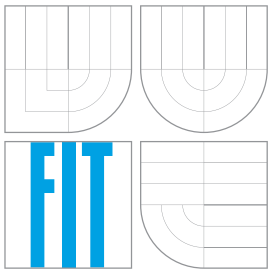
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EXTENSION OF OPENSTACK MODULES FOR ANSIBLE PLATFORM

ROZŠÍŘENÍ MODULŮ OPENSTACK PRO PLATFORMU ANSIBLE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
ADAM ŠAMALÍK

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. MARTIN HRUŠKA

BRNO 2016

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

ŠAMALÍK, Adam. *Extension of OpenStack Modules for Ansible Platform.* Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hruška Martin.

Extension of OpenStack Modules for Ansible Platform

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Adam Šamalík
May 8, 2016

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

1 Introduction **3**
1.1 Structure of the document . . . 3

2 Technology Overview **4**
2.1 Cloud Computing Introduction . . . 4
2.1.1 Difference Between Traditional Virtualisation and Cloud Computing 4
2.1.2 Cloud Applications and Scaling . . . 5
2.1.3 Models of Cloud Computing . . . 5
2.1.4 Models of Cloud Deployment . . . 5
2.2 OpenStack . . . 5
2.2.1 Core OpenStack Components . . . 6
2.3 Ansible . . . 7
2.3.1 What is Ansible . . . 7
2.3.2 Advantages of the Ansible Technology . . . 8
2.3.3 Ansible Playbooks . . . 9
2.3.4 Tracking of Host State . . . 11
2.3.5 Roles . . . 12

3 OpenStack Architecture **13**
3.1 Conceptual Architecture . . . 13
3.2 Logical Architecture . . . 13
3.3 Physical Architecture . . . 15
3.4 Installation of Core Components . . . 15
3.4.1 OpenStack Identity Service . . . 15
3.4.2 OpenStack Image Service . . . 16
3.4.3 OpenStack Compute Service . . . 17
3.4.4 OpenStack Networking Service . . . 19
3.4.5 OpenStack Block Storage Service . . . 21
3.4.6 OpenStack Dashboard Service . . . 23

4 Existing Methods of Automated OpenStack Deployment **25**
4.1 Packstack . . . 25
4.2 OpenStack-Ansible . . . 25

5 Implementation Design **26**
5.1 Design of the Physical Architecture . . . 26
5.1.1 Selecting OpenStack Services . . . 26
5.1.2 Selecting Specific Implementations . . . 26

5.1.3 Physical architecture 27
5.2 Design of the Ansible Playbook 27
5.2.1 Ansible Roles 27
5.2.2 Applying Roles to the Hosts 33

6 Implementation and Testing **35**
6.1 Third-party Modules in the Playbook 35
6.2 Testing of the Deployment 35
6.2.1 Description of the Host Environment 35
6.2.2 Using Ansible with Vagrant 36
6.2.3 Running the Deployment 36

7 Conclusion **37**

Bibliography **38**

Appendices **40**
List of Appendices 41

Chapter 1

Introduction

OpenStack is a cloud operating system with distributed architecture. Deploying OpenStack can be a very complex task requiring deep knowledge in distributed system architecture, networking, storage, and virtualisation. Because of its complexity, it is very hard to create an installation script. These days, OpenStack does not come with an installer. However, there are several projects that try to simplify the process. These solutions are often limited and might not be suitable for production use.

The goal of this thesis is to design and implement an Ansible installation script, called Ansible Playbook, which would enable to install basic OpenStack deployment and would also offer flexibility in configuring more complex scenarios by reusing the Ansible Roles designed in this thesis, and building new, more complex ones on top of them. In other words, the Ansible Playbook designed and implemented here would create a decent starting point for more complex project.

1.1 Structure of the document

This thesis has been divided into five chapters, excluding this Introduction and Conclusion 7.

The second chapter, called Technology Overview 2, introduces the reader into the technologies that will be used in this thesis. The third chapter 3 describes the OpenStack architecture, the core services, and how they communicate with each other.

The fourth chapter 4 offers a short analysis of two existing solutions to deploy the OpenStack cloud and explains their limitations.

Following chapter 5 describes an implementation design of the Ansible Playbook, and also describes a custom architecture, which would be used as a reference and a testing environment. The sixth chapter 6 shows some details of the implementation process and testing.

Chapter 2

Technology Overview

2.1 Cloud Computing Introduction

In the beginning of computer times, we used dedicated servers to run enterprise applications. One application would run on a single physical machine. These machines could be small servers or large mainframes. When an organization grown and their application required more resources, the server would scale up, which means adding more processors, memory, or storage to the host. This method was not flexible enough, as every change required physical access to the datacentre and buying new hardware. This problem has been solved by virtualization.

Virtualization is a technology that enables running several virtual servers on a single host. Every virtual server would have its own operating system and would act as a physical host. These servers could be dynamically scaled up and down without the need of a physical access to the datacentre. New virtual servers could be provisioned, unused servers could be deleted to make space for scaling, which would reflect the business needs in more flexible way then dedicated servers. This method is also referred to as a traditional virtualization.

2.1.1 Difference Between Traditional Virtualisation and Cloud Computing

In a traditional virtualization, virtual servers are created and managed as part of a single physical host. Virtual machines on every physical host must be managed separately, directly on the host. This approach, again, might not seem flexible enough for large corporations running tens, hundreds, or even thousands of virtual servers on many physical hosts in several datacentres. This is why a technology called cloud computing has emerged.

Cloud computing, in a similar way as traditional virtualization, is a technology that manages running of virtual machines. The main difference is that in cloud computing, a virtual layer of resources, such as computing power, memory, and storage, is created over one or more datacentres. These resources are then used to provision virtual servers. However, it is up to the cloud computing technology to choose on which host the server would run.

This approach is very flexible in terms of creating, deleting, and scaling virtual servers. It also brings new technology, such as software-defined networking, or an object store, both of which will be described later in the thesis.

2.1.2 Cloud Applications and Scaling

The cloud computing flexibility has completely changed the way enterprise applications are designed and created. Instead of building a large monolithic application that would run on a single host, the application has been broken down to smaller parts communicating with each other. These parts are called microservices and each part runs on a separate virtual machine. This approach of creating an application consisting of several microservices is called distributed architecture.

This architecture style enabled another innovation in terms of scaling. As I described in the beginning of this section, with physical servers and also in traditional virtualization, applications are scaled up by adding more processors, memory, or storage. In cloud computing, the application can scale by adding more virtual machines to handle the workload. This approach is called scaling out. Such scaling can be also done automatically. This means that the application does not need to have all the resources reserved for itself all the time. For example, an application can run in business' private datacentre, which provides the computing power for the majority of time. In a need of extra computing power, the application can automatically scale out and use resources from another datacentre, which might be run by different company as a paid service. This leads us to different models of cloud computing and deployment models.

2.1.3 Models of Cloud Computing

- **Infrastructure as a Service (IaaS)** provides computing resources, which can be used to provision virtual machines and software-defined networks. In this model, the cloud consumer manages the application, operating system, storage, networking, and computing resources. OpenStack is an IaaS solution.
- **Platform as a Service (PaaS)** provides an operating system including libraries and programming languages. In this model, the cloud consumer manages the application, but not the underlying infrastructure.
- **Software as a Service (SaaS)** provides an operating system and the application. In this model, the cloud consumer does not manage the application nor the underlying infrastructure.
  [16]

2.1.4 Models of Cloud Deployment

- **Public cloud** is a cloud run by a cloud provider, provided as a service.
- **Private cloud** is a cloud run and used by a single organization.
- **Hybrid cloud** is a combination of public and private cloud. The cloud consumer has the ability to run an application in their datacentre and expand to the private cloud when more resources are needed.
  [16]

2.2 OpenStack

OpenStack is an open-source cloud operating system - a platform that controls large pools of compute power, storage, and networking resources of one ore more datacentres. These

resources are then used to provision virtual machines.

The service is managed by a dashboard, which gives the administrators control over the whole cloud, and users the ability to provision and manage resources via single web interface. OpenStack consists of several components. The base six components provide identity, networking, object storage, block storage, a service for managing images, and the compute service. Other services may include an orchestration service, telemetry service to measure usage of the cloud, or a database as a service solution (DBaaS).

