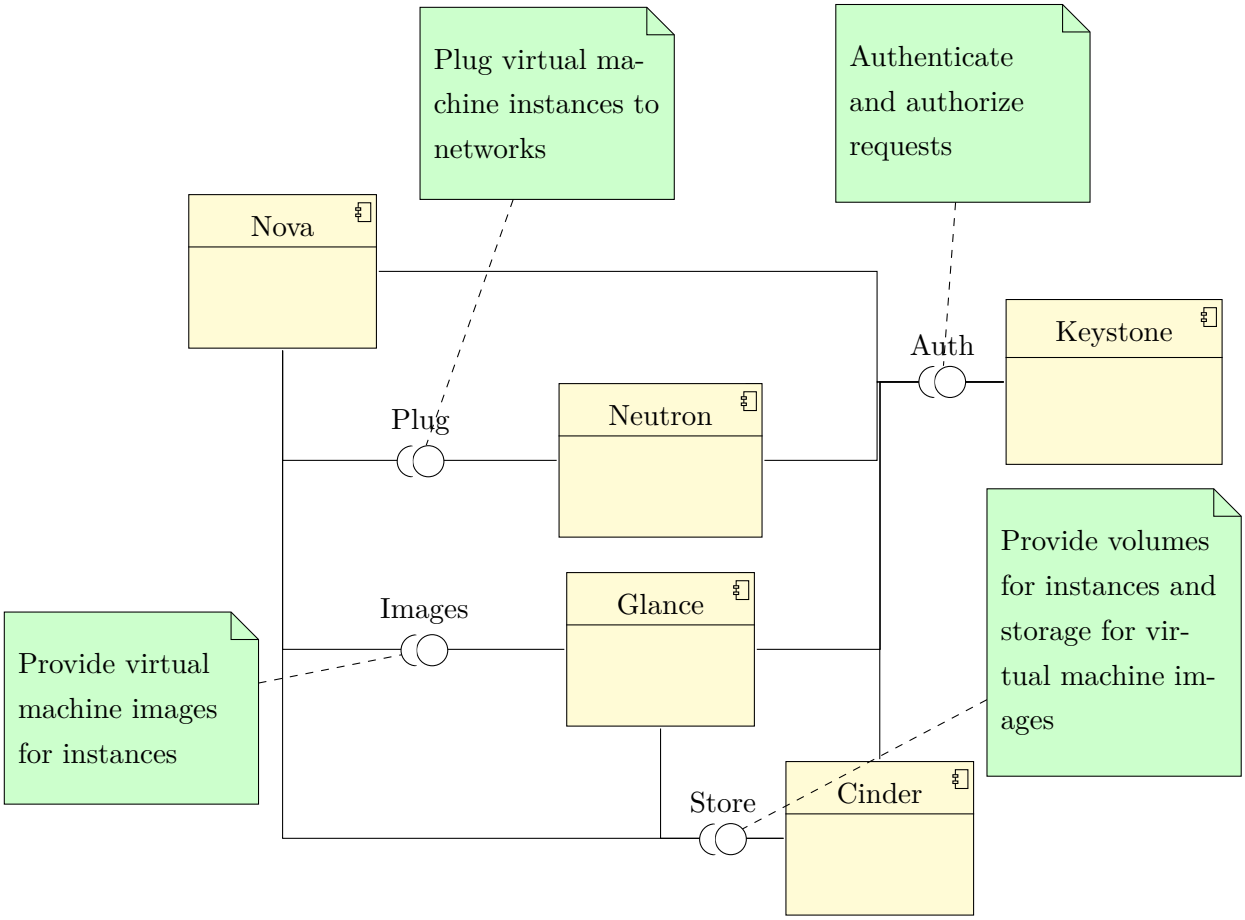

Figure 2.1: Component diagram of core OpenStack services in an example architecture

servers. In addition to services being requested by core services included in the diagram, they may receive requests from other services, such as Horizon, or Heat. They may also be directly requested by end users. The component diagram in Figure 2.1 displays the high-level functions of different services. There are many different ways OpenStack clouds can be designed. For instance Cinder does not have to be used as a storage service for machine images hosted by Glance, but other solutions such as Swift object storage can be used instead.

OpenStack deployments used for providing cloud services have to be designed according to the needs of the service. Quality requirements related to availability, efficiency, security, and any potential Service Level Agreements have to be taken into consideration when designing a cloud. Since cloud networking is more complicated than regular data center networking, decent knowledge of the underlying technologies like software defined networking have to be assured for both the designers and the operators. OpenStack Deployments typically include host machines of a few different flavors. Host machines may be classified

as compute, storage, networking, and controller nodes. Each of the node types includes only OpenStack services or the infrastructure services that are used for its core purposes. Appropriate hardware to support its use purpose should be installed on a node machine. Nodes may be classified differently depending on the use case of the cloud. Host flavors have an important role when designing a cloud architecture as they have a big impact on the cloud performance, power consumption and overall cost.

Compute nodes are used for housing guest virtual machines. They provide guests with CPU cores, RAM, root file system, and network access. The more CPU cores, and RAM the node has the more guests it can run simultaneously. It is possible to overcommit vCPUs with a defined ratio. Overcommitting increases the capacity of CPU cores with a tradeoff efficiency in guests. For a compute node in a production deployment having 16 CPU cores would be a feasible amount. Compute nodes should have available RAM in a proportionally to available cores. Guest instances are created from instance flavors that specify their resource use. Flavors in the deployed cloud can provide guidelines to the compute node hardware specifications.

Guest root volumes may be housed in shared file system in a network or in a node, or they may be provided by directly mounting a storage device on the compute node's file system. In case the compute node file system is being mounted directly, compute nodes need to have available volume capacity when creating new guest instances. While using network volumes for guest root volumes may provide better capacity utilization, it suffers from slower read and write operations. Other options related to volume provisioning is the device technology such as choice between tape or SSD storage. Storage nodes provide data persistence in the cloud. There are various alternative backends for distributed storage across nodes, such as Ceph, GlusterFS or NFS. Volume devices are naturally the most crucial hardware for storage nodes.

2.1.2 Docker

Docker [9] is a software for managing containerized applications. Containers provide process isolation by using Linux Kernel CGroup [21] functionality. Use of containers simplifies management of multiple microservices on same host machines. It adds overhead compared to regular process isolation, but is more efficient than using hypervisors [41]. Docker containers use Docker Engine to communicate directly with host operating system kernel. This keeps containers lightweight since the kernel functionality does not have to be emu-

lated by the hypervisor. Podman [31] is a daemonless alternative to Docker.

In addition to providing runtime isolation for application processes, Docker includes functionality for creating container images. Images can be created from existing containers or based on files describing container configuration tasks. Created images can be published to Docker *registries*, which can be public or private. DockerHub is a public container registry which houses official Docker images, and is also available for other publishers. Docker's image build system is sophisticated and provides multi-layered caching for created images. Image is cached after distinct operations in the creation process. By running configurations in the proper order only a subset of them have to be repeated when image is modified and rebuilt. As an example, by updating base image package manager as the first configuration operation and installing application dependencies later, updated package manager can be used if application dependencies need to be reinstalled.

Docker also includes functionality for managing other resources than container. Docker Engine can provision volumes for persisting data and networks for container connectivity. There are different drivers for volumes and networks. Simplest drivers simply bind resources directly to Docker host. For scaling Dockerized application deployments overlay networks can be created for containers so that they can connect to containers running on different hosts by using layer 2 semantics.

One of the reasons for Docker becoming popular among software developers is how it simplifies container creation and sharing container images. Since container are portable to other hosts running Docker Engine, applications can be deployed easily, while having any software dependencies included in images makes deployment quicker and more reliable. Lightweight container lifecycle also enables containers to be recreated as a method of disaster recovery. Docker's benefits for application deployment are ideal for teams adapting DevOps principles. Not only do image registries provide an interface between developer and operator teams, they also make development environment configuration simple enough to be appealing for application developers. Continuous deployment pipelines can use from Docker registries similarly how they would use software package repositories.

By installing all the used software into containers, developer can keep the host operating system environment uncluttered. Uninstalling software and its files is simply done by removing the container. Similarly to application developers benefitting from uncluttered host system, operators running applications in containers do not have to worry about cleaning trash when updating outdated software versions. Containers provide logistics for managing software that updates frequently. Managing multiple microservices running on

a single host becomes less error-prone with isolation provided by containers.

2.1.3 Kubernetes

Kubernetes [10] is a software for managing Docker container clusters. It was developed by Google which has been running applications in containers long before Docker became popular. Kubernetes is intended for providing highly available production environments by running Docker container. It provides a RESTful API for cluster management. Due to container overlay networks it can be used for both centralized cloud platforms, and de-centralised edge computing infrastructures. Applications can be deployed to a Kubernetes cluster based on YAML formatted definitions. Additionally Kubernetes includes a package manager called Helm which can be used to bundle more complex applications into templates called charts. While applications can be deployed in a Kubernetes cluster without Helm packages, having Helm charts helps to bundle applications.

Kubernetes consists of API server, scheduler, cluster controller and worker nodes. Clusters may provision resources from cloud providers by using separate cloud controller process. In order to optimize Kubernetes cluster runtime for production environments, bare-metal Kubernetes clusters may be run. In this case cloud controller is not necessary. Kubernetes uses Docker Engine for managing applications running in containers. It can scale application deployments by creating and deleting containers. It also provides health monitoring for deployed applications and can automatically recreate containers that have entered failed state. Other infrastructure resources can also be managed by Kubernetes. It manages firewall rules within networks, and provides ingress to application containers through reverse proxy.

Kubernetes is specifically useful for developers since application stacks can be created faster than with hypervisors. For administrators Kubernetes provides functionality for deploying, updating and scaling applications. As a tradeoff, having to install and operate Kubernetes cluster adds overhead to maintaining applications. There are commercial Kubernetes services, that offer configured clusters for application developers. This makes it easy to operate scalable web-based applications without the need to administer or configure host systems, which is favourable for teams consisting only of software developers. On the other hand administrators can provide hosted Kubernetes clusters as a service. Kubernetes provides an interface between software developers and administrators in a similiar manner than cloud platforms, by providing a RESTful API.

OpenStack includes a service called Magnum, which can be used for providing Kubernetes clusters as a service. Magnum uses Heat orchestration for configuring cluster hosts that are created with Fedora Atomic [33] machine images, which include preconfigured Kubernetes software. Similarly to other OpenStack services further customizations to Magnum are also possible. In addition to Kubernetes being available to be provisioned as a service, there also exists methods to deploy OpenStack services into Kubernetes clusters. Since virtualization allows various possible stacks to be built from the same components, it may sometimes create confusion. How different stacks are built depends on the intended use cases, but for the most parts, the more virtualization stack is used, the more overhead is created, and thus it should be avoided in production-grade environments.

2.2 Software Configuration Management

Large distributed systems, such as OpenStack deployments, consist of many configuration items, such as databases, message queues, proxies, and memory caches. Infrastructure services need to be configured according to applications using them. As infrastructure services, or the application services, are updated, compliancy to other services must also be ensured. Shell scripting has traditionally been a common way of automating different administrative tasks in Linux environment with its counterpart being batch scripting in Windows. Recently different software automation tools have become a popular replacement for shell scripting.

Automation tools commontly take an input file that describes operations to be executed, and apply it. Input files are often given in declarative format, like JSON [15] or YAML [44], as opposed to procedural format of script files. The difference between the two is that input files describe the desired outcome and not the methods of achieving it. How configuration is applied to the targeted system is determined by the automation tool implementation. Since the methods of configuring the system are for the most part not relevant to administrators, using declarative input files simplifies automating software configuration management. Automation tools provide reliability and cross-platform support by using application layer modules for software configuration tasks. They can abstract operating system dependent considerations for common administrative tasks. This also enables configurations to be shared more easily among administrators.