All services communicate with each other via public HTTPS endpoints and use Advanced Message Queuing Protocol (AMQP), which is a protocol supporting sending and receiving messages between distributed systems. [16]

2.2.1 Core OpenStack Components

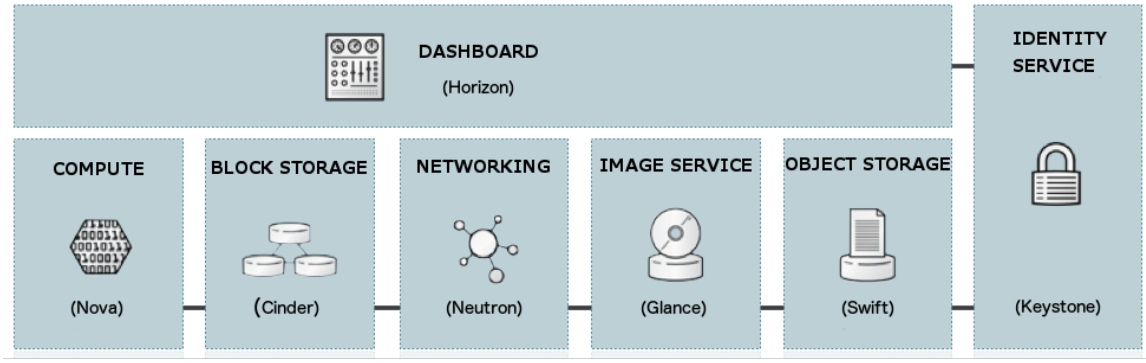


Figure 2.1: OpenStack core services [Source: [1]]

OpenStack is composed of several components [16] called services, which can be seen on picture 2.1, and are also described in this section.

- **Horizon (dashboard)** is a web interface for managing OpenStack services providing graphical user interface. Horizon supports operations like launching instances, managing networking, and setting access controls.
- **Keystone (identity)** is a centralized identity service providing authentication and authorization for other services. It also provides a catalogue of services running in the OpenStack cloud.
- **Neutron (networking)** provides virtual networking infrastructure in the OpenStack cloud, including networks, subnets and routers. Other advanced services such as firewalls, virtual private networks (VPN) or quality of service (QoS) are also supported. This service handles the creating and management of the networking infrastructure for the cloud administrator and users.
- **Cinder (block storage)** manages persistent block storage volumes used by virtual machines. The service supports creating of snapshots which can be used for backing up data. Then backup can be then used to restore data or to create some new block storage volumes. This service is often used by the virtual machines as a storage.
- **Swift (object storage)** provides an object storage that allows users to store and retrieve files. Swift has a distributed architecture that enable horizontal scaling and

redundancy. Data replication is managed by software, which allows larger scalability and redundancy than dedicated hardware.

- **Glance (image)** is a registry of virtual machine images. Users can copy server images and use them as templates when setting up new virtual servers.
- **Nova (compute)** is a service that manages virtual machines running on compute nodes. Nova is designed to scale horizontally on standard hardware and to download images to launch new instances. Nova is a distributed component that interact with Keystone for authentication, Glance for images and Horizon for a web interface. It uses libvirtd, qemu, and kvm for the hypervisor.
- **Ceilometer (metering)** is a service that provides a centralized source for metering and monitoring data, which can be used to meter and bill users.
- **Heat (orchestration)** an advanced service to orchestrate multiple cloud application using the Amazon Web Services (AWS) template format. The software integrates other core components of OpenStack into a one-file template system. Templates can be used to create most of OpenStack resources, such as instances, floating IPs, volumes, security groups or users. Heat also offers advanced functionality such as high availability, auto scaling and nested stacks.

2.3 Ansible

2.3.1 What is Ansible

Ansible is an IT automation engine which can be used as a configuration management tool, for application deployment, orchestration of deployment and provisioning new servers, all of which will be described later in the section. [3]

Configuration management

Configuration management typically means writing some kind of description of our servers - a state in which we want the servers to be. This definition of a state can include information about packages that should be installed (or, more precisely, present) on the server, configuration files containing specific values and have specific permission and ownership, that the correct services are running, and so on.

Except Ansible, common configuration management tools are for example Chef, Salt, or Puppet. [15]

Deployment

Application deployment is a process of taking a source code of a software that has been written internally, building it to get the binaries, copying the required files to the servers that are supposed to run the application, and then starting the necessary services.

Tools other than Ansible that can be used for deployment are for example Capistrano, or Fabric, which are both open-source. [15]

As Ansible can manage both deployment and configuration management, using a same tool for these tasks can make it easier and well-arranged for the people using it.

Orchestration of deployment

We need to use orchestration when there are more servers involved and we need to run tasks in a specific order. This can mean, for example, bringing up the database server first and then starting the web servers. Or in case of updating web servers, we need to take them out of the load balancer one at a time to preserve stable service without outages.

Ansible is designed to support orchestration by using a simple model of dependencies build directly into the core architecture - so the tasks are automatically executed in the correct order.

Provisioning

The term provisioning a new server is often used in a cloud environment. It means creating a new virtual machine instance with operating system installed and having the networking and access configured in a way that the machine can be used.

However, provisioning is also used with baremetal machines, because, in the end, all virtual and cloud machines run on a physical server. Ansible can provision all the machines in the datacenter and can be also integrated with several dataentre management tools, like (not limited to) Red Hat Satellite, Hanlon, or Cobler.

Ansible supports several cloud providers including Amazon EC2, Microsoft Azure, Digital Ocean, Google Compute Engine, Linode, Rackspace and clouds supporting the OpenStack APIs. [15]

2.3.2 Advantages of the Ansible Technology

There are several features of the Ansible technology that are important to mention:

Syntax is Easy to Read

The ansible configuration management scripts are called playbooks. Playbooks and their structure will be described later in this section. Important to know right now is that they are build on top of YAML syntax, which is a data format language that was designed to be easy to read for people.

When used properly, the ansible playbooks might be seen as an executable documentation. And it will never be outdated, because it is also the code, that gets executed.

No Agents Required

To manage a server by ansible, it only needs a Python version 2.5 or later and an SSH to be installed. Ansible does not need any agents or additional software to be installed on the servers. It also does not need any special management interfaces, as it runs on the existing network infrastructure using SSH.

Push-based mechanism

There are two main concepts of deploying a change to servers: push-based and pull-based mechanisms.

Some configuration management, that use agents, are pull-based. An example of these tools can be Puppet or Chef. The pull-based mechanism work in the following way:

1. Administrator makes change to the configuration scripts

2. Administrator pushes the changes to a central service
3. Agent running on the server checks for changes periodically
4. Agent downloads the change from the central service
5. Agent executes the configuration script locally and changes the state of the server

Ansible uses the push-based mechanism, in which the administrator controls when a change is applied to the server. [15] The administrator does not need to wait for a timer to expire before the change is applied. This is an advantage of the Ansible solution, as it offers more control over the overall system. The push-based mechanism works in the following way:

1. Administrator makes change to the configuration scripts
2. Administrator runs the playbook
3. Ansible connects to the servers and executes modules, making changes to the server

Ansible, however, also supports a pull-based mechanism, using a tool called ansible-pull.

Thin Layer of Abstraction

Some configuration management tools provide a thick layer of abstraction, which enables the administrator to manage multiple operation systems using a single script. For example, an abstraction called package could be used to install a package on a system, regardless its type. It could be a yum-based, or a apt-based system. However, these abstractions can be even bigger.

Ansible does not use these kind of abstractions. This allows the administrator to write scripts for a specific system without the need of learning the abstractions, which makes the learning process shorter, and reading the scripts easier. [15]

2.3.3 Ansible Playbooks

As already mentioned previously, ansible configuration scripts are called playbooks. Playbooks can define the configuration of the target servers, and they can orchestrate the steps in which the configuration is applied. Playbooks consist of several parts: plays, tasks, and modules. [15] [13]

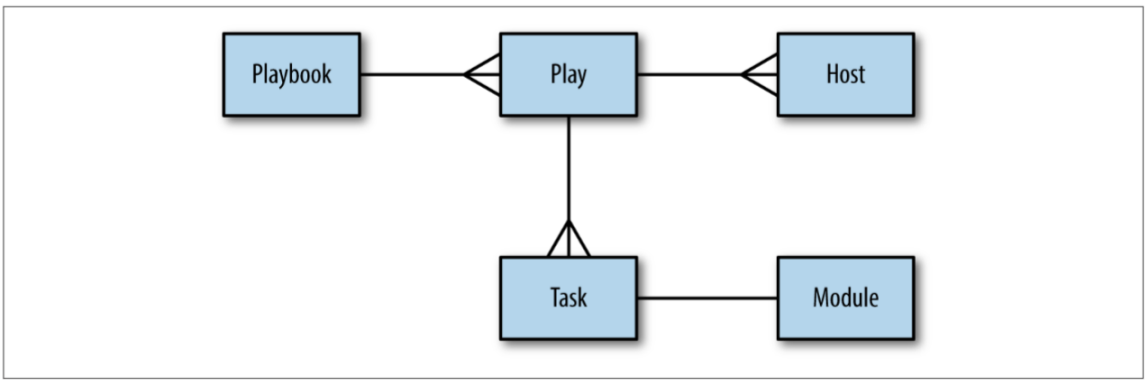


Figure 2.2: Ansible playbook anatomy [Source: [15]]

Plays

Playbook is a list of plays. Plays can be thought of as a connection of tasks to hosts. A play must contain:

- A list of hosts
- A list of tasks that would be executed on the hosts

Plays also support several optional settings that affect the way it is executed. An example of these settings can be:

- `name` - A description of the respective task. The names should be always used, as they will be printed at the time of run, showing what changes has been made to a particular server.
- `sudo` - A boolean value determining if the tasks will be executed with root privileges.
- `vars` - A list of variables that would be used for the particular task. They act as a parameters.