Typical recommendation when designing automated configuration tasks is to try to make the operations idempotent. This means that the automated task can be repeated indef-

initely without changing its outcome. In other words, if the targeted system is in the desired state, applying the configuration does not change the target system state. Idempotence can be assured at low level by the automation tool, but when making high level configurations, it has to be taken into account by the configuration designer. Having idempotent configuration tasks makes the use of automation tool more reliable. With complex configurations ensuring idempotency may be difficult.

One crucial difference between software configuration management tools and prebuilt virtual machine or container images, is that configuration management tools are responsible for applying configurations, while machine images contain the necessary software and configurations. It is possible to use software configuration management tools for creating machine images, but most of its usefulness comes from dynamic maintenance operations. Dynamic configurations provide versatility and the ability to request newest package versions over a network during execution. Tradeoff is that the configuration takes longer than having preconfigured software available at disk. For repeating configuration operations, such as CI/CD pipelines, virtual machines containing some of the required software is likely more beneficial. Another option is to use cache mechanism for installed dependencies. Optimizing build time of a functioning pipeline is often possible in many different ways.

There is a large number of different software automation tools available. While there are differences in the architecture of different software configuration management tools, some similarities exist as well. Most of the software configuration management tools provide some mechanism for sharing configuration methods. This makes it possible to create communities of administrators. Similarly to open-source software projects, configuration tool communities often help to develop the tool as well. As with many software projects, configuration management tools depend on active communities in order to keep being developed. Some tools that are used in the deployment methods presented in Chapter 3 are introduced next in more detail.

2.2.1 Ansible

Ansible [5] is a software configuration management tool developed by RedHat with community contributions. Ansible is an open-source program written in Python. It can be run with ad-hoc commands, without written input files, but generally it is used with YAML formatted files describing operations to be executed. Use of Ansible is based on

```
---
- name: Copy files to destination
  copy:
    src: "files/{{ item }}"
    dest: "/opt/{{ item }}"
  with_items:
    - file.txt
    - other_file.txt
```

Figure 2.2: Example Ansible task definition

establishing SSH connection to target machine and running Python modules executing administrative tasks. Hence the only software requirements on the target machines are SSH daemon and Python compiler, both of which are available in basic installations of many mainstream Linux distributions. Not requiring additional software makes the use of Ansible simple and keeps target machines optimized.

Architecturally Ansible consists of a few concepts that divide the responsibility of configuration execution. At lowest level, tasks are executed in *modules*, which are Python scripts that generally use standard library API's for interacting with host operating system. For the most part the modules do not need be written when using Ansible, since Ansible standard library includes modules for the most common administrative tasks. One of the standard modules enables user to execute arbitrary shell commands, avoiding the need to write modules for programs that do not have one implemented.

Input files of Ansible reference modules via units called *tasks*. In essence, tasks describe the module to be executed and arguments passed to it. Tasks may include additional metadata, such as descriptive names that are displayed during execution. Variables can be used with defined tasks in order to make them re-usable. As an example, arguments passed to a module could be defined with variables. Figure 2.2 illustrates an example of Ansible task, which uses *files* module to copy files from *files* directory to the *opt* directory on targeted host. *src* and *dest* are arguments for the *copy* module. *with_items* keyword specifies a loop that binds listed entries to the *item* placeholder.

Complex software configurations consist of many tasks. In order to structure templates, Ansible uses *roles*, that describe higher level administrative operations. Roles are grouped in directories that follow a conventional structure that Ansible expects. In order to roles to make changes to target machines, they must include some tasks. Roles cannot be executed

directly, but rather have to be referenced externally. They are supposed to be kept self-contained, in order to port them easily. Roles can define default variable values. Input files that Ansible can execute are called *playbooks*. They describe hosts to be targeted, and reference roles to be applied or tasks to be executed. Environment specific information, such as host IP addresses, and customizable configuration parameters are described in *inventories*. Inventories can be built from groups of hosts, and variable values can be specified for individual hosts or for host groups.

While Ansible is easy to get started with, and it provides a modular method for building configuration libraries, it does not provide built-in functionality for provisioning infrastructure or health monitoring deployments. One option is to implement these tasks with Ansible by leveraging IaaS providers [39]. In its simplicity Ansible is a powerful tool for managing system configurations.

2.2.2 Juju

Juju [16] is an application modeling tool developed by Canonical. It is capable of deploying, configuring and scaling software. Juju was originally written it Python, but its current version is implemented with Go programming language. It still has an API for developing modules in Python.

Juju uses a *cloud* abstraction for provisioning infrastructure. It includes functionality for interacting with public cloud platforms like AWS, GCP, and Azure, and a number of private clouds, including OpenStack. For production application deployments, Canonical recommends using MAAS [40], which can be installed in datacenter for provisioning bare-metal infrastructure. Juju also supports manual clouds for deploying in pre-existing server infrastructure. Manual clouds lack some of the functionality available when using other clouds, mainly related to automated provisioning.

Juju uses a concept of *application* for managing deployed software. All of the components, including infrastructure services needed by deployed software are, called applications by Juju. Operations related to applications, such as infrastructure provisioning, installation and scaling, are executed by Juju by running software packages, called *charms*. Execution of charms is triggered by administrator running commands on Juju client.

When deploying applications, Juju creates a special management node called *controller*. Controller maintains a database including data used by *models*. Models manage environment specific information of deployed components, such as applications, storage volumes

and network spaces. Hence models provide an interface between application model, and its implementation in cloud being used. Models additionally control access to infrastructure.

Juju is especially useful for application developers as it provides a simple set of commands for operating application deployment and supports many of the common infrastructure providers. Configuration tasks are implemented with software modules using software libraries, making it more approachable for developers familiar with the tools. Administrators who do not use Python scripts for system configurations might find tools that provide declarative input format more approachable. In case all of the configurations are provided by application developers, Juju can provide an interface between developers and administrators.

2.2.3 Puppet

Puppet [34] is a Ruby based configuration management tool. Similarly to other presented tools, configuration tasks are grouped in modules. Puppet has open-source and commercial versions. Commercial version, called Puppet Enterprise (PE), simplifies large-scale configuration management by providing grahical user interface.

Puppet runs *agent processes* on target machines to keep them in desired state. Desired state can be described with Puppet's *Domain Specific Language (DSL)*, which is a declarative coding language. Puppet deployment includes a *master server*, which stores desired states in database called *PuppetDB*. Agent processes translate Puppet DSL into executable commands. Master and agents use HTTPS protocol for communication [34]. Information about target hosts is gathered by agent processes with Puppets inventory tool, *Facter*. Gathered data is sent to master server in Puppet DSL format in files called *manifests*. Based on received manifests, master server compiles JSON files, called *catalogs* that describe the desired state to agent processes. Puppet separates configuration data from the code executing configurations by using tool called *Hiera*. Separating code from data makes modules more testable [34].

Puppet provides accurate configuration monitoring by using agent processes on target hosts. At the same time it includes many configuration items making the installation process more complex. Overhead of configuration tool has a potential of vendor lock-in and means that the tools must provide value worth the added work.

	Ansible	Juju	Puppet
Module compiler	Python	Go/Python	Ruby
Uses agents on targets	No	Yes	Yes
Provisions infrastructure	No	Yes	No
Provides health monitoring	No	Yes	Yes

Table 2.1: Key features of the software configuration management tools

3 OpenStack Deployment

Largest OpenStack deployments consist of thousands of nodes in multiple datacenters. In the year 2015 Cern reported 5500 nodes being used across two datacenters, running more than 12000 virtual machines [8]. World's largest telecommunication company China mobile has published multiple scalability tests with OpenStack clusters of 1000 and more nodes [45]. Paypal operates a 4000 host node OpenStack deployment, and Walmart invested in an OpenStack cloud consisting of more than 100000 CPU cores [17]. Various approaches has been taken to cloud administration and update process by different organizations. Cern for instance has traditionally followed a model where one item is upgraded roughly every two weeks [8], minimizing the impact of potential failures.

Installation of a functional OpenStack cloud includes many steps. Most of the OpenStack services use infrastructure services, which need to be installed and configured. As new versions of OpenStack services are release biannually, configuration management is a big part of operating an OpenStack cloud. After the initial deployment the services will need to be upgraded, while ensuring the cloud operation. In practice the initial deployment is only a small part of the OpenStack cloud lifecycle. It is common for the tools used for initial deployment to be also used for many of the administrative tasks during the cloud operation. For the most parts OpenStack is beneficial for large-scale deployments. Many of the infrastructure services are intended to provide scalability for the OpenStack services. Using them for small-scale deployments adds performance overhead and complexity that does not provide significant value. Small-scale datacenters do not benefit from RESTful APIs for infrastructure provisioning as much either since their use level is likely to be low.

Different parts of installation process has been automated as much as possible by the OpenStack commuity. OpenStack services may be installed from source with pip package management tool for python. Many Linux distributions' package management tools, such as apt for Ubuntu and yum for CentOS, have distributions of many OpenStack services as well. However depending on the used repository, distribution packages might not include the latest versions of the software. Software configuration management tools have also been used by OpenStack community for some time now. Many of the deployment methods for automation tools have official OpenStack repositories. Some deployment methods are also designed for particular use cases, such as easy setup of a development environment or use

for automated testing.

3.1 Deployment Methods

OpenStack community has created official deployment methods with many popular software configuration management tools. The methods have been during different time periods and they follow the best contemporary practices. Many deployment methods may have been developed for a particular use case. Some of the methods provide a simple setup for an OpenStack development environment, while others provide more configurability with little assumptions of the target environment. Currently many new developers being introduced to OpenStack start with an automated installation of the services. Being able to install OpenStack proof-of-concept easily without knowledge of the automation tool makes getting introduced to OpenStack easier and as a result increases the potential size of the developer community. OpenStack as many other open-source software projects is developed with community contributions, so having an active community provdes an indirect benefit the software itself.

Setting up a development environment is drastically easier than to deploying a production-grade OpenStack environment. Only practical requirement in development environments is that the developed software and can be executed and modified. Having all of the applications deployed to the same host machine is often sufficient. This makes it possible to make accurate assumptions of the target environment, and avoid manual configurations. Additionally there exists a number of popular tools for development environments, such as Docker or Vagrant. This way setting up a development environment can be achieved by running a single command. When comparing different deployment methods, this thesis focuses on production environments. This means that the deployment should be configurable for the needs of Service Level Agreements. Requirements environments may include high availability, responsiveness, security, and horizontal scalability. A Production-grade OpenStack clouds should be designed based on its use purposes without considering what automation tools will be used. As OpenStack consists of a complex set of services it can be used for various computing environments. Some clouds invest on computational power, while others may focus on high network throughput or storage capacity. Once the architecture of the cloud is established, any potential automation tool should be decided based on how well it can be used to support the intended architecture. Cloud operators should also feel comfortable with the management tool of choice as it will likely be used in many

lifecycle operations of the cloud.

In practice, when choosing an automation tool to be used for operating the cloud, one important consideration is, how easy it is to find operators capable of using the tool. This can be the most important factor for project managers, as even the most optimal tool is not practical if there are no operators familiar with it. Many automation tools are promoted by companies making the decision of the automation tool less of a technical consideration and more of a business decision. An organization might end up selecting the tool that is developed by a company whose other products are already in use. Often times products developed by the same company have better support for interoperability. Even though using a family of software products from the same vendor has its benefits, it can cause an architectural vendor lock-in. In case one of the products gets discontinued, replacing it with other solutions can become difficult when other software solutions are built around it. If the product provider company gets aquired or goes down altogether, it may have a significant effect on other organizations relying on their products or support. One way to avoid vendor lock-in is to use software products from various organizations. This requires more effort and expertise than relying on products from the same vendor. Alternatively similar products from various vendors can be used simultaneously. This approach is likely cumbersome and potentially expensive.