An example of a play is shown below:

```
- name: Configure webserver with nginx
  hosts: webservers
  sudo: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: >
        src=files/nginx.conf
        dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file: >
        dest=/etc/nginx/sites-enabled/default
        src=/etc/nginx/sites-available/default
        state=link

    - name: copy index.html
      template: >
        src=templates/index.html.j2
        dest=/usr/share/nginx/html/index.html mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

[15]

Tasks

Play is a list of tasks. Every task represents a single module that would be executed. Every task must contain:

- A name of the particular module
- A list of arguments for the module

Task can also include a definition of its name. Using of names is a good practise, as they are printed at the time of run and can be used to track changes made by the scripts. They also act as comments, improving readability of the playbook.

Module

Modules are scripts that perform the desired action on the target server.

They are designed to be idempotent, which means that they can be run multiple times, but will only make changes when there is a difference between the present and desired state. In other words, running a playbook for the second time without making changes will not affect the target server in any way.

An example of modules can be:

- `copy` - Copies a file from the local machine to the remote host.
- `service` - Manages the services running on the remote host. It can start, stop, or restart a services.

2.3.4 Tracking of Host State

When ansible execute modules on a particular server, they may or may not make a change. They only make change when the present state is different from the desired state. At the time of run, ansible prints out the names of tasks that are being executed. If the state has changed, the module will return changed state. And if no action was needed, an ok state will be returned instead. Ansible will then print the result, as seen on the following example:

```
TASK: [Install packages] *********************************
changed: [webserver1]

TASK: [Set configuration] ********************************
ok: [webserver1]
```

Handlers

This detection of a change can be used by a mechanism called handlers. [15] Handlers are actions triggered at the end of run only when a specific change has occurred. However, the action will be executed only once.

For example, a change of a web server configuration might require restart of the respective service. Also, an update of the web server package might require a restart of the service. If any of these actions occur, a handler responsible for restarting the service will be notified and executed at the end of run. It will be executed only once, even when both of the changes above will occur.

2.3.5 Roles

When managing a large number of servers, a single playbook describing changes of all of the servers would become too long and hard to maintain. Also, applying a common configuration to multiple servers would result in a duplication of code. To solve this problem, a mechanism called Roles has been introduced to Ansible. [13]

Roles are the main mechanism to break large playbooks into multiple files, which simplifies writing complex playbooks. It also makes them easier to reuse.

A role can be thought of as something that should be assigned to one or more hosts. For example, a database role will be assigned to servers acting as a database servers.

Roles can also be dependent, which means that a role can require other role to be applied before. Using this feature, Ansible will automatically ensure that roles are applied in the correct order.

Chapter 3

OpenStack Architecture

This chapter describes the OpenStack architecture, how the components communicate and what they are used for. Selected components, which are needed for the deployment option used later in this thesis, will be described in more detail.

3.1 Conceptual Architecture

Conceptual architecture is a high level architecture that shows the core components of a system and their relations including description of responsibilities for each component.

The relationships among the OpenStack services can be seen on picture 3.1.

3.2 Logical Architecture

OpenStack consists of several independent parts called OpenStack services.[16] Authentication for all services is provided by a common identity service. The services interact with each other via public APIs. [4]

OpenStack services are internally composed of several processes. Each service has at least one API process. This API process listens for API requests, does necessary preprocessing and passes them to the respective process or processes within the service. The actual work of a service is done by distinct processes. An exception to this rule is the identity service.

The processes within a service communicate with each other via message broker using AMQP protocol. The most used AMQP broker is RabbitMQ, which is also used in this thesis.

The state of a service is stored in a database. OpenStack support several databases, such as MySQL, MariaDB, and SQLite. The database solution used in this thesis is MariaDB. [4]

Users can interact with OpenStack in several ways. There is a graphical web-based interface, which is implemented by OpenStack Dashboard. Other ways include command-line clients provided by all basic services, and by API requests using web browser plugins or curl.

The most common, but not the only possible logical architecture can be seen on picture 3.2.

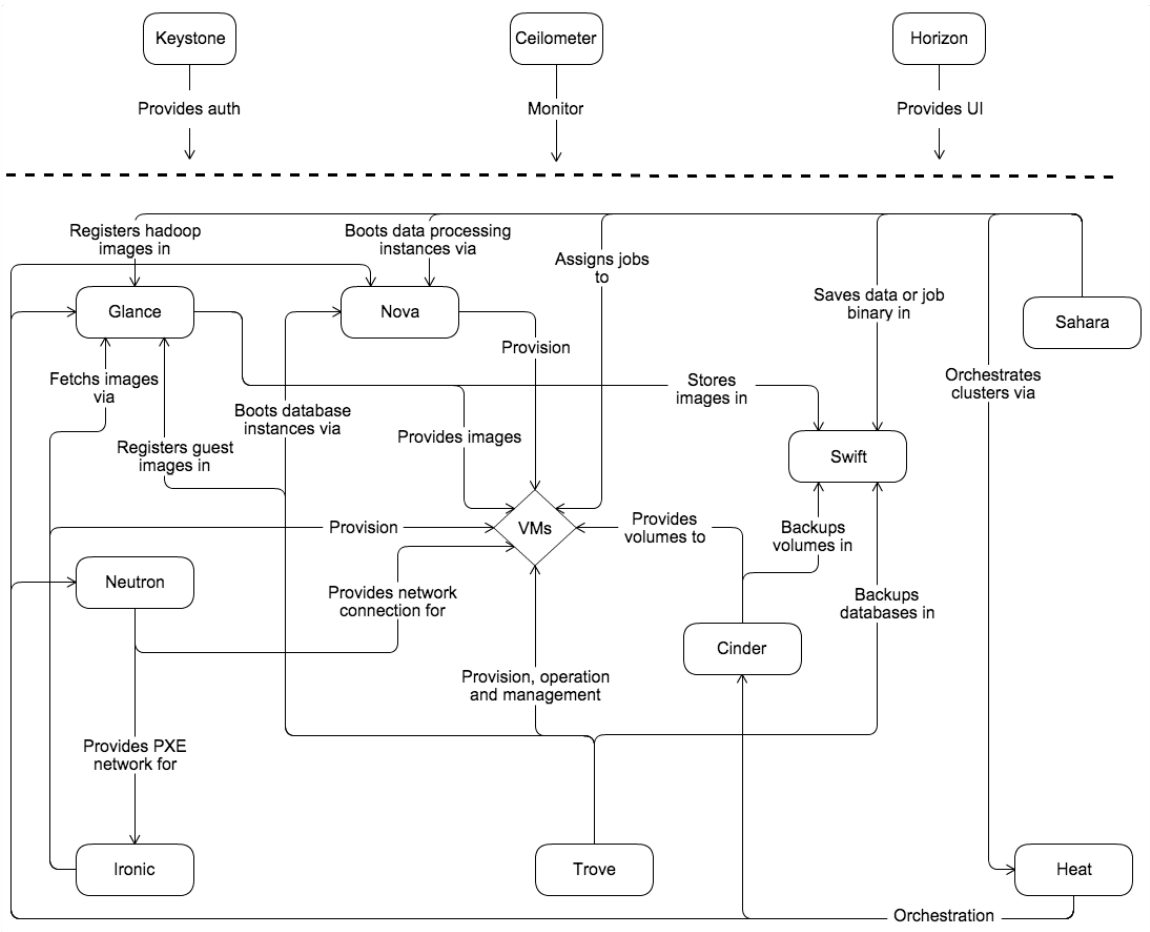


Figure 3.1: OpenStack conceptual architecture [Source: [2]]

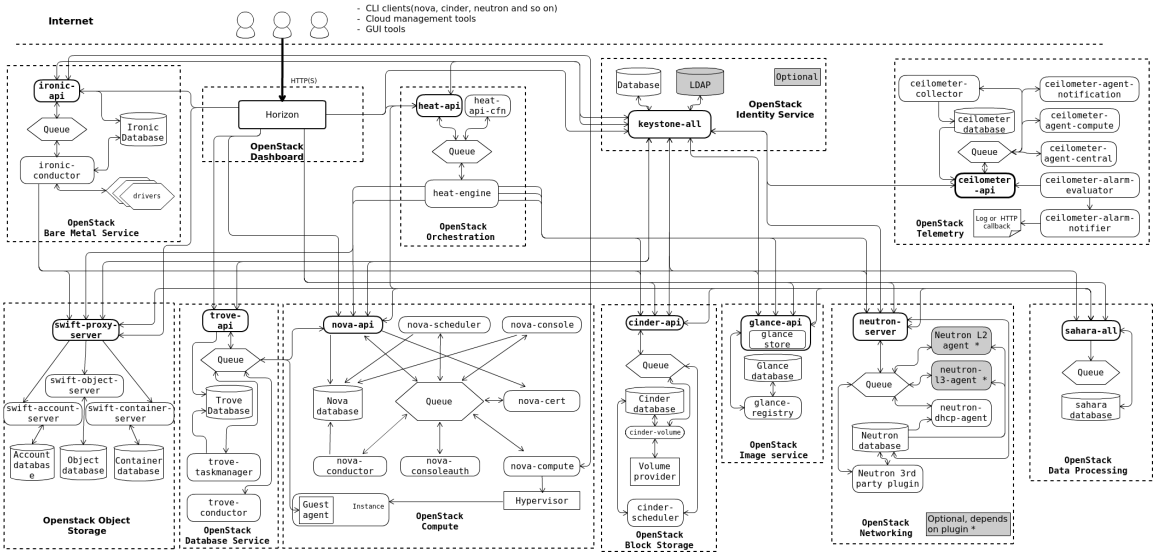


Figure 3.2: OpenStack logical architecture [Source: [**?**]]

3.3 Physical Architecture

OpenStack is a distributed system, which enables cloud architects to design varieties of physical architecture reflecting needs of the cloud consumer.

"Designing an OpenStack cloud is a great achievement. It requires a robust understanding of the requirements and needs of the cloud's users to determine the best possible configuration to meet them. OpenStack provides a great deal of flexibility to achieve your needs..." [14]

In other words, designing a physical architecture for particular deployment is a non-trivial task.

3.4 Installation of Core Components

3.4.1 OpenStack Identity Service

The OpenStack identity service provides a single point of authentication and authorisation. When openStack services receive a request from a user, they check with the identity service to verify the authorisation of the user.

It also provides a catalog of OpenStack services. When installing OpenStack cloud, each service needs to be registered in the identity service. The identity service then tracks which services are installed and where they are located on the network.

OpenStack services also use the identity service as a common unified API.

Identity Abstractions

The identity service uses the following abstract organisations:

- **Service** is an OpenStack service, such as the identity (Keystone) service, compute (Nova) service, etc.
- **Endpoint** is an address accessible via network and acs as an access-point of a service.
- **Region** represents a general division of the OpenStack deployment. The identity service also supports sub-regions, which allows the creation of a tree-structured hierarchy.
- **Authentication** is the process of confirming the identity of a user. To confirm an incoming authentication request, the identity service validates a set of credentials supplied by the user. These can be a name and password, or a name and API key. After validation of the user credentials, the identity services then issues an authentication token. This process is done only one on in the beginning of a session and user then provide this token in all subsequent requests.
- **Credentials** are data to identify the user, such as name and password, or name and API key.
- Authentication **token** is an alpha-numeric string used to access to OpenStack APIs and resources. Token is valid for a finite duration and can be revoked at any time.
- **Domain** is a collection of projects and users. This collection defines administrative boundaries for managing identity entities. Domains can be used to represent an

individual person, a company, or an operator-owned space. User can be granted the administrator role within a domain. This domain administrator can then create users, projects, and groups in the domains as well as create and assign roles to users or groups.

- **Project** is used to isolate resources or identity objects. Projects can be used to map to a customer, account, organisation, or tenant.
- **Role** is a personality containing a set of user rights and privileges. A user contains a list of roles.
- **Group** is a collection of users and is owned by a domain. A group can be assigned a role that applies to all users in the group.
- **User** is a representation of a person, system or service using the OpenStack services. [16]

OpenStack Identity Components

The identity service consists of these components:

- **Server** - A centralised server, providing authentication and authorisation services. It uses a RESTful interface.
- **Drivers** - Drivers are used to access identity information provided by external repositories like LDAP, or existing SQL databases.
- **Modules** - Middleware modules that check service requests, extract user credentials and send them to the Server for authorisation. The modules are integrated with the OpenStack components by using the Python Web Server Gateway Interface. [5]

3.4.2 OpenStack Image Service

The OpenStack Image service controls image storage and management and allows users to discover, register, and retrieve images.

Images provide templates for virtual machine filesystems. Each virtual machine runs from a copy of a base image and several virtual machines can be run from a single image. Any changes made to the virtual machines will not affect the image. Users can also create a snapshot - a state of a virtual machines running disk - and build a new image based on these snapshots.

The implementation of the image service is called Glance. It supports various backends as a storage, which can be normal filesystems, OpenStack object storage, HTTP, RADOS block devices, and Amazon S3. Some of the storage backends can be read-only. The architecture in this thesis will use normal filesystem mounted to the controller node.

Basic Components

The image service consists of the following components:

- **glance-api** - Accepts API calls.

- **glance-registry** - Stores, processes, and retrieves metadata about images like size and type. This is an internal service only and should not be exposed outside of the image service.
- **Database** - An SQL database to store image metadata.
- **Storage** repository for image files - Backend that stores the images itself.

3.4.3 OpenStack Compute Service

The OpenStack Compute service is the major part of an Infrastructure as a Service (IaaS) system. It runs and manages the virtual machines running in the OpenStack cloud. It can also provide networking for the virtual machines. [5]

The OpenStack Compute service scales horizontally and is commonly deployed on multiple hosts, [14] often called compute nodes.

The service manages virtualisation, but it does not include any virtualisation software. Instead, it uses drivers to interact with the underlying virtualisation backend, also called virtualisation provider. It is also possible to use multiple providers in different availability zones.

The OpenStack Compute service supports these following virtualisation providers: [4]

- Baremetal
- Docker
- Hyper-V
- Kernel-based Virtual Machine (KVM)
- Linux Containers (LXC)
- Quick Emulator (QEMU)
- User Mode Linux (UML)
- VMware vSphere
- Xen

Service Architecture

The architecture of the Compute service could be divided into the following four parts:

- **API Server** is the compute controller that commands and controls the hypervisor, storage, and networking available to the end users. It manages the API endpoints, which are basic HTTP web services that handle the authentication, authorisation, and basic command and control functions. It support various API interfaces including Amazon, Rackspace, and others, which enables compatibility with multiple existing tools already created for other cloud platforms. This open approach also prevents vendor lock-in.