Since OpenStack is supported by many organizations, there is a lot of different alternatives for automation methods. Some of the methods are developed by open-source communities, some by commercial organizations. Some of the methods can be combined, or even have an official OpenStack project for combining the methods. Some deployment methods that are suitable for OpenStack production environments are introduced next. Methods will be approached from the perspective of the research questions shown on Figure 1.1. Software dependencies for different features of the deployment methods are displayed on Table 3.1. Since all of the deployment methods listed below are open-source projects, they can theoretically be modified for any configuration needs. However in order to keep their comparisons reasonable, all of the required features should be achieved with configuration instead of changes to the methods. In practice it is common to fork and modify a deployment method repository. OpenStack community encourages merging any modifications to the upstream project and avoiding the use of long-term forks. This unifies practices followed in the OpenStack community and helps to develop the deployment methods.

	TripleO	OS Helm	OS Ansible
Provisioning	OpenStack	Kubernetes	Bare-metal
Configuration	Heat+Puppet	Helm	Ansible
Runtime	Nova/Ironic	Docker	LXC/Bare
	Kolla Ansible	**OS Charms**	**OS Puppet**
Provisioning	Docker	MAAS/Cloud	Bare-metal
Configuration	Ansible	Juju	Puppet
Runtime	Docker	LXC/Bare	Bare

Table 3.1: Software dependencies used for the different parts of the deployment methods

3.1.1 TripleO

TripleO [42] is a method for deploying OpenStack by using another OpenStack cloud for infrastructure provisioning and configuration. OpenStack instance that is used for deployment by cloud administrators is called 'undercloud', and the instance being deployed for the end-users is called 'overcloud'.

Even though TripleO is largely based on OpenStack itself and no other software configuration tools is needed, it has a steap learning curve due to the complexity of resulting in two OpenStack deployments. On the other hand, TripleO method benefits from prior knowledge of OpenStack, and it provides more experience in using OpenStack services while designing and deploying overcloud.

After first installing OpenStack either manually or by using some of the other automated deployment methods, TripleO can be used to deploy another OpenStack instance by using OpenStack services to provision server machines. For a performant deployment, Ironic bare-metal service is recommended instead of Nova. This way the runtime environment of the overcloud is not using a virtual machine.

TripleO uses Heat orchestration service to deploy overcloud as a stack. After initial deployment Heat can be used to update overcloud by adding or removing nodes, provided that undercloud has appropriate resources available. Nodes are provisioned with Nova compute service with assistance of Ironic. Glance is used for node machine images and Neutron for network provisioning.

Before deploying overcloud, undercloud must be prepared. In addition to installing used

OpenStack services, there are some preparation tasks for the undercloud. Images used by the overcloud nodes must be added to Glance. When using Ironic for bare-metal provisioning, available nodes have to be registered to the service. In a production-grade overcloud deployment, machine flavors must also match hardware on target nodes.

Overcloud nodes are split into *roles* that specify used image, flavor, number of nodes with role, and Heat templates used for node configuration. In addition to number of nodes with different roles, deployer can customize overcloud OpenStack service configurations, network configuration and Ceph storage cluster configuration. Otherwise TripleO deployment specifies much of the overcloud features, as deployment is executed by OpenStack services running on the undercloud.

3.1.2 OpenStack Helm

OpenStack Helm [29] is a collection of Helm Charts for deploying OpenStack services into an existing Kubernetes cluster. Kubernetes includes functionality for deploying, scaling, and upgrading OpenStack services running in Docker containers while Helm charts describe components to Kubernetes. Tradeoff with OpenStack Helm is the level of overhead for managing a Kubernetes cluster for running Docker containers. This adds a potential for ossifying Kubernetes into OpenStack environment.

Runtime environment, when using OpenStack Helms, depends on the Kubernetes deployment. Kubernetes can be hosted by another cloud, but for production environments, bare-metal installation of Kubernetes is recommended. For setting up a production-grade Kubernetes installation for OpenStack Helm, tools such as Kubeadm or Airship [2] are recommended. However confguration of the Kubernetes cluster is outside the scope of OpenStack Helm project [29].

OpenStack Helm includes Helm charts for deploying OpenStack services and the needed infrastructure services. In order to customize service configurations, these charts will have to be modified to suit deployment needs. Especially in production use, charts will likely need to be modified [29]. Doing so requires knowledge of both Helm and configured OpenStack services.

Since Kubernetes platform includes many configuration items and provides RESTful API for users, there are questions from architectural point of view, about the need to operate OpenStack cloud in Kubernetes. For operators familiar with Kubernetes who want to provide OpenStack services, OpenStack Helm could be an appropriate deployment method.

Kubernetes in general makes it easy to deploy changes in applications. It has a number of available methods for application updates, such as rolling updates and canary deployments. Many of these methods are beneficial for applications that change rapidly while being actively used. OpenStack services need to be updated twice a year at most, and in general they do not require zero-downtime, as infrastructure provisioning for the most part can wait for hours as long as the existing resources stay functional.

3.1.3 OpenStack Ansible

OpenStack Ansible [27] uses Ansible roles created by OpenStack community for deploying OpenStack. It is a low level deployment method and requires good understanding of target environment and Ansible. OpenStack Ansible was largely developed by Rackspace in the year 2014, but has received community contributions later as well. While it is a relatively old deployment method, it has had a consistent user base.

OpenStack Ansible repository includes an all-in-one deployment method for a proof-of-concept installation on a single host. This method is not intended for production as any further adjustments, such as horizontable scaling would require drastic changes to the infrastructure.

OpenStack Ansible does not include high level server provisioning as it is intended to be used for pre-existing server infrastructure. This especially makes adaptation of the method difficult as the infrastructure has to be provided separately. As opposed to running services in Docker containers, like with OpenStack Helm or Kolla Ansible, services can be installed into Linux containers [20]. Some OpenStack service processes, however are run as native SystemD services for performance optimization.

OpenStack Ansible configuration enables deployed applications to be split to a number of different node groups. This model supports various different OpenStack architectures but determines some of the applications that will be running on the same nodes. For the most part the divisions are sensible for instance having infrastructure services in the same node group.

3.1.4 Kolla Ansible

Kolla Ansible [19] method uses Ansible to deploy OpenStack services into Docker containers built with Kolla [18]. Kolla is an official OpenStack project for building production-

ready Docker containers for OpenStack services. Even though Kolla project is not dependent of Ansible, Kolla Ansible is the primary deployment method used for containers built with Kolla. OpenStack services will be deployed in Docker containers without orchestration tool such as Kubernetes. While avoiding the need to maintain a Kubernetes cluster, managing Docker containers manually results in more low-level administrative tasks related to scaling and updating the deployment.

Kolla Ansible is a popular deployment method for developers getting introduced to OpenStack as it provides a simple all-in-one setup for OpenStack. For developers already familiar with Docker, Kolla Ansible is relatively easy to get into. Another benefit of using Kolla Ansible project as opposed to deploying OpenStack with Ansible to LXC containers, is to leverage the functionality of Docker Engine for virtual resource provisioning. Docker provides simple commands for provisioning networks and volumes in a host-agnostic way. With native virtualization tools, device provisioning is typically done by different programs, such as LVM for volumes and bridgeutils for virtual network interfaces.

With a low-level containerization tool such as LXC, virtual networks for containers have to be created manually. This includes creating veth pairs, and bridging appropriate interfaces to wanted physical network interfaces. These operations typically require an operator who is capable with networking. For this reason, Docker provides simple commands for setting up virtual networks, and a variety of different network types. Even overlay networks accross multiple host operating systems can be setup with Docker. In addition to provisioning virtual devices, Docker provides other commands for operating on them in a simple fashion. Devices are easy to attach or detach from containers. Simplicity of use and reliability of operations is one of the reasons for Docker popularity.

According to recent OpenStack user surveys container-based deployment methods have recenlty become popular among OpenStack community. This follows the trend of containers becoming generally popular in software development and administration. Having container based method for environment setup will likely bring more potential developers to OpenStack community.

3.1.5 OpenStack Charms

OpenStack Charms [28] method uses Juju for deployment of OpenStack. Due to OpenStack Charms deployment method being largely maintained by Canonical, it has a relatively large support from Canonical but at the same time it has a potential for vendor

lock-in. OpenStack Charms only supports Ubuntu target hosts. On the other hand, according to OpenStack user surveys, Ubuntu is the most commonly used host distro in OpenStack deployments. Likewise Juju is a popular method for both developers getting introduced to OpenStack and OpenStack administrators. OpenStack Charms is a versatile deployment method that is applicable for both OpenStack development and production use. It supports a variety of existing cloud providers. Once developer or operator is familiar Juju, it can be used for easy setup of a number of other services as well.

Canonical recommends MAAS to be used for datacenter provisioning in a production OpenStack deployment. A large part of the Juju functionality comes from cloud controllers, manual provisioning is not recommended even though it is supported with Juju as well. MAAS provides lightweight bare-metal server provisioning has a good support for use with Juju. OpenStack Charms is a deployment method that includes a lot of software developed by Canonical. It is a good example of a set of software solutions from a single vendor that provide a good support for each other. Canonical also provides consulting and exerice for custom OpenStack cloud builds. OpenStack marketplace includes also tailored OpenStack software solutions with decent support. One reason for using Canonical's support for building OpenStack cloud with OpenStack Charms could be to get an open-source installation of OpenStack on-premises and get mentoring for data center administrators in the process.

From a techincal perspective one of the reasons why developers may like Juju and as a result OpenStack Charms deployment method, is the fact that the configuration operations are defined with application modules. Even though many of the configuration management tools and OpenStack deployment methods use application modules for low-level tasks, they often times are configured in a declarative language such as YAML. While this makes configurations understandable at high level, the concrete configuration tasks are hidden in the application modules that often times are included in separate source repositories.

3.1.6 Puppet OpenStack

Puppet OpenStack project houses Puppet [30] modules for deploying OpenStack services. Additionally it includes scipts to setup deployments including an all-in-one proof of concept setup. In order to use Puppet OpenStack, Puppet needs to be installed in data center. Puppet OpenStack is largely used for development environment setups, and for automated testing of OpenStack services. Additionally it can be used for large-scale deployments of

OpenStack. Puppet OpenStack consists of Puppet modules for each of the OpenStack components. Each of the modules has its own source repository.

4 Questionnaire

A small scale questionnaire was conducted for OpenStack administrators. Questionnaire explored the methods being used, and different aspects of using automation from administrator's point of view. Questionnaire was conducted as an online form that was linked to respondents. Questionnaire is complementary to OpenStack community user surveys, and it provides data to answer research questions of this thesis in particular. Comparing results from the questionnaire to the OpenStack user surveys when appropriate provides means of verifying some of the responses. In addition the questionnaire focuses to a particular area of OpenStack administration, software automation. Questionnaire was formed with a particular respondent set of OpenStack system administrators in mind. However it is universal enough to be used for larger set of respondents, such as developers, as long as they have some use experience of some of the automated OpenStack deployments methods. It could potentially be used for a larger scale respondent set, but we did not do so during the work of this thesis.

Only one of the research questions shown on Figure 1.1 that cannot be directly answered with documentation, reference literature, or practical testing only, is RQ3 about the benefits of software automation. This is largely due to the subjective nature of benefits in general. Any measures provided for evaluating benefits are easily complex, ambiguous or incomplete and could be a topic for research in other fields such as data analysis. While interview research can be questioned for many of the same reasons, combining it to other forms of research can reinforce results and hypothesis aquired from other methods.

4.1 Goals

Questionnaire started with some questions about the background of the respondents displayed on 4.1. As the expected group of respondents was small, more qualitative information regarding the background could be used. At the same time the questions related to the use of automation were formed in a way that they could be used in a larger scale. Country where the respondents' branch office is located was asked as well. There may be regional differences when comparing OpenStack or any IT infrastructure design choices and software stacks. Although national differences may be relatively small for countries close

1. How many years of professional IT experience do you have?
2. Which company is your current employer?
3. Which country is your team / branch office located at?
4. Select one or more roles that best describe your current area of responsibility.
   (a) Product Owner
   (b) Project Manager
   (c) Software Developer
   (d) Software Tester
   (e) System Administrator

Figure 4.1: Questions about the background of the respondent

to one another, continental differences may be more significant. One of the background questions asked the respondent to choose one or more roles seen as the most relevant for her. Even though some of the questions about the use of automation may be the most relevant for administrators, it is possible for people to have a number of responsibilities simultaneously.

Especially when it comes to selecting software tools for a project, administrators may provide recommendations, but generally the choice needs to be approved by project manager. Technical advise received from experts has often a big impact on the decision. Other potential influencers may be vendors of other software that is in use in the organization. It is possible for an organization to drive itself into a vendor lock-in which may determine the automation methods for OpenStack deployment. However finding out such situations is outside of the scope of this questionnaire, partly because it may be considered a subject to non-disclosure in some organizations.

Potential value of the questionnaire as an approach to answering the research questions shown on Figure 1.1 is to gain insight from administrators with experience in using some automated solutions for operating OpenStack. As RQ1 and RQ2 can for the most part be answered by referencing documentation, benefits of the use of automation are best described by people with experience of operating a cloud on a daily basis. After all, automation tools have a big impact on administrators' work. Ideally the tools being used support the work of operating a service in some way. Questions about the details of automation with OpenStack administration are displayed on Figure 4.2. One purpose of

1. Which automation tool / method are you using at the moment?
2. Which year did you start using the aforementioned tool?
3. Have you used other automation tools for OpenStack production environments?
4. Which Linux distribution do you use for hosts?
5. Have you used the underlying tool of your OpenStack automation (E.g. Ansible, Puppet, Chef) for other software deployments / configurations?

Figure 4.2: Questions about the use of automation

the questions was to find out methods being used. Since the development of the different deployment methods is a subject to current trends in the industry, year of adaptation was asked as well; since OpenStack source code was published in the year 2011, its unlikely that any tool would be adapted before that.

Linux distribution used for target hosts was asked as well. Some deployment methods support only certain distros, and according to OpenStack community user surveys [26] Ubuntu is the most popular Linux distribution used by the OpenStack community. There can potentially be a relation between the node operating system distribution and the selected deployment method. Some of the methods support only certain Linux distributions.

Respondents were asked whether they have used the automation tool with other software configurations. That may provide some indirect insight to research question RQ1 in figure 1.1 about the factors affecting the deployment method design. While some deployment methods are only applicable for OpenStack, others are based on tools that are so widely used that they may have had an influence on the decision of the deployment method. On the other hand in some cases the automation tools may be introduced for the administrator with OpenStack deployment, and afterwards it will used for other software configurations as well.

Respondents were given a scale from one to five to grade different aspects of their primarily used deployment method. Lowest meant that the description in question does not match experiences with the used tool and highest grade meant that it does. Questions for grading are displayed on Figure 4.3. Questions essentially cover three aspects of the used method: ease of use, reliability, and completeness. First two aspects were approached from three different lifecycle operations: deployment, administration, and upgrading.

While previous experience with the automation tool certainly affects all of the aspects

1. How easy do you find deployment of OpenStack with your currently used tool?
2. How easy do you find administering OpenStack with your currently used tool?
3. How easy do you find upgrading OpenStack with your currently used tool?
4. How reliable do you find deployment of OpenStack with your currently used tool?
5. How reliable do you find administering OpenStack with your currently used tool?
6. How reliable do you find upgrading OpenStack with your currently used tool?
7. How well does your currently used tool meet your requirements?
8. Feel free to point out other aspects related to automating OpenStack, that may not have been covered in this questionnaire. (optional free text)

Figure 4.3: Automation tool grading

asked, trends in overall gradings may provide some insight into benefits of automated software management in general. Some tools may be more suitable for different lifecycle operations, but for a more granular analysis, large dataset would be required. Questionnaire focused on experiences in using automated solutions for administering OpenStack. While this is sufficient for gaining basic knowledge about tools being used, and how the tools are seen, there are certain aspects that were left outside of the scope of the questionnaire, but may be crucial for designing custom cloud platform. Since scalability is often a requirement for a successful cloud platform, deployment methods have to be able to scale along with the deployment environment without adding complexity.

Another aspect of deployment that was not covered in questionnaire is the specific use purpose of the deployment. Since OpenStack clouds may be used for different purposes, some of which focus on computing, others high throughput, architectures of OpenStack deployment may vary. Variance of architectures reflects to deployment and make some deployment methods more suitable for a specific use purpose.

Since answering to the questionnaire was voluntary and the focus of the questions is in automation tools, there is some potential for bias in responses. People who are insecure about the deployment method of their choice might retain from answering questionnaire, providing a limited view in responses. This is a concern in any questionnaires that can be read before committing to provide response.

With all the potential sources of inaccuracy, when interpreted correcly the questionnaire

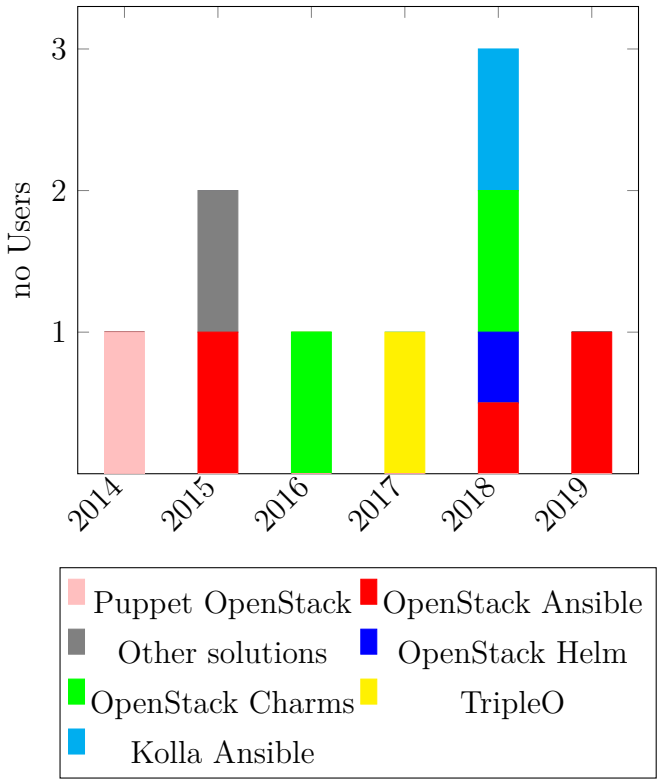

Figure 4.4: Users of different methods as a distribution of adaptation year

can provide more understanding on the general usefulness of the use of automation in OpenStack deployments. Especially with small answer datasets a lot of the responses have to be combined to form a bigger picture. Any interpretations of the results should have some similarities to those of OpenStack user surveys.

4.2 Results

We sent the questionnaire to people from thesis work group contact network and received altogether nine answers. Amount of answers was small due to questionnare being sent to a relatvely small contact network and due to aswering to it being voluntary. While providing small dataset, these facts make the answer data more reliable on other aspects. It would be possible to repeat the questionnaire to a larger respondent group and reflect those results to the ones from the contact network. However in this thesis any comparisons from the results will be done against OpenStack user surveys.

Most of the respondents worked in academic organizations. Countries where the respondents were located were Denmark, Greece, Hungary, Norway, Sweden, Switzerland, Fin-

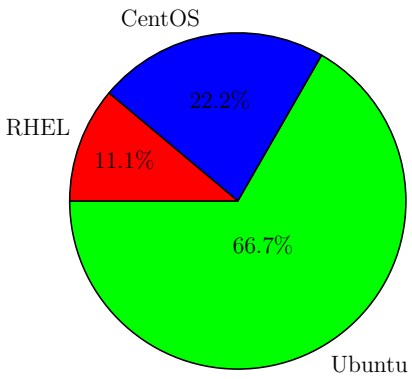

Figure 4.5: Linux distributions used in target hosts

land and the Netherlands. According to OpenStack user surveys, Europe has the third biggest user base of OpenStack, after Asia and North America. Whether this affects the usefulness of automation form the administrator's point of view is doubted.

All of the respondents were fairly experienced system administrators; more than half had over ten years of professional IT working experience, with almost half having twenty or more years of working experience. The experience of the respondents provides some credibility. Earliest year for adapting a method was 2014. At highest the time of using the method has been a little over half of the professional career of the respondent, more likely less than half of the professional career. This would suggest that the respondent is unlikely to be biased.

Almost all of the repondents considered system administrator as their role, on third selecting product owner, and one project manager and software developer. This was to be expected, but it was also insightful to see that so many of the respondents also have responsibilities as product owner. This role may provide them with more authority over decisions regarding selected tools.

Figure 4.4 displays number of users for deployment methods per year of adaptation. OpenStack-Ansible is the most used deployment method with three users, one of which uses it in combination with OpenStack-Helm. OpenStack-Charms is the second most used method with two users. Year of adaptation does not indicate any specific trends with these two methods.

For other deployment methods, there is some correlation with the trends in the community. Puppet-OpenStack, which is an early method for OpenStack deployment, was adapted early, while container based methods, Kolla Ansible and OpenStack Helm have been adapted more recently.

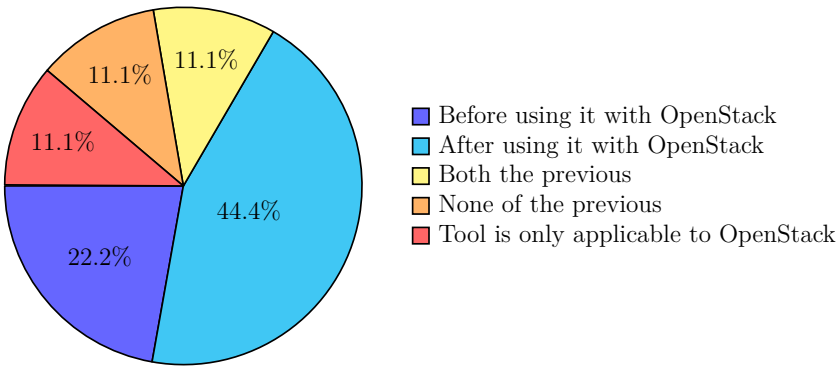

Figure 4.6: Use of tool with other software configurations

Majority of the respondents have not used other deployment methods for production environments. While they possibly have used other methods for other use purposes, such as proofs-of-concept, this indicates that the deployment method chosen is not changed during operation.

Linux distributions used for target hosts are shown on Figure 4.5. Ubuntu which has been the most widely used host operating system according to OpenStack community user surveys, is used by more than half of the respondents to this questionnaire as well.

Figure 4.6 displays correlations of the used method to other software configurations. Only a minority does not or cannot use the underlying automation tool for other software configurations. This result suggests that automated deployment of OpenStack does not differ crucially from other system deployments. Almost half of the respondents were introduced to the automation tool by using it with OpenStack, but have since started to use it for other software configurations.

Results from grading the deployment methods are displayed on Figure 4.7. Grade distributions for ease of use and reliability are averages of distributions of questions 1-3 and 4-6 on Figure 4.3 correspondingly.

Ease of use is graded slightly above medium in average. Overall the distribution is somewhat even showing no obvious trends. Correlation between previous experience with the tool and professional experience in overall cannot be analysed based on this testing due to the small data gathered. Previous experience with software automation might make any deployment methods feel easy which could affect this grading as well.