- **Message Queue** manages the interaction between compute nodes, the networking controllers, API server, and the scheduler. It is provided by the Advanced Message Queuing Protocol (AMQP). Several implementations are available, the most common are RabbitMQ, Qpid, ZeroMQ, and others.
- **Compute Worker** manages the virtual machines that run on the compute nodes. It is managed via API that provides commands to run, delete, and reboot instances, attaching and detaching volumes, and to get local console output.
- **Network Controller** manages the networking resources available on the compute nodes. It is managed via API that provides commands for allocating fixed IP addresses, configuring VLANs for projects, or configuring networks for compute nodes. [4]

OpenStack Compute Components

The OpenStack Compute service consists of the following components:

- **`nova-api` service** - Accepts API calls and responds to them. It supports the OpenStack Compute API, the Amazon EC2 API, and a special Admin API.
- **`nova-api-metadata` service** - Accepts metadata requests from instances.
- **`nova-compute` service** - A worker daemon that manages virtual machines via virtualisation provider API. It supports XenAPI, libvirt, VMwareAPI, and others. The state of the instances is always saved in the database.
- **`nova-scheduler` service** - This service determines on which compute node a new virtual machine should be started.
- **`nova-conductor` module** - It is a point of access to the database for other nova services. It eliminates direct access to the database by other services. This service scales horizontally and can be deployed on multiple hosts.
- **`nova-cert` module** - This is only needed to use with the Amazon EC2 API, used to generate certificates.
- **`nova-network` worker daemon** - Similar to the nova-compute service, but for networking. Accepts networking tasks from the queue and manipulates the network.
- **`nova-consoleauth` daemon** - Provides authorisation for users provided by the console proxies - see nova-novncproxy and nova-xvpvncproxy for more information. It needs to run for the proxies to work.
- **`nova-novncproxy` daemon** - Proxy for accessing running virtual machines via a VNC connection. Supports browser-based HTML5 clients.
- **`nova-spicehtml5proxy` daemon** - Proxy for accessing running virtual machines via a SPICE connection. Supports browser-based HTML5 clients.
- **`nova-xvpvncproxy` daemon** - Proxy for accessing running virtual machines via a VNC connection. Supports an OpenStack-specific Java client.

- **`nova-cert` daemon** - Manages x509 certificates.
- **`euca2ools` client** - A set of command-line interpreter commands to manage the cloud resources via Amazon EC2 interface.
- **`nova` client** - A command-line client for the end user.
- **Message Queue** - Passes messages between nova daemons. Provided by the Advanced Message Queuing Protocol (AMQP). It is usually implemented by RabbitMQ.
- **SQL Database** - Stores most of the build-time and run-time states of the infrastructure, including information about available virtual machine types, virtual machines in use, available networks, and information about projects. OpenStack Compute supports SQLite3 for test and development work, MySQL and MariaDB, and PostgreSQL.
  [5]

3.4.4 OpenStack Networking Service

The OpenStack Networking service enables users to create virtual networks and attach the devices managed by other OpenStack services to them. It provides an API that lets users to configure and manage network connectivity and addressing. The network services include L3 forwarding, NAT, load balancing, firewalls, and VPN.

Networking provides the end users the ability to create virtual networks, subnets, routers, and firewalls. All of these will be explained later in this section.

It also supports variety of plug-ins which can enable interoperability with several commercial and open source technologies. This plug-in architecture provides flexibility when designing custom OpenStack architecture and deploying it.

Networking

Before using or deploying the OpenStack Networking service, the following general facts about networking [6] should be understood:

- **Ethernet** is a networking protocol that is being used by most wired network interface cards. In the OSI model of networking, Ethernet operates on the second layer, also referred to as layer 2, L2, link layer, or data link layer. Every host has unique identification called Media Access Control (MAC) address.

  Ethernet can be conceptually think of as a single bus, to which each of the network host is connected. However, modern networks use devices called switches, and every device is connected directly to them.

  The OpenStack Dashboard uses this simple model to visualise the network topology to the end user. This ethernet network is sometimes referred to as a layer 2 segment.

- **VLAN** is a networking technology that creates separate virtual network on a single switch in a way, that devices connected to the same switch can not each other's traffic, if they are on different VLANs.

  OpenStack uses VLANs to isolate the traffic of different tenants, even when their instances run on a single host.

Each VLAN has a numerical ID between 1 and 4095. For example, a VLAN with an id of 10 will be referred to as VLAN 15.

- **Address Resolution Protocol (ARP)** - As pointed out above, network devices use MAC addresses to be identified. However, TCP/IP applications use IP addresses as an identifiers. The Address Resolution Protocol (ARP) bridges the gap by translating IP addresses into MAC addresses.
- **Dynamic Host Configuration Protocol (DHCP)** dynamically assigns IP addresses to network hosts. These hosts are called DHCP clients.
- **TCP, UDP, and ICMP** - Software applications communicating over an IP network use another protocols above IP. In the OSI model of networking, they use the fourth layer, which is also referred to as layer 4, or transport layer. There are three main protocols:
  - *Transmission Control Protocol (TCP)* is the most commonly used layer 4 protocol. It is a connection-oriented protocol.
  - *User Datagram Protocol (UDP)* is mostly used to transfer real-time information like voice or video. It is a connectionless and unreliable protocol.
  - *Control Message Protocol (ICMP)* is used for sending control messages.
- **Switch** is device that allow packets to travel from one node to another. They connect hosts belonging to the same layer-2 network. Switches forward the traffic based on the destination Ethernet address in the packet header.
- **Router** is device that allows communication between nodes on different layer-3 networks. They route the traffic based on the destination IP address in the packet header.
- **Firewall** is device that restricts traffic to and from hosts on a network by special rules defined on the device. They are supposed to protect hosts from unauthorised access and attacks.
- **Load Balancer** is device that allows even distribution of traffic across several hosts. They are supposed to avoid overload of a single host. They also prevent a single point of failure as they enable the traffic to be processed by multiple hosts.

Networking Concepts in OpenStack

OpenStack uses the following concepts to enable users to create their own virtual network infrastructure:

- **Network** is an isolated L2 segment. There are two types of network:
  - *Tenant Networks* are managed by the end user and are used within their projects. These networks are fully isolated from other projects.
  - *Provider Networks* are managed by the OpenStack administrator and map ti the existing physical network in the datacenter. These networks are mainly used to enable external connectivity to the virtual machines running in the cloud. Each project should have at least one public provider network.

- **Subnet** is a block of IP addresses and associated configuration state. They are used to allocate IP addresses to ports that are created on a network.
- **Port** - Not to be confused with TCP or UDP port. In the OpenStack terminology, port is a connection point for attaching a single device (such as NIC of a virtual machine) to a virtual network. It also describes the configuration like MAC address and IP address.
- **Security Groups** enable users to define their own firewall rules in groups. They can control traffic in both direction (called ingress and egress). These rules are then applied to a port. A port can be assigned with multiple security groups.
  [6]

OpenStack Networking Components

The OpenStack Networking service is composed of the following components:

- **neutron-server** - accepts API requests and routes them to the appropriate Networking plug-in
- **OpenStack Networking plug-ins and agents** - the main logic is implemented by several plug-ins. They plug and unplug ports, create networks, subnets, and provide IP addressing. OpenStack Networking ships with plug-ins and agents for Cisco virtual and physical switches, NEC OpenFlow products, Open vSwitch, Linux bridging, and the VMware NSX product. The architecture in this thesis will use Linux bridging.
- **Messaging queue** - routes the information between neutron-server and other agents. It also act as a database for particular plug-ins. The architecture in this thesis will use centralised RabbitMQ service running on a controller node.
  [5]

The architecture in this thesis will use the following plug-ins and agents for networking:

- **Modular Layer 2 plug-in** - builds layer-2 (bridging and switching) virtual networking infrastructure for virtual machines. IT uses the Linux bridge mechanism.
- **Linux bridge agent** - builds the networking infrastructure for virtual machines and also manages VXLAN tunnels for private networks and security groups.
- **Layer 3 agent** - is the most commonly used agent that provides layer-3 (routing) services for virtual networks.
- **DHCP agent** - provides DHCP services for virtual networks.

3.4.5 OpenStack Block Storage Service