Reliability of the used automation tools is graded well, with highest frequency grades being five and four. This is an important result as it points out that reliability is one of the benefits of the use of automation.

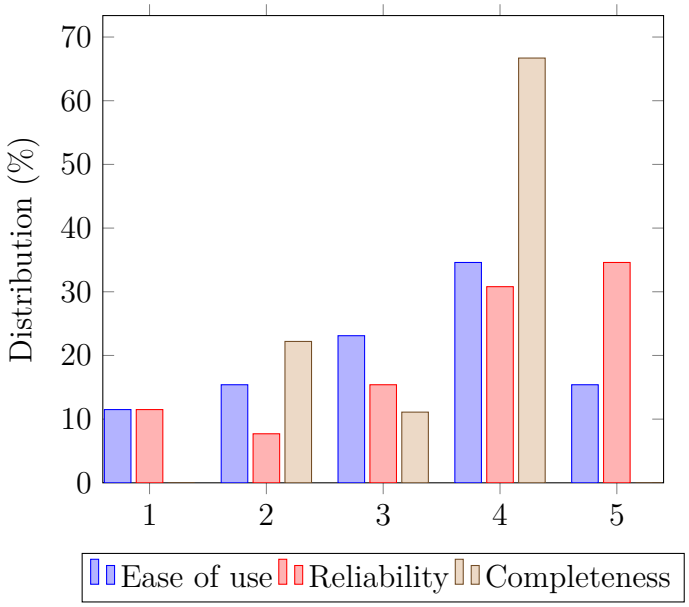

Figure 4.7: Grading of deployment method

Grading of completeness has more variance than the two other graded features. One technical reason for this can be the fact that completeness had only one graded scale, while distributions of usability and reliability are averaged over three scales for different lifecycle operations. Regardless, the fact that there were no highest or lowest grades given for completeness is unique for this feature. Additionally, medium grade was the least frequent of given grades, making the grade distribution of completeness different from other features as well. Most frequent grading for completeness with biggest difference from the next most frequent grade is four, providing a generally positive evaluation.

Last optional question about any comments about the use of automation received a few responses. One of the reponses pointed out how vendors can affect the selection of tools, especially for smaller organizations; not relying on big vendors like RedHat or Canonical for small organizations could even provide a strategic risk. Another free text commented how easy adaptation of the tool is important, but that the method should be customizable for later development.

4.3 Hypothesis

Results from the questionnaire provide some insight to answer research question three in Figure 1.1 regarding the benefits of automated deployment methods. While a larger scale questionnaire might represent the community with more detail and provide data for more

fine-grained analysis, some trends can be seen from the received answers.

Even with a small set of respondents there are many different deployment methods in use. Since the questionnaire focuses on the use of automated solutions, it cannot be used for comparison with the use of manual deployments. However the fact that such a wide variety of different methods are being used suggests that there is no de facto solution for automating OpenStack deployment.

The trends in the adaptation of different methods correlates to trends in the OpenStack community. This is to be expected as the nature of OpenStack is community-driven development. On top of that it is typical the the adapted methods take influence from regional and organizational factors.

The fact that only a minority of respondents have not used the same automation tool for other software configurations indicates that usefulness of software configuration management often goes beyond deployment of a single software, such as OpenStack. A big part of the respondents started using the tool after being introduced to it by deploying OpenStack.

For the most part, the use of current automation methods is graded well in terms of usability. Even though easiness did not get the highest grading for the most, it got an overall positive grading. While usability is not generally a priority on a tool used by professionals, many of the commercial software products invest in it as well. For instance Puppet Enterprice provides a browser based user interface, as does OpenStack Horizon.

Reliability is the most commonly seen benefit of the deployment methods according to gradings in this questionnaire. While gradings of reliablity have some variance, it distributes strongly towards positive grades with the highest grade being the most frequent.

Used methods are graded well in terms of completeness, however it distributes more unevenly than other features. This might indicate that there are some varying opinions regarding how suitable different methods are for the current needs of the deployment. It is a fact that automated deployment methods cannot reach a higher level of configurability than manual configurations. However the relatively positive grading in terms of completeness suggests that for many users, the level of configurability with the used automation tool is sufficient for the current needs.

In summary, conducted questionnaire suggests that mainly system administrators benefit from the development of automated deployment methods. Use of automation provides reliability for administrative tasks, while being easy to use and sufficient for requirements.

5 Evaluation

OpenStack-Ansible deployment method was tested in a virtualized environment. Method was picked due to its popularity in the results of questionnaire shown on Chapter 4 and its low-level nature. Scope of evaluation is to explore the tasks needed for automated deployment of an operational production-grade OpenStack cloud.

Goal of the evaluation is to provide more insight to research questions shown on Figure 1.1 from one deployment method's point of view. Instead of relying merely on reference literature, and available documentation, deployment provides experience on low-level aspects to support other sources. According to OpenStack user survey 2018 [26], documentation of OpenStack is seen as one of the areas requiring improvements while at the same time it is seen as one of the most appreciated aspects of OpenStack.

Deployment was kept as simple as possible while also being technically sufficient for a production-grade environment. OpenStack production environments differ from development environments much the same than any in any software applications: production logging is less verbose and software running in production mode is more optimized by omitting unnecessary functionality.

Many OpenStack development environment deployments utilize only a single machine. With performant and reliable cloud deployment, services will have to be distributed among multiple target machines, and deployment host containing secrets such as authorized SSH keys and internal service credentials, has to be separated from runtime enviroments for security reasons. Having separate machine for deployment assets, makes it possible to have network-level security policies. For instance SSH access to hosts running OpenStack services can be allowed from the deployment hosts only.

Since production architectures are designed based on specific needs, deployment is kept as simple as possible, with the possibility of extending functionality according to custom needs. Thus there are only few requirements of the cloud functionality. Basically it should be possible to create virtual networks and virtual machines.

5.1 Setup

Setup used for test deployment simulates physical dataset topology depicted in Figure 5.1. Topology is based on deployment guide of OpenStack-Ansible [27]. By using virtual local area networks, three separate physical switches as shown on the diagram would not be needed, but instead a topology such as FatTree or Clos could be used in a data center. Separate physical network devices do however ensure that congestion in one network does not propagate to the other. It is also recommended to use multiple physical network interfaces connected to separate switches and bond them into the same bridge for redundancy.

In the simplest multi-node deployment setup, two hosts are used. *Deployment host* includes all of the assets needed for deployment, including OpenStack-Ansible repository, Ansible roles used in deployment, installation of Ansible, deployment configuration files, and secrets. *Target host* includes all of the OpenStack services and necessary infrastructure services. When scaling deployment horizontally, instead of having a single target host for all of the deployed software, setup could have distinct hosts dedicated to compute services, storage services, and infrastructure services.

Management network is the only network routed to the Internet. It is mainly used for management access to target host, and for egress internet traffic. All of the hosts used in deployment will have to be connected to this network. *Tunnel network* is used for guest

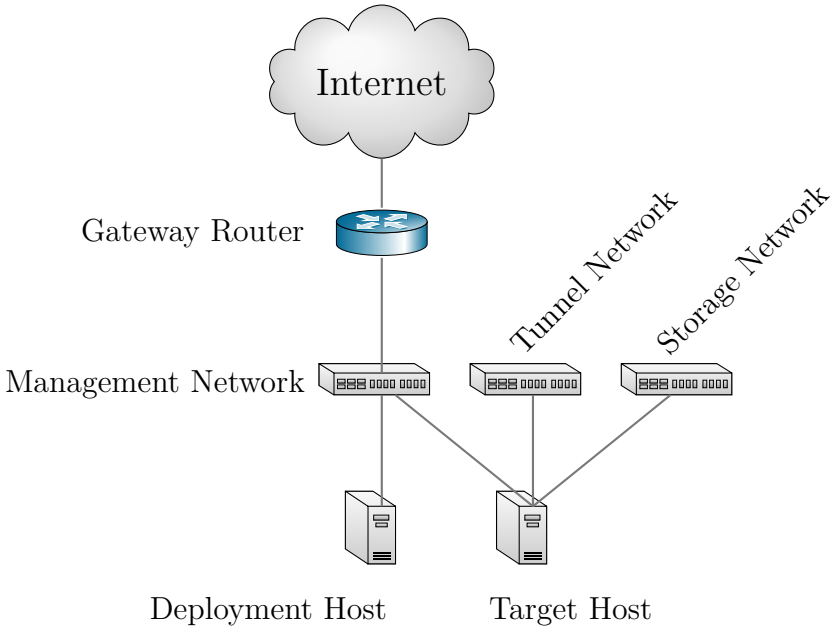

Figure 5.1: Test Setup Network Topology

network traffic. Compute and network nodes will have to be connected to it. *Storage network* is used for network volume traffic. Volume nodes are connected to it.

OpenStack cloud environment, called cPouta, provided by CSC Center for Science Ltd. was used for virtualized test setup. While using a cloud service provides benefits such as automated resource provisioning and configuration, it also adds room for error. There is more components to troubleshoot when experiencing issues. There may also be known issues in the cloud service that affect tasks executed during evaluation. The same applies to any used software, including operating systems. The benefits of using a cloud platform include fast and automated infrastructure provisioning which makes testing more easily repeatable.

Using virtualized environment as an infrastructure for testing differs from physical data center in some aspects. Even though hypervisors can be run within virtual machines and overlay networks can be constructed on top of existing overlay networks, there are some details that have to taken into consideration when deploying a cloud inside a cloud.

Since infrastructure provisioning is outside of the scope of OpenStack-Ansible, host machines and virtual networks were provisioned with custom Heat stack templates (see Appendix A). Infrastructure provisioning from OpenStack cloud is conceptually similiar to how TripleO uses undercloud or how Juju uses cloud providers.

Test environment was provisioned by using Heat orchestration tool. Heat provisions servers for hosts and networks used in deployment. Additionally some basic configurations were applied with cloud-init. Host configurations executed with Heat make prerequisite system preparations shown on Figure 5.2. Nameserver specified on the target host *resolv.conf* file had to be changed in order to provide functioning dns resolution.

Assumption with OpenStack-Ansible is that the subnets belong to separate VLANS. With the test setup running in OpenStack, VXLAN segregation was used instead. Difference with this approach is that, additional VLAN tagging is not employed on guest hosts, but instead network interfaces are created by OpenStack Neutron, and they are bridged without additional package tagging.

1. External network bridges are configured for nodes.
2. SSH keys used in deployment are injected to target machines.

Figure 5.2: Minimal prerequisites for OpenStack-Ansible multi-node deployment

Ubuntu Bionic (18.04) was used for host machines. Since from release 17, Ubuntu uses Netplan [24] as the default network interface renderer. In order to use legacy NetworkD service without additional setup programs, Netplan is disabled and masked. Network bridges are then defined with NetworkD configuration files.

5.2 Deployment

Once deployment host is prepared, OpenStack-Ansible repository includes shell script for bootstrapping OpenStack Ansible deployment. Script installs Ansible and the roles used in deployment. It hooks configuration files to be used in OpenStack Ansible execution. Once OpenStack Ansible is prepared, configuration files have to be installed on the deployment server, which was done with a custom Ansible playbook. OpenStack-Ansible uses three configuration files for describing the deployment. See Appendix B for preparation tasks automated with Ansible as well as configuration files for the deployment.

openstack_user_config.yml file describes which hosts are used for particular services, and IP addressing. It maps different node types to hostnames or IP addresses. Same IP can be used for multiple different node types, and when OpenStack is deployed to a single host, all of the node groups point to the same IP.

CIDRs for each of the configured networks are listed in *openstack_user_config.yml* file. Container network interfaces are defined as well. Type of the mechanism used for plugging container interfaces to appropriate network bridges can be specified in configuration. Virtual ethernet pairs were used in the test deployment as it is the most common mechanism according to OpenStack-Ansible documentation [27].

Most of the configurations specified in *openstack_user_config.yml* file do not need to be specified with some of the other deployment methods such as Kolla Ansible or OpenStack Charms. More high-level deployment methods can provide a default configuration for network bridges and container interfaces. These bridges could also be configured automatically for a proof-of-concept installation with OpenStack-Ansible. Reason for low-level network configuration is likely the fact that OpenStack-Ansible is used for data center deployments with varying topologies. While providing more configurability, this method has potential for error due to environment setup not matching the configuration specified for OpenStack-Ansible.

user_variables.yml defines parameters used by Ansible roles in deployment. Many of the

specified parameters describe configuration parameters used for the installed programs, but some specify installation features.

In order to get the deployment functioning, HTTPS protocol had to be specified for public, internal, and administrative api endpoints, and TLS verification had to be set off. Without these specifications some of the verification tasks failed due to Ansible trying to use HTTP requests for HTTPS endpoints, or due to warnings about untrusted self-signed TLS certifiates.

Default installation method is to used source code and install programs with pip into virtual enviroments. While this method takes longer than using prepared distribution packages it is the most reliable method for test deployments, since it guarantees that appropriate versions are being used for OpenStack services. New OpenStack releases are not always published to public repositories for package managers such as apt or yum. For long term operations it would be possible to create custom distribution packages for new OpenStack releases.

While providing a minimal required configuration for a successful deployment, in a production level deployment TLS verification would not have to be turned off, since the deployer would likely provide certificates signed by a trusted authority. There would likely be other configuration variables that would be specified according to custom needs. However a simple deployment can be run successfully with five specified configuration parameters in *user_variables.yml* file.

Third configuration file, *user_secrets.yml* contains secrets used in deployment, including credentials to various services used by administrators and other deployed service users. OpenStack-Ansible repository includes a Python script for generating random secrets. While having all of the deployment secrets in a single file makes it easier to leak all of the passwords at the same time, it has benefits as well. For deployer it provides easy access to all of the deployed services without having to separately read a distinct file for service to be accessed. Proper secret management is not part of the scope of OpenStack-Ansible, and other solutions for it can be used as needed.

In case some of the secrets would leak, generating new ones with the script would be fast. However installation playbook is not idempotent and thus using it as a methods for disaster recovery would require the entire OpenStack to be re-installed. Depending on how this is done it could have a big impact on the service operation.

After preparation, OpenStack can be installed according to configurations by running a

single Ansible playbook. Installation consists of three phases and in test setup it took approximately two hours when installing from source code. First Ansible prepares target hosts. During this phase LXC containers for each of the services are initialized and plugged into networks. During the second phase the infrastructure services required by OpenStack services are installed and configured. Finally OpenStack services themselves are installed according to configuration specifications.

5.3 Verification

As with any software, verification of an operational OpenStack deployment depends on the expected functionality. With any requirements however, there are some indications of a failed deployment, such as SystemD services in failed state or network connectivity issues. After running Ansible playbooks successfully, these features were checked with operating system functionality and network diagnostic tools. Many of the basic issues such as problems with connectivity, will likely result in failed playbook execution.

Approaches to verifying software configurations can be highly analytical, and based on static checks. Rehearsal [37] is a software developed for verifying system configurations applied with puppet. While similiar tools can be created for other software configuration management tools, this thesis focuses on core functionality which is simple enough to verify manually.

OpenStack-Ansible includes tasks for both setting up some basic resources such as provider networks and machine images, as well as verifying functional cloud by using automated tests. These features can be enables with configuration parameters. These tasks were not used in evaluation in order to focus on the core deployment functionality. In order to provide transparent testing of the deployment, running the basic verification operations manually was seen sufficient. Once the deployment is confirmed functional these tasks could be used for further cloud deployments.

The initial deployment is difficult to keep as small as adding a single service at a time. It entails a lot of tasks that depend on one another. For this reason it is kept as simple as possible while enabling expansion as potential future work.

For verifying operational deployment, tasks shown on Figure 5.3 were executed. Tasks verify the functionality of most essential OpenStack services, Keystone, Glance, Neutron, and Nova. More services can be added, once the basic services are confirmed operational.

1. Upload CirrOS image to Glance.
2. Create private Neutron network and subnet.
3. Enable ICMP ingress security group rule in default security group.
4. Create test flavor for virtual machines.
5. Create two Nova virtual servers in private network using CirrOS image.
6. Open console session with virsh to one of the created Nova servers.
7. Confirm network access between the two deployed servers by pinging one from the other.

Figure 5.3: Verification tasks for deployment

CirrOS image used for virtual machine testing is a machine image commonly used for verifying OpenStack deployment. Its raw image file was downloaded from the official source and provided for Glance API in image creation request.

Once machine image was uploaded successfully, tenant network was created for virtual machines. With tenant network created, host flavor for virtual machines had to be created. With an image, network and a flavor guest hosts could be created to verify that Nova API works properly. When plugging guest hosts to the same tenant network, connectivity can be verified by using tenant network private ip addresses. For ICMP network diagnostics ingress firewall rules have to be set up accordingly.

While functionality verified for the test setup contains only a subset of necessary features of an operational OpenStack deployment, it was sufficent to confirm that the deployment matched the specified configuration. Some of the deployed features were not verified as they were not seen necessary for providing answers to research questions. Verification performed during the evaluation is sufficient to bring out features offered by OpenStack-Ansible deployment method as well as its benefits and tradeoffs. Any further enhancements to the deploymet could be left as future work.

5.4 Hypothesis

Designing and deploying a testbed for OpenStack-Ansible provided clarified how automated deployments for OpenStack are prepared and executed. Not many manual admin-

istrative tasks were required during deployment. However due to Ansible's verbose logging and descriptive task naming, it was easy to see which tasks were executed during deployment. Tasks describe what configuration items are involved in deployment, and what kind of configurations are applied.

To answer research question RQ1 in Figure 1.1, one of the key factor affecting the design of OpenStack-Ansible is the ability to provide automated deployment of OpenStack on top of a pre-existing server infrastructure. This choice provides configurability at the cost of having to carry the responsibility of low-level setup of the infrastructure. In comparison to some of the other deployment methods, OpenStack-Ansible architecture has to be designed in detail before the deployment. The deployment configuration has correspond to the existing infrastructure. This design is fundamental to OpenStack-Ansible which provides a batteries-included deployment method, as stated in its manifest [27].

Getting started with using OpenStack-Ansible for a multi-node deployment was not as easy as using a development environment setup, such as Packstack. Getting the deployment working required multiple trials and some troubleshooting. Installing OpenStack services manually might be more easy in the beginning. However with automation tools once the issues were resolved the deployment could be repeated easily. This applied also to infrastructure provisioning with Heat. Even though it was not tested, automated deployment is likely to scale better than manual installation.

Ideally when a change in the configuration of the production environment would have to be made, the configuration could be properly tested in a staging environment and the be run in production environment avoiding operational errors. This approach requires a testbed that is structurally similiar to the production environment. Since hosts can be accurately emulated by using hypervisors, network setup used in evaluation was the most different from a datacenter network, since VLAN provisioning was not available in the used OpenStack platform.

Due to OpenStack deployment test setups never totally matching production environments, there is always room to question the benefit of automated deployment methods as experienced in a test. However based on the experiences during evaluation, use of software automation shows a lot of potential for deployment of OpenStack production environment. Reliability which was seen as one of the strengths of automated deployment methods in questionnaire on Chapter 4 could be confirmed as it was to reproduce once successfully executed deployment.

Features listed above benefit primarily system administrators, as they would be the ones

running the configurations manually otherwise. In case of issues, the administrators would still have do the troubleshooting. If the deployment is configured correctly, administrators do not have to worry about applying the configuration to the targeted system. While repeating the same configuration tasks creates a routine for the administrator, when updates are applied to the system, some of the confiuration tasks may be new and require some prior testing.

In addition to administrators benefiting from the development of automation tools, developers benefit from it as well. Software developers have a tendency to focus on application development and not environment setups. Due to Ansible playbooks describing environment setup tasks concretely, they function as a type of documentation. This could be seen during evaluation of OpenStack Ansible when at one point a role was resulting in failure. The role's tasks described in YAML format, provided detailed description of the task that was run. Combining this description to the error log enabled the root issue to be revealed and fixed. This process brought out some details of the overall deployment.

The evaluation of OpenStack Ansible provided an introduction to automated deployment of OpenStack. Not using an all-in-one deployment method but instead configuring an optimized deployment required understanding of both OpenStack services, underlying infrastructure services as well as networking concepts related to both the test environment and LXC containers. Evaluation brought out some details that were not found in any documentation. It is common in a large software stack for some details to change constantly. Keeping documentation up to date with these changes is difficult. For this reason it is often stated in the documentation to consult the documentation of any software dependencies directly instead.

OpenStack-Ansible provided very little features as a deployment method. Following the design principle of Ansible, OpenStack-Ansible included no other configuration items that deployment host, which had to be prepared by the deployer. Most of the work during the evaluation was related the preparation of the deployment and not the configuration of the actual deployment. This is characteristic to OpenStack-Ansible. It is agnostic of the underlying infrastructure and focuses on automating the configuration operations only.

Once deployment was configured properly, its execution succeeded without exception. Reliability from administrator's point of view, which was one of the key findings of questionnaire in Chapter 4 could thus be confirmed during this evaluation. Other benefits of configuration management tool included easy execution. Configuration file sources made it easier to learn how OpenStack is deployed.

6 Summary

This thesis has presented contemporary methods for automating configuration tasks for a cloud platform deployment. Different designs, as well as tools and components included in deployment methods have been introduced. Other differences between methods have been explored as well. Lastly, some of the benefits of the development of automated tools have been presented and reasoned. Answers to research questions shown on Figure 1.1 are based on reference publications, technical documentation, a small scale questionnaire, as well as a deployment test conducted in a virtualized environment. There have not been conflicting results gained from different sources although some of the methods focus on different aspects of the research. Topic of this thesis is motivated by contemporary practices in IT infrastructure management utilizing cloud computing and popularity of automating system configuration tasks. These trends have emerged from years of providing large scale internet-based services. Recent developments in virtualization technologies have helped to automate different administrative tasks. At the same time organizations adapting agile philosophies for IT service management has created need to increase the level of automation in service management.

Cloud computing has provided scalable infrastructure for services provided over the internet. Cloud platforms can provide infrastructure as a service for external organizations or different departments in the same organization. Cloud platforms are administered much the same way than other IT services. In order to be profitable cloud platforms need to be large. Administrative tasks in cloud datacenters can become time-consuming, difficult, and entail a lot of operational risks. OpenStack is a family of open-source software projects for running cloud platform. It has been developed by various organizations and community contributions. Currently OpenStack is the most commonly used software for running private clouds. It is being used by big organizations in telecommunication, retail, finance and scientific research. Architecturally OpenStack bears close resemblance to other popular public cloud platforms AWS, Azure and GCP.

Software configuration management has become a popular solution for administering datacenters. It provides means to create configurations that are applied to the system by a computer program. While there are technically no limits to configurations that can be applied with automated tools, different tools provide varying functionality. Automation

tools have different software architectures that affect how complex configuration tasks are structured.

OpenStack community has created various official deployment methods that utilize software configuration management tools. Deployment methods follow best practices of the underlying automation tools and focus on configuring OpenStack services to available server infrastructure. Some deployment methods include infrastructure provisioning while others are run against pre-existing server infrastructure. Design of different automated deployment methods is driven by their use purposes. Methods may focus on easy adaptation, extendability, configurability or operability with varying emphasis. Some of the deployment methods only include assets for an initial deployment of OpenStack. Further operation and maintenance would then have to be provided with custom solutions or by using other tools available. Other deployment methods include functionality for scaling, updating, or health monitoring the deployment. The latter type of deployment methods usually contains more configuration items, whose operation has to be assured during cloud operations. The former type does often not include any additional software dependencies than tool used for the deployment itself.