The OpenStack Block Storage service adds persistent block storage to a virtual machine instances and provides an infrastructure for managing volumes. This service is similar to the Amazon EC2 Elastic Block Storage (EBS) offering.

The service supports various backends which can be consumed using a Block Storage driver. An example of supported drivers that are available can be NAS/SAN, NFS, iSCSI, Ceph, and more. A usage of multi-backend configuration is also supported. [5]

To install the OpenStack Block storage service, it is important to understand a number of concepts, because there are several choices of deployment. [14] Apart from choosing the right storage backend, it mostly depends on the final architecture - which can be single node or multi-node. For example, this thesis uses a multi-node architecture with separate block storage node, and will use a local LVM storage as a backend.

Logical Volume Management (LVM)

Because this thesis will use an LVM as a block storage provider, it is important to understand the basic concepts of LVM.

LVM creates a layer of abstraction over physical storage, which allows you to create multiple logical storage volumes. From an application view, these logical volumes are the same as traditional disk partitions. The hardware storage configuration is also hidden from the software, so it can be resized and moved without stopping services or unmounting filesystems. And in an opposite way, the logical volumes can be also resized without changing the underlying physical storage. This solution provides much more flexibility than using traditional partitions directly. [7]

As shown on the picture 3.3, LVM consists of three basic layers:

- **Physical Volume (PV)** is the underlying physical block storage device, which can be a partition or the whole disk. To use the device for LVM, it needs to be initialised as a Physical Volume (PV) by placing a label near the start of the device.
- **Volume Group (VG)** is a combination of several physical volumes. However, volume group can also consist of a single physical volume. This layer provides a pool of disk space used to create logical volumes (LV) in the same way disks are divided into partitions.
- **Logical Volume (LV)** is the final volume which is used by filesystems and applications. In LVM, a volume group is divided into several logical volumes.

Block Storage Components

The OpenStack Block Storage service consists of the following components:

- **cinder-api** - Accepts API requests and routes them to the cinder-volume process for action.
- **cinder-volume** - This process manages the read and write requests sent to the Block Storage service. It interacts with the cinder-scheduler and cinder-backup processes and the storage providers, using respective drivers.
- **cinder-scheduler daemon** - In multi-node deployments, it selects the optimal storage node on which the volume will be created. This process is similar to the nova-scheduler.
- **cinder-backup daemon** - Provides the ability to back up volumes. It can interact with multiple backup storage providers using drivers, in a similar way as the cinder-volume process.

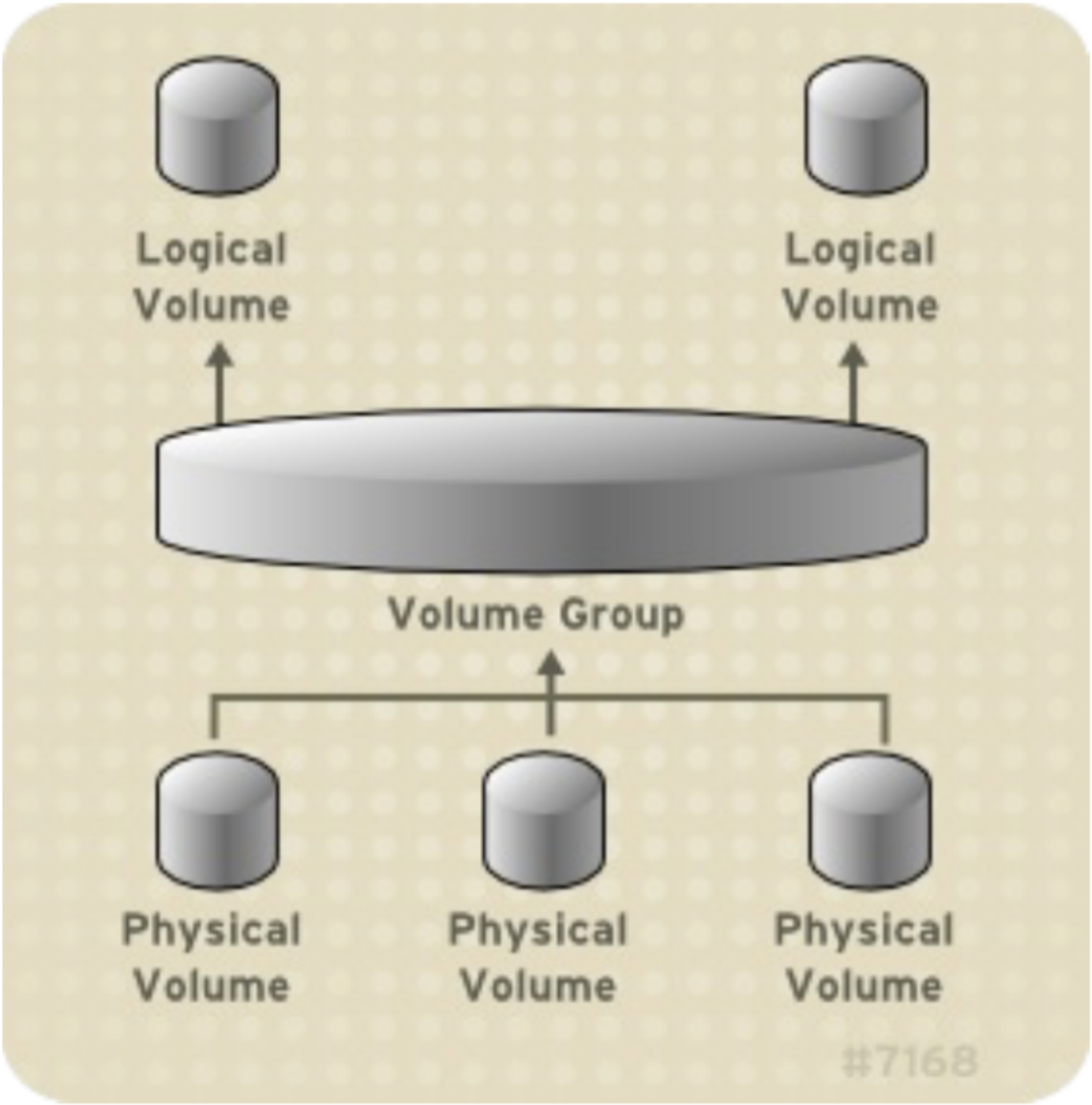


Figure 3.3: Logical volume architecture [Source: [**?**]]

- **Messaging queue** - Routes information between the processes within the OpenStack Block Storage service. This thesis uses a centralised RabbitMQ service, running on the controller node.
  [5]

3.4.6 OpenStack Dashboard Service

The OpenStack Dashboard service enables you to manage the OpenStack resources and services. It is used by administrators and the end user. It provides a web-based interface that communicates through the OpenStack APIs. It also allows customising the brand of the dashboard to match the cloud provider's needs. [5]

Handling of User Session Data

The Dashboard service supports several session backends to handle the user session data:

- **Local Memory Cache** - The quickest and easiest session backend, which does not require external dependencies and is the easy to set up. However, it does not support shared storage across processes or workers, and does not offer data persistency after a process terminates. This is the reason why it is not recommended for production use.
- **Memcached** - An external caching service. Supports shared storage and offers data persistency after process or worker terminates. It is extremely fast and efficient cache backend. It requires the memcached service to be running and accessible and a python memcached module installed.
- **Database** - A database can be also used as a caching backend. It is scalable, can be highly-available and offers data persistency. However, it is slow in comparison to other caching methods.
- **Cached Database** - This is a hybrid setting using database and caching infrastructure together.
- **Cookies** - Stores data in a cookie in the user's browser. It supports a cryptographic signing to ensure that the data has not been changed during transport. It should be noted, that signing is not the same as encryption, so that the data are still readable by potential attacker.
  [8]

Chapter 4

Existing Methods of Automated OpenStack Deployment

4.1 Packstack

Packstack is a command line installation utility that support deployment of OpenStack on existing server using SSH connection. The installation can be configured interactively or by a configuration file called answer file. [16]

It is suitable for deploying proof-of-concept installations. It supports these two basic types of deployment:

- **An all-on-one installation** - All services are installed on a single physical host that would run all controller services and the virtual machines.
- **Multiple nodes** - Using several hosts to run the installations, where there is a single controller node running the controller services, and one or more compute nodes that would run the virtual machines.
  [12]

However, packstack is not suitable for production deployments. This is mainly because it makes many assumptions in the configuration in order to simplify the installation process. It can not deploy the services in a high availability (HA) mode or using load balancers. It also does not support advanced networking configuration, which might be required by more complex setups. [11]

4.2 OpenStack-Ansible

OpenStack-Ansible is an official OpenStack project, still under development. The goal of this project is to be able to deploy OpenStack cloud in a production environment directly from source code. It is focused on Ubuntu Linux and the OpenStack components are installed into Linux Containers (LXC). [9] [10]

Chapter 5

Implementation Design

5.1 Design of the Physical Architecture

5.1.1 Selecting OpenStack Services

The first decision when designing an OpenStack cloud deployment is about the services we need to have running.

For this thesis, I have chosen these five core services:

- OpenStack Identity (Keystone)
- OpenStack Image (Glance)
- OpenStack Compute (Nova)
- OpenStack Block Storage (Cinder)
- OpenStack Dashboard (Horizon)

These services will be also required:

- SQL Database
- AMQP message broker

5.1.2 Selecting Specific Implementations

The second decision is about technology. The most important is to choose a specific version of OpenStack. The OpenStack release chosen for this thesis will be Liberty. This is mostly because it is the latest release at the time of writing the thesis.

I also need to choose an operating system platform that will run on the physical host machines. Choosing the right Linux distribution is important as there might be big differences in terms of available packages, stability, the lifecycle, and an option of commercial support. The target platform of deployment will be CentOS 7.2. CentOS is free and open source operating system based on the Red Hat Enterprise Linux platform. It uses the RPM packaging model and the OpenStack Liberty packages are included in the repositories. The system is stable, has a long lifecycle and potential migration to to a commercially supported environment would be easy, as it is based on the Red Hat Enterprise Linux platform.

OpenStack also needs an SQL database to store the state of the services, and it also might me used as a storage for the Identity service. In this deployment, it will be used

in this way. I have chosen MariaDB because it is open source, widely used and handles memory well.

The next component is an AMQP broker. It is used by processes within the OpenStack service for communication, and it also might be used as a short-term storage by some networking plugins. For this particular deployment, I will use RabbitMQ because it is an open source solution and it is widely used in production environments. It is also well-supported by the OpenStack itself.

The last decision will be about storage backend for the OpenStack Block Storage service. I will use a local disk and an LVM technology. LVM is a robust and stable open source solution for storage that runs on commodity hardware, which is important especially for this testing environment.

5.1.3 Physical architecture

Networks

I have used three different networks in my architecture and each one have different purpose in the whole system. Their descriptions are as follows:

- **Utility Network** - This network is used the datacenter administrators to maintain the physical hosts. In this example it will also be used by Ansible to deploy the OpenStack and all other services.
- **Management Network** - This network is used by the OpenStack services to communicate with each other. It will also provide connectivity for the services to the MySQL database and the RabbitMQ Message Queue. All Keystone endpoints based on hostnames will also use this network. It will be also used to connect storage from Storage1 host to virtual machines running on the Compute1 host.
- **VM Network** - This network is used to provide outside connectivity to the virtual machines running on the compute node.

It is also important to note that the eth2 interfaces on Controller and Compute1 hosts will use special configuration without an IP address and will be attached to virtual network bridges. Their specific configuration will be mentioned later in the text.

5.2 Design of the Ansible Playbook