There are parties that may benefit from the use of automation tools indirectly. In case automation can be used for providing high quality service in an affordable fashion, both the service providers and end users benefit from it. While hhis benefit may even seem obvious it is difficult to prove, since the actual measurements on savings can be complex and a subject to non-disclosure. Since automated tools are primarily used by system administrators, they are the primary beneficiaries from the development of the tools. Results from questionnaire presented in Chapter 4 suggest that system administrators benefit from reliability when using automated software configuration tools. Reliability was also noticable in evaluation shown on Chapter 5 as succesful execution was easily repeatable.

At the moment it would seem that the use of automated solutions for system administration will keep on gaining popularity. Since use of automation in general is increasing and being applied in areas such as decision making, using it to be able to automate system administration tasks is not totally unexpected. In addition to providing reliability for repeating tasks, it may even create new possibilities for management of large scale IT infrastructures. There are no obvious solutions or de facto software for automating infrastructure management, although some methods are more popular than others. Even though some methods will inevitably become deprecated, automation for infrastructure

management has a number of reasons to provide value for IT administators.

Use of automation in IT service management has been steadily increasing and seems to continue to do so. Many solutions have been developed recently, but there are still many potential directions for future work. As we have pointed out in this paper, automated deployment methods are affected by the deployed software and paradigms for administering the software. Research of new virtualization technologies and network protocols may have a big impact on the deployment methods. They can potentially obsolete crucial features of some of the deployment methods that we have presented. However as we also noticed some of the benefits of the automated software configuration methods are no restricted to the context where they are being used. Even if there were to be some fundamental changes in the way that IT services are being provided, being able to reduce operative risk by automating administrative tasks can still be found useful. As such, other potential direction for future work could be further development of software automation tools or even study of idempotency in the context of software configurations.

Bibliography

[1] *A secure web is here to stay*. Online; accessed 2020-10-27. URL: https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html.

[2] *Airship*. Online; accessed 2020-10-27. URL: https://www.airshipit.org/.

[3] *Amazon Web Services (AWS)*. Online; accessed 2020-10-27. URL: https://aws.amazon.com/.

[4] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. "Software Engineering for Machine Learning: A Case Study". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 291–300.

[5] *Ansible*. Online; accessed 2020-10-27. URL: https://www.ansible.com/.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[8] T. Bell, B. Bompastor, S. Bukowiec, and M. Fermin Lobo. "Scaling the CERN OpenStack cloud". In: *Journal of Physics: Conference Series*. 2015.

[9] D. Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, omega, and kubernetes". In: *Queue* 14.1 (2016), pp. 70–93.

[11] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[12] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. "DevOps". In: *Ieee Software* 33.3 (2016), pp. 94–100.

[13] *Eucalyptus*. Online; accessed 2020-10-27. URL: https://www.eucalyptus.cloud/.

[14] *Google Cloud Platform*. Online; accessed 2020-10-27. URL: https://cloud.google.com/.

[15] *JSON*. Online; accessed 2020-10-27. URL: https://www.json.org/.

[16] *Juju Charms*. Online; accessed 2020-10-27. URL: https://jujucharms.com/.

[17] A. Kanso, N. Deixionne, A. Gherbi, and F. F. Moghaddam. "Enhancing openstack fault tolerance for provisioning computing environments". In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE. 2017, pp. 77–83.

[18] *Kolla*. Online; accessed 2020-10-27. URL: https://opendev.org/openstack/kolla/.

[19] *Kolla Ansible*. Online; accessed 2020-10-27. URL: https://opendev.org/openstack/kolla-ansible/.

[20] *Linux Containers*. Online; accessed 2020-10-27. URL: https://linuxcontainers.org/.

[21] *Linux Manuals: CGroups*. Online; accessed 2020-10-27. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[22] *Microsoft Azure*. Online; accessed 2020-10-27. URL: https://azure.microsoft.com/.

[23] P. Miller and L. E. Nelson. "Brief: OpenStack Is Now Ready For Business". In: *Forrester Research* (2015).

[24] *Netplan*. Online; accessed 2020-10-27. URL: https://netplan.io/.

[25] *OpenNebula*. Online; accessed 2020-10-27. URL: https://opennebula.io/.

[26] *OpenStack 2018 User Survey Report*. Online; accessed 2020-10-27. URL: https://www.openstack.org/user-survey/2018-user-survey-report.

[27] *OpenStack Ansible*. Online; accessed 2020-10-27. URL: https://opendev.org/openstack/openstack-ansible/.

[28] *OpenStack Charms Deployment Guide*. Online; accessed 2020-10-27. URL: https://opendev.org/openstack/charm-deployment-guide/.

[29] *OpenStack Helm*. Online; accessed 2020-10-27. URL: https://docs.openstack.org/openstack-helm/latest/.

[30] *OpenStack Puppet Deployment Guide*. Online; accessed 2020-10-27. URL: https://docs.openstack.org/puppet-openstack-guide/latest/.

[31] *Podman*. Online; accessed 2020-10-27. URL: https://podman.io/.

[32] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran. "What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling". In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.2 (June 2019). DOI: 10.1145/3341617.3326142. URL: https://doi.org/10.1145/3341617.3326142.

[33] *Project Atomic*. Online; accessed 2020-10-27. URL: https://www.projectatomic.io/.

[34] *Puppet*. Online; accessed 2020-10-27. URL: https://github.com/puppetlabs/puppet/.

[35] K. Saarelainen and M. Jäntti. "Quality and human errors in IT service infrastructures—Human error based root causes of incidents and their categorization". In: *2015 11th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2015, pp. 207–212.

[36] O. Sefraoui, M. Aissaoui, and M. Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[37] R. Shambaugh, A. Weiss, and A. Guha. "Rehearsal: A configuration verification tool for puppet". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 416–430.

[38] L. Shwartz, D. Rosu, D. Loewenstern, M. J. Buco, S. Guo, R. Lavrado, M. Gupta, P. De, V. Madduri, and J. K. Singh. "Quality of IT service delivery—Analysis and framework for human error prevention". In: *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2010, pp. 1–8.

[39] N. K. Singh, S. Thakur, H. Chaurasiya, and H. Nagdev. "Automated provisioning of application in IAAS cloud using Ansible configuration management". In: *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*. IEEE. 2015, pp. 81–85.

[40] A. Sirbu, C. Pop, and F. Pop. "MaaS advanced provisioning and reservation system". In: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. 2015, pp. 13–18.

[41] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 275–287. ISBN: 9781595936363. DOI: 10.1145/1272996.1273025. URL: https://doi.org/10.1145/1272996.1273025.

[42] *TripleO*. Online; accessed 2020-10-27. URL: https://docs.openstack.org/tripleo-docs/latest/.

[43] A. Wiedemann, N. Forsgren, M. Wiesche, H. Gewald, and H. Krcmar. "The DevOps Phenomenon". In: *Queue* 17.2 (2019), pp. 93–112.

[44] *YAML*. Online; accessed 2020-10-27. URL: https://yaml.org/.

[45] J. L. Yingching Cheng Malini Bhandaru. *Analysing and Tuning China Mobile's OpenStack Production Cloud*. Tech. rep. Online; accessed 2020-10-27. China Mobile, Intel, 2016. URL: https://01.org/sites/default/files/performance_analysis_and_tuning_in_china_mobiles_openstack_production_cloud_2.pdf.