5.2.1 Ansible Roles

The services will be installed on the target hosts using Ansible roles. Roles offer flexibility in a way, that each role can be installed separately on a given host, or, if needed, on multiple hosts. These roles can be then reused to deploy a different physical architectures.

All roles use handlers to restart services in case of a configuration change. All roles are also designed to be idempotent.

Role `sql-database`

The role sql-database will deploy a MariaDB database on the target host. It will set the root password accordingly and will also remove the anonymous user as well as the test database.

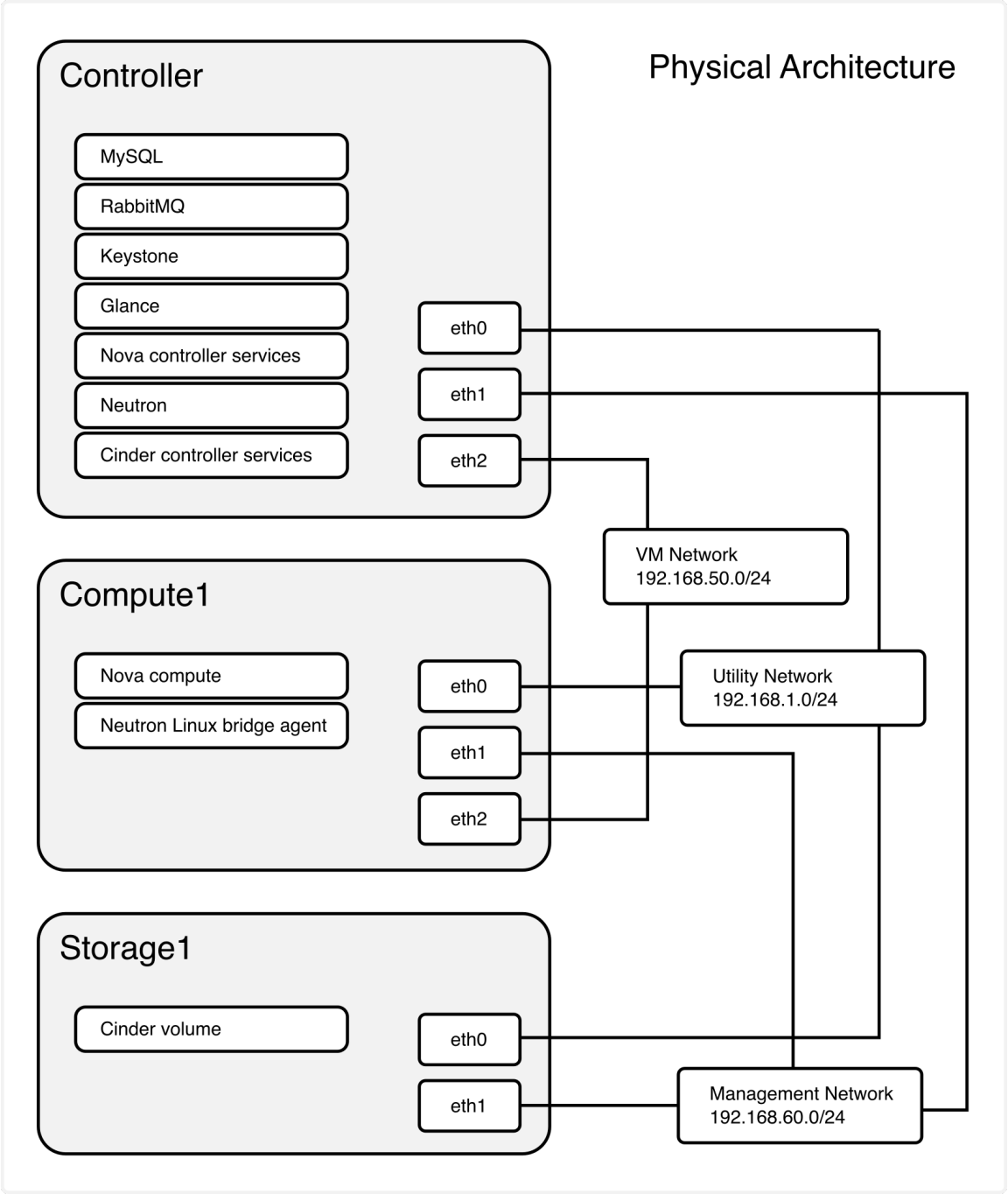


Figure 5.1: OpenStack physical architecture design

The following packages will be installed:

- `mariadb`
- `mariadb-server`
- `MySQL-python`

And the following MariaDB service will be enabled and started:

- `mariadb`

Role `rabbit`

The rabbit role will deploy a RabbitMQ message broker on the target host. It will also create a new user needed for the OpenStack deployment.

The following package will be installed:

- `rabbitmq-server`

And the following RabbitMQ service will be enabled and started:

- `rabbitmq-server`

Roles `controller-basic` and `compute-basic`

The roles controller-basic and compute-basic enable the OpenStack Liberty repository for CentOS and install the SELinux package. These roles are the first that should be run before any other openstack roles.

These roles will install these two packages:

- `python-openstackclient`
- `openstack-selinux`

Role `keystone`

The keystone role will install the OpenStack Identity service, codename Keystone, on the target host. It requires an SQL database to be running on the network. The example in this thesis will run the database on the controller node.

The role will create a database for the Keystone service called keystone.

This role will install the following packages:

- `openstack-keystone`
- `httpd`
- `mod_wsgi`
- `memcached`
- `python-memcached`
- `python-keystoneclient`

And will enable and start the httpd service on the target host.

Role `glance`

The glance role will install the OpenStack Image service, codename Glance, on the target host. It requires an SQL database, and the OpenStack Identity service to be running on the network. The example in this thesis will run both on the controller node.

The role will create a database for the Glance service called glance, and registers the Glance service and creates endpoints in the Keystone service.

Glance supports several backends for storing images. This role uses local filesystem to do so. Metadata about the images will be stored in the SQL database running on the controller node.

This role will install the following packages:

- `openstack-glance`
- `python-glance`
- `python-glanceclient`

And the two following services will be enabled and started:

- `openstack-glance-api`
- `openstack-glance-registry`

Role `nova-controller`

The nova-controller role will install some parts of the OpenStack Compute service, codename Nova, on the target host. It requires an SQL database, RabbitMQ message bus, and the OpenStack Identity service to be running on the network. The example in this thesis will run all these services on the controller node.

The role will create a database for the Nova service called nova, and registers the Nova service and creates endpoints in the Keystone service.

Besides the standard configurations like setting hostname, configuration of database, and message bus access, the Nova service will be configured to use the OpenStack Networking (as opposed to legacy Nova networking) with the Linux bridge driver.

This role will install the following packages:

- `openstack-nova-api`
- `openstack-nova-cert`
- `openstack-nova-conductor`
- `openstack-nova-console`
- `openstack-nova-novncproxy`
- `openstack-nova-scheduler`
- `python-novaclient`

And the following services will be enabled and started:

- `openstack-nova-api`

- `openstack-nova-cert`
- `openstack-nova-consoleauth`
- `openstack-nova-scheduler`
- `openstack-nova-conductor`
- `openstack-nova-novncproxy`

Role `nova-compute`

The nova-compute role will install the nova-compute process of the OpenStack Compute service, codename Nova, on the target host. It requires the RabbitMQ message bus to be running on the network. The example in this thesis will run it on the controller node.

The role will configure an access to the necessary Nova processes installed by the nova-controller role, and will also configure Nova to use the OpenStack Networking with the Linux bridge driver.

It will also configure the hypervisor. This role uses the libvirt provider using QEMU. This is because VirtualBox, which is used for testing the deployment, does not support nested virtualisation, so KVM can not be used.

This role will install these two packages:

- `openstack-nova-compute`
- `sysfsutils`

And will start and enable the two following services:

- `libvirtd`
- `openstack-nova-compute`

Role `neutron-controller`

The neutron-controller role will install the OpenStack Networking service, codename Neutron, on the target host. It requires an SQL database, RabbitMQ message bus, and the OpenStack Identity service to be running on the network. The example in this thesis will run all these services on the controller node. This role also requires the nova-compute role to be run on the same host before.

The role will create a database for the Neutron service called neutron, and registers the Neutron service and creates endpoints in the Keystone service.

The OpenStack Networking service uses plug-ins and agents for the actual networking functionality. This role will use:

- `Modular Layer 2 (ML2) plugin`
- `Linux bridge agent`
- `Layer-3 agent`
- `DHCP agent`

This setup will allow to create tenant networks as well as public provider networks.

The ML2 plugin will be configured to use the Linux bridge technology and VXLAN to create the tenant networks.

The Linux bridge agent will be configured to use VXLAN for the tenant networks and iptables firewall driver to manage security groups.

The layer-3 agent will be configured to use the Linux bridge driver and to support multiple external networks.

The DHCP agent will be also configured to use the Linux bridge driver and the MTU is set to 1450 bytes. This is because the VXLAN includes additional packet header and virtual machines running in the cloud use the default MTU of 1500 bytes. Using this settings, the virtual machines will use the smaller MTU, which would allow space for the additional header.

Finally, the Nova service will be configured to use the OpenStack Networking service, installed by this role.

This role will install the following packages:

- `openstack-neutron`
- `openstack-neutron-ml2`
- `openstack-neutron-linuxbridge`
- `python-neutronclient`
- `ebtables`
- `ipset`

It will restart this Nova service:

- `openstack-nova-api`

And the following services will be started and enabled:

- `neutron-server`
- `neutron-linuxbridge-agent`
- `neutron-dhcp-agent`
- `neutron-metadata-agent`
- `neutron-l3-agent`

Role `neutron-compute`

The neutron-compute role will install the Linux bridge agent of the OpenStack Networking service, codename Neutron, on the target host. It requires the RabbitMQ message bus to be running on the network. The example in this thesis will run it on the controller node. It also require the role nova-compute to be run on the same host before.

This role will configure the Linux bridge agent to use the correct network interface as a public interface, enables VXLAN for the tenant networks, and configures iptables as a firewall driver to manage security groups.

It will also configure the Nova service to use the OpenStack Networking.

This role will install the following packages:

- `openstack-neutron`
- `openstack-neutron-linuxbridge`
- `ebtables`
- `ipset`

It will restart this Nova service:

- `openstack-nova-compute`

And also the Linux bridge agent service will be started and enabled:

- `neutron-linuxbridge-agent`

Role `dashboard`

The dashboard role will install the OpenStack Dashboard service, codename Horizon, on the target host. The Dashboard service installed by this role uses the Apache httpd web server.

It will install the following packages:

- `openstack-dashboard`
- `httpd`
- `memcached`

And the following services will be started and enabled:

- `httpd`
- `memcached`

5.2.2 Applying Roles to the Hosts

The Ansible playbook will apply the roles to the target hosts to match the physical architecture, which has been designed in **??**

The list of nodes and playbooks that will be applied to them:

Controller node

- Role controller-basic
- Role rabbit
- Role sql-database
- Role dashboard
- Role glance
- Role keystone
- Role neutron-controller
- Role nova-controller
- Role cinder-controller

Compute node

- Role compute-basic
- Role neutron-compute
- Role nova-compute

Storage node

- Role storage-basic
- Role cinder-storage

Chapter 6

Implementation and Testing

6.1 Third-party Modules in the Playbook

The implementation requires two third-party modules for managing Keystone endpoints and services. Both modules have been created by Davide Guerri <davide.guerri@gmail.com>, licensed under Apache License, Version 2.0, and are included as part of the playbook.

6.2 Testing of the Deployment

6.2.1 Description of the Host Environment

The reference testing environment consists of three virtual machines with the following specification:

Controller

- CPUs: 2
- RAM: 2048 MB
- Disk: 30 GB

Compute

- CPUs: 4
- RAM: 6144 MB
- Disk: 30 GB

Storage

- CPUs: 1
- RAM: 1536 MB
- Disk1: 20 GB
- Disk2: 60 GB

These virtual machines were running on a laptop with the following configuration:

- CPU: 2.8 GHz Intel i7 4980HQ
- RAM: 16 GB
- SSD: PCIe 3.0 x4 8.0 GT/s (25.6 Gbit/s)

6.2.2 Using Ansible with Vagrant

Vagrant is a tool that manages virtual machines for development environment. It uses private key authentication and generates its own keys for the virtual machines.

First, Vagrant needs to be configured to use a single private key to authenticate to all three virtual machines. To achieve this, the following line needs to be put in the `Vagrantfile`:

```
config.ssh.insert_key = false
```

Vagrant routes SSH ports of the guest virtual machines to the localhost and uses different port for each. A command `vagrant ssh-config` will show the ports of each virtual machine. These ports need to be used in the `hosts` file. An example might look like this:

```
controller ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
compute1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
compute1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

The last step is to configure Ansible to use the correct user name and private key. This can be done by creating file called `ansible.cfg` with the following content:

```
[defaults]
hostfile = hosts
remote_user = vagrant
private_key_file = ~/.vagrant.d/insecure_private_key
host_key_checking = False
```

6.2.3 Running the Deployment

To use the Ansible Playbook, the following command needs to be issued:

```
ansible-playbook deploy-openstack.yml
```

The deployment of all three nodes takes approximately 7 minutes and requires internet connection.

Chapter 7

Conclusion

Bibliography

[1] *Introduction to OpenStack [online].* http://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html, 2015-10-22 [quoted 2015-10-22].

[2] *Conceptual architecture [online].* http://docs.openstack.org/liberty/install-guide-obs/common/get_started_conceptual_architecture.html, 2015-10-25 [quoted 2015-10-25].

[3] *How Ansible Works [online].* https://www.ansible.com/how-ansible-works, 2015 [quoted 2015-11-14].

[4] *OpenStack Administrator Guide [online].* http://docs.openstack.org/admin-guide/, 2015 [quoted 2015-11-27].

[5] *OpenStack Installation Guide for Red Hat Enterprise Linux and CentOS [online].* http://docs.openstack.org/liberty/install-guide-rdo/, 2015 [quoted 2015-11-27].

[6] *OpenStack Networking Guide [online].* http://docs.openstack.org/liberty/networking-guide/, 2015 [quoted 2015-11-27].

[7] *Red Hat Enterprise Linux 6 Logical Volume Manager Administration [online].* https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Logical_Volume_Manager_Administration/Red_Hat_Enterprise_Linux-6-Logical_Volume_Manager_Administration-en-US.pdf, 2015 [quoted 2015-11-27].

[8] *Deploying Horizon [online].* http://docs.openstack.org/developer/horizon/topics/deployment.html, 2015 [quoted 2015-11-28].

[9] *About OpenStack-Ansible [online].* http://docs.openstack.org/developer/openstack-ansible/install-guide/overview-osa.html, 2015 [quoted 2015-12-09].

[10] *Ansible playbooks for deploying OpenStack [online].* https://github.com/openstack/openstack-ansible, 2015 [quoted 2015-12-09].

[11] *Deploying OpenStack: Proof-of-Concept Environments (PackStack) [online].* https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_OpenStack_Platform/5/html/Getting_Started_Guide/The_PackStack_Deployment_Utility1.html, 2015 [quoted 2015-12-09].

[12] *Packstack quickstart: Proof of concept for single node [online].* https://www.rdoproject.org/install/quickstart/, 2015 [quoted 2015-12-09].

[13] Inc. Ansible. *Ansible Documentation [online].* http://docs.ansible.com/ansible/index.html, 2015 [quoted 2015-11-14].

[14] Tom Fifield, Diane Fleming, Anne Gentle, Lorin Hochstein, Jonathan Proulx, Everett Toews, and Joe Topjian. *OpenStack Operations Guide.* O'Reilly Media, 2014.

[15] Lorin Hochstein. *Ansible: Up and Running.* O'Reilly Media, 2014.

[16] Vazquez A. Taylor F., Mahroua R. *Red Hat Enterprise Linux OpenStack Platform 6.0 CL210, Red Hat OpenStack Administration.* Red Hat, 2015. Edition 2 20150415.

Appendices

List of Appendices