Appendix A Heat templates used for test environment setup

```
heat_template_version: 2016-10-14
description: |
    Heat stack for OpenStack-Ansible deployment test setup.
parameters:
    public_network:
        type: string
    mgmt_cidr:
        type: string
    mgmt_gateway_ip:
        type: string
    dns_nameservers:
        type: comma_delimited_list
    tunnel_cidr:
        type: string
    tunnel_gateway_ip:
        type: string
    storage_cidr:
        type: string
    storage_gateway_ip:
        type: string
    timezone:
        type: string
    deploy_flavor:
        type: string
    target_flavor:
        type: string
    deploy_image:
        type: string
    target_image:
        type: string
    deployment_private_key:
        type: string
    deployment_public_key:
        type: string
    deployment_authorized_key:
```

```
        type: string
    floatingip:
        type: string
resources:
    mgmt_network:
        type: ./common-heat-templates/network/access-net.yaml
        properties:
            public_net: { get_param: public_network }
            cidr: { get_param: mgmt_cidr }
            gateway_ip: { get_param: mgmt_gateway_ip }
            dns_nameservers: { get_param: dns_nameservers }
    tunnel_network:
        type: ./common-heat-templates/network/secondary-net.yaml
        properties:
            cidr: { get_param: tunnel_cidr }
            gateway_ip: { get_param: tunnel_gateway_ip }
    storage_network:
        type: ./common-heat-templates/network/secondary-net.yaml
        properties:
            cidr: { get_param: storage_cidr }
            gateway_ip: { get_param: storage_gateway_ip }
    base_config:
        type: OS::Heat::CloudConfig
        properties:
            cloud_config:
                final_message: |
                    Server configured successfully!
                timezone: { get_param: timezone }
                package_update: true
                package_upgrade: true
                packages:
                  - bridge-utils
                  - ifupdown
                power_state:
                    mode: reboot
                    message: Rebooting
                    timeout: 30
                    condition: True
    target_packages_config:
```

```
75          type: OS::Heat::CloudConfig
76          properties:
77              cloud_config:
78                  packages:
79                    - bridge-utils
80                    - debootstrap
81                    - ifenslave
82                    - ifenslave-2.6
83                    - lsof
84                    - lvm2
85                    - chrony
86                    - openssh-server
87                    - sudo
88                    - tcpdump
89                    - vlan
90                    - python
91                    - ifupdown
92      disable_netplan:
93          type: OS::Heat::SoftwareConfig
94          properties:
95              config: |
96                  #!/bin/sh
97                  sudo systemctl stop systemd-networkd.socket systemd-networkd
                    ↪  networkd-dispatcher systemd-networkd-wait-online
98                  sudo systemctl disable systemd-networkd.socket systemd-networkd
                    ↪  networkd-dispatcher systemd-networkd-wait-online
99                  sudo systemctl mask systemd-networkd.socket systemd-networkd
                    ↪  networkd-dispatcher systemd-networkd-wait-online
100                 sudo apt-get --assume-yes purge nplan netplan.io
101     deploy_inject_root_private_key:
102         type: ./common-heat-templates/config/inject-root-private-key.yaml
103         properties:
104             private_key: { get_param: deployment_private_key }
105     deploy_inject_root_authorized_key:
106         type: ./common-heat-templates/config/inject-root-authorized-key.yaml
107         properties:
108             authorized_key: { get_param: deployment_authorized_key }
109     target_inject_root_authorized_key:
110         type: ./common-heat-templates/config/inject-root-authorized-key.yaml
```

```
111         properties:
112             authorized_key: { get_param: deployment_public_key }
113     mgmt_port_deploy:
114         type: OS::Neutron::Port
115         properties:
116             network: { get_attr: [ mgmt_network, private_net ] }
117             fixed_ips:
118               - subnet: { get_attr: [ mgmt_network, private_subnet ] }
119                 ip_address: 172.29.236.50
120     floating_ip_association:
121         type: OS::Neutron::FloatingIPAssociation
122         properties:
123             floatingip_id: { get_param: floatingip }
124             port_id: { get_resource: mgmt_port_deploy }
125     add_interfaces_deploy:
126         type: OS::Heat::CloudConfig
127         properties:
128             cloud_config:
129                 final_message: |
130                     Added interface files successfully!
131                 write_files:
132                   - path: /etc/network/interfaces
133                     owner: root:root
134                     permissions: '0644'
135                     content: |
136                         auto lo
137                         iface lo inet loopback
138 
139                         auto ens3
140                         iface ens3 inet manual
141 
142                         source /etc/network/interfaces.d/*.cfg
143                   - path: /etc/network/interfaces.d/aio.cfg
144                     owner: root:root
145                     permissions: '0644'
146                     content: |
147                         auto br-mgmt
148                         iface br-mgmt inet dhcp
149                             bridge_stp off
```

```
                            bridge_waitport 0
                            bridge_fd 0
                            bridge_ports ens3
    host_config_deploy:
        type: OS::Heat::MultipartMime
        properties:
            parts:
              - config: { get_resource: add_interfaces_deploy }
              - config: { get_resource: base_config }
              - config: { get_resource: disable_netplan }
              - config: { get_resource: deploy_inject_root_private_key }
              - config: { get_resource: deploy_inject_root_authorized_key }
    osa_host_deploy:
        type: OS::Nova::Server
        properties:
            flavor: { get_param: deploy_flavor }
            image: { get_param: deploy_image }
            networks:
              - port: { get_resource: mgmt_port_deploy }
            user_data_format: RAW
            user_data: { get_resource: host_config_deploy }
    mgmt_port_target:
        type: OS::Neutron::Port
        properties:
            network: { get_attr: [ mgmt_network, private_net ] }
            fixed_ips:
              - subnet: { get_attr: [ mgmt_network, private_subnet ] }
                ip_address: 172.29.236.100
    tunnel_port_target:
        type: OS::Neutron::Port
        properties:
            network: { get_attr: [ tunnel_network, private_net ] }
            fixed_ips:
              - subnet: { get_attr: [ tunnel_network, private_subnet ] }
                ip_address: 172.29.240.100
    storage_port_target:
        type: OS::Neutron::Port
        properties:
            network: { get_attr: [ storage_network, private_net ] }
```

```
189             fixed_ips:
190               - subnet: { get_attr: [ storage_network, private_subnet ] }
191                 ip_address: 172.29.244.100
192     add_interfaces_target:
193         type: OS::Heat::CloudConfig
194         properties:
195             cloud_config:
196                 final_message: |
197                     Added interface files successfully!
198                 write_files:
199                   - path: /etc/network/interfaces
200                     owner: root:root
201                     permissions: '0644'
202                     content: |
203                         auto lo
204                         iface lo inet loopback
205 
206                         source /etc/network/interfaces.d/*.cfg
207                   - path: /etc/network/interfaces.d/aio.cfg
208                     owner: root:root
209                     permissions: '0644'
210                     content: |
211                         auto ens3
212                         iface ens3 inet manual
213 
214                         auto ens4
215                         iface ens4 inet manual
216 
217                         auto ens5
218                         iface ens5 inet manual
219 
220                         auto br-mgmt
221                         iface br-mgmt inet static
222                             address 172.29.236.100
223                             netmask 255.255.252.0
224                             gateway 172.29.236.1
225                             dns-nameservers 193.166.4.24
226                             bridge_ports ens3
227                             bridge_stp off
```

```
                            bridge_waitport 0
                            bridge_fd 0

                        auto br-vxlan
                        iface br-vxlan inet static
                            address 172.29.240.100
                            netmask 255.255.252.0
                            bridge_ports ens4
                            bridge_stp off
                            bridge_waitport 0
                            bridge_fd 0

                        auto br-storage
                        iface br-storage inet static
                            address 172.29.244.100
                            netmask 255.255.252.0
                            bridge_ports ens5
                            bridge_stp off
                            bridge_waitport 0
                            bridge_fd 0
    host_config_target:
        type: OS::Heat::MultipartMime
        properties:
            parts:
              - config: { get_resource: add_interfaces_target }
              - config: { get_resource: base_config }
              - config: { get_resource: target_packages_config }
              - config: { get_resource: disable_netplan }
              - config: { get_resource: target_inject_root_authorized_key }
    osa_host_target:
        type: OS::Nova::Server
        properties:
            flavor: { get_param: target_flavor }
            image: { get_param: target_image }
            networks:
              - port: { get_resource: mgmt_port_target }
              - port: { get_resource: tunnel_port_target }
              - port: { get_resource: storage_port_target }
            user_data_format: RAW
```

```
267             user_data: { get_resource: host_config_target }
```

```
1   # ./common-heat-templates/network/access-net.yaml
2   heat_template_version: 2016-10-14
3   description: |
4       Heat stack for common network that is routed to external 'public' network.
5   parameters:
6       public_net:
7           type: string
8       cidr:
9           type: string
10          default: 172.29.236.0/22
11      gateway_ip:
12          type: string
13          default: 172.29.236.1
14      dns_nameservers:
15          type: comma_delimited_list
16          default: 8.8.8.8
17  resources:
18      private_net:
19          type: OS::Neutron::Net
20      private_subnet:
21          type: OS::Neutron::Subnet
22          properties:
23              cidr: { get_param: cidr }
24              gateway_ip: { get_param: gateway_ip }
25              dns_nameservers: { get_param: dns_nameservers }
26              network: { get_resource: private_net }
27      router:
28          type: OS::Neutron::Router
29          properties:
30              external_gateway_info:
31                  network: { get_param: public_net }
32      router_interface:
33          type: OS::Neutron::RouterInterface
34          properties:
35              router_id: { get_resource: router }
36              subnet_id: { get_resource: private_subnet }
37  outputs:
```

```
    private_net:
        value: { get_attr: [ private_net, name ] }
    private_subnet:
        value: { get_attr: [ private_subnet, name ] }
```

```
# ./common-heat-templates/network/secondary-net.yaml
heat_template_version: 2016-10-14
description: |
    Heat stack for non-primary network, meaning that it will not be used as the
    default gateway.
parameters:
    cidr:
        type: string
        default: 172.29.240.0/22
    gateway_ip:
        type: string
        default: 172.29.240.1
resources:
    private_net:
        type: OS::Neutron::Net
    private_subnet:
        type: OS::Neutron::Subnet
        properties:
            cidr: { get_param: cidr }
            gateway_ip: { get_param: gateway_ip }
            host_routes:
              - destination: { get_param: cidr }
                nexthop: { get_param: gateway_ip }
            network: { get_resource: private_net }
outputs:
    private_net:
        value: { get_attr: [ private_net, name ] }
    private_subnet:
        value: { get_attr: [ private_subnet, name ] }
```

```
# ./common-heat-templates/config/inject-root-private-key.yaml
heat_template_version: 2016-10-14
description: |
    Heat stack for injecting given private key for root user.
```

```
parameters:
    private_key:
        type: string
resources:
    config:
        type: OS::Heat::SoftwareConfig
        properties:
            config:
                str_replace:
                    template: |
                        #!/bin/sh
                        echo "$private_key" | su - root -c 'tee -a .ssh/id_rsa'
                        su - root -c 'chmod 600 .ssh/id_rsa'
                    params:
                        $private_key: { get_param: private_key }
outputs:
    OS::stack_id:
        value: { get_resource: config }
```

```
# ./common-heat-templates/config/inject-root-authorized-key.yaml
heat_template_version: 2016-10-14
description: |
    Heat stack for injecting given public key as authorized key for root user.
parameters:
    authorized_key:
        type: string
resources:
    config:
        type: OS::Heat::SoftwareConfig
        properties:
            config:
                str_replace:
                    template: |
                        #!/bin/sh
                        echo "$authorized_key" | su - root -c 'tee
                        ↪  .ssh/authorized_keys'
                        su - root -c 'chmod 600 .ssh/authorized_keys'
                    params:
                        $authorized_key: { get_param: authorized_key }
```

```
20 outputs:
21     OS::stack_id:
22         value: { get_resource: config }
```

Appendix B Ansible preparation and configuration for the test deployment

```yaml
---
- name: Clone OpenStack-Ansible repository
  git:
    repo: https://github.com/openstack/openstack-ansible.git
    version: 20.1.2
    dest: /opt/openstack-ansible

- name: Ensure openstack_deploy directory exists
  file:
    path: /etc/openstack_deploy
    state: directory

- name: Copy configuration files
  copy:
    src: "files/{{ item }}"
    dest: "/etc/openstack_deploy/{{ item }}"
  with_items:
    - openstack_user_config.yml
    - user_secrets.yml
    - user_variables.yml
```

```yaml
---
# openstack_user_config.yml
cidr_networks:
  container: 172.29.236.0/22
  tunnel: 172.29.240.0/22
  storage: 172.29.244.0/22

used_ips:
  - "172.29.236.1,172.29.236.50"
  - "172.29.236.100"
  - "172.29.240.1,172.29.240.50"
  - "172.29.240.100"
  - "172.29.244.1,172.29.244.50"
```

```
14    - "172.29.244.100"
15    - "172.29.248.1,172.29.248.50"
16    - "172.29.248.100"
17
18  global_overrides:
19    internal_lb_vip_address: 172.29.236.100
20    external_lb_vip_address: 172.29.236.100
21    management_bridge: "br-mgmt"
22    provider_networks:
23      - network:
24          container_bridge: "br-mgmt"
25          container_type: "veth"
26          container_interface: "ens3"
27          ip_from_q: "container"
28          type: "raw"
29          group_binds:
30            - all_containers
31            - hosts
32          is_container_address: true
33      - network:
34          container_bridge: "br-vxlan"
35          container_type: "veth"
36          container_interface: "ens4"
37          ip_from_q: "tunnel"
38          type: "vxlan"
39          range: "1:1000"
40          net_name: "vxlan"
41          group_binds:
42            - neutron_linuxbridge_agent
43      - network:
44          container_bridge: "br-storage"
45          container_type: "veth"
46          container_interface: "ens5"
47          ip_from_q: "storage"
48          type: "raw"
49          group_binds:
50            - glance_api
51            - cinder_api
52            - cinder_volume
```

```
53            - nova_compute
54            - swift_proxy
55
56  # Galera, Memcached, RabbitMQ, Utility host
57  shared-infra_hosts:
58    aio1:
59      ip: 172.29.236.100
60
61  # Package repository host
62  repo-infra_hosts:
63    aio1:
64      ip: 172.29.236.100
65
66  # Glance, Nova, Heat API and Horizon host
67  os-infra_hosts:
68    aio1:
69      ip: 172.29.236.100
70
71  # Keystone
72  identity_hosts:
73    aio1:
74      ip: 172.29.236.100
75
76  # Neutron
77  network_hosts:
78    aio1:
79      ip: 172.29.236.100
80
81  # Nova compute
82  compute_hosts:
83    aio1:
84      ip: 172.29.236.100
85
86  # Cinder API
87  storage-infra_hosts:
88    aio1:
89      ip: 172.29.236.100
90
91  # Cinder volume service
```

```
storage_hosts:
  aio1:
    ip: 172.29.236.100
    container_vars:
      cinder_storage_availability_zone: cinderAZ_1
      cinder_default_availability_zone: cinderAZ_1
      cinder_backends:
        lvm:
          volume_backend_name: LVM_iSCSI
          volume_driver: cinder.volume.drivers.lvm.LVMVolumeDriver
          volume_group: cinder-volumes
          iscsi_ip_address: "{{ cinder_storage_address }}"
        limit_container_types: cinder_volume

# Rsyslog
log_hosts:
  aio1:
    ip: 172.29.236.100

# Load balancer
haproxy_hosts:
  aio1:
    ip: 172.29.236.100
```

```
---
# user_variables.yml
debug: false

openstack_service_publicuri_proto: https
openstack_service_adminuri_proto: https
openstack_service_internaluri_proto: https

keystone_service_internaluri_insecure: true
keystone_service_adminuri_insecure: true
```