

# Chapter 12: Scaling Out KVM with OpenStack

Being able to virtualize a machine is a big thing, but sometimes, just virtualization is not enough. The problem is how to give individual users tools so that they can virtualize whatever they need, when they need it. If we combine that user-centric approach with virtualization, we are going to end up with a system that needs to be able to do two things: it should be able to connect to KVM as a virtualization mechanism (and not only KVM) and enable users to get their virtual machines running and automatically configured in a self-provisioning environment that's available through a web browser. OpenStack adds one more thing to this since it is completely free and based entirely on open source technologies. Provisioning such a system is a big problem due to its complexity, and in this chapter, we are going to show you – or to be more precise, point you – in the right direction regarding whether you need a system like this.

In this chapter, we will cover the following topics:

- Introduction to OpenStack
- Software-defined networking
- OpenStack components
- Additional OpenStack use cases
- Provisioning the OpenStack environment
- Integrating OpenStack with Ansible

- Let's get started!

# Introduction to OpenStack

In its own words, **OpenStack** is a cloud operating system that is used to control a large number of different resources in order to provide all the essential services for **Infrastructure-as-a-Service (IaaS)** and **orchestration**.

But what does this mean? OpenStack is designed to completely control all the resources that are in the data center, and to provide both central management and direct control over anything that can be used to deploy both its own and third-party services. Basically, for every service that we mention in this book, there is a place in the whole OpenStack landscape where that service is or can be used.

OpenStack itself consists of several different interconnected services or service parts, each with its own set of functionalities, and each with its own API that enables full control of the service. In this part of this book, we will try to explain what different parts of OpenStack do, how they interconnect, what services they provide, and how to use those services to our advantage.

The reason OpenStack exists is because there was the need for an open source cloud computing platform that would enable creating public and private clouds that are independent of any commercial cloud platform. All parts of OpenStack are open source and were released under the Apache License 2.0. The software was

created by a large, mixed group of individuals and large cloud providers. Interestingly, the first major release was the result of NASA (a US government agency) and Rackspace Technology (a large US hosting company) joining their internal storage and computing infrastructure solutions. These releases were later designated with the names Nova and Swift, and we will cover them in more detail later.

The first thing you will notice about OpenStack is its services since there is no single *OpenStack* service but an actual stack of services. The name *OpenStack* comes directly from this concept because it correctly identifies OpenStack as an open source component that acts as services that are, in turn, grouped into functional sets.

Once we understand that we are talking about autonomous services, we also need to understand that services in OpenStack are grouped by their function, and that some functions have more than one specialized service under them. We will try to cover as much as possible about different services in this chapter, but there are simply too many of them to even mention all of them here. All the documentation and all the whitepapers can be found at <http://openstack.org>, and we strongly suggest that you consult it for anything not mentioned here, and even for things that we mention but that could have changed by the time you read this.

The last thing we need to clarify is the naming – every service in OpenStack has its project name

and is referred to by that name in the documentation. This might, at first glance, look confusing since some of the names are completely unrelated to the specific function a particular service has in the whole project, but using names instead of official designators for a function is far easier once you start using OpenStack. Take, for example, Swift. Swift's full name is *OpenStack Object Store*, but this is rarely mentioned in the documentation or its implementation. The same goes for other services or *projects* under OpenStack, such as Nova, Ironi, Neutron, Keystone, and over 20 other different services.

If you step away from OpenStack for a second, then you need to consider what cloud services are all about. The cloud is all about scaling – in terms of compute resources, storage, network, APIs – whatever. But, as always in life, as you scale things, you're going to run into problems. And these problems have their own *names* and *solutions*. So, let's discuss these problems for a minute.

The basic problems for cloud provider scalability can be divided into three groups of problems that need to be solved at scale:

- **Compute problems** (Compute = CPU + memory power): These problems are pretty straightforward to solve – if you need more CPU and memory power, you buy more servers, which, by design, means more CPU and memory. If you need a quality of service/**service-level agreement (SLA)** type of concept, we can introduce a concept such as

compute resource pools so that we can slice the compute *pie* according to our needs and divide those resources between our clients. It doesn't matter whether our client is just a private person or a company buying into cloud services. In cloud technologies, we call our clients *tenants*.

- **Storage problems:** As you scale your cloud environments, things become really messy in terms of storage capacity, management, monitoring and – especially – performance. The performance side of that problem has a couple of most commonly used variables – read and write throughput and read and write IOPS. When you grow your environment from 100 hosts to 1,000 hosts or more, performance bottlenecks are going to become a major issue that will be difficult to tackle without proper concepts. So, the storage problem can be solved by adding additional storage devices and capacity, but it's much more involved than the compute problem as it needs much more configuration and money. Remember, every virtual machine has a statistical influence on other virtual machines' performance, and the more virtual machines you have, the greater this entropy is. This is the most difficult process to manage in storage infrastructure.
- **Network problems:** As the cloud infrastructure grows, you need thousands and thousands of isolated networks so that the network traffic of tenant **A** can't communicate with the network traffic of tenant **B**. At the same time, you still need to offer a capability where you can

have multiple networks (usually implemented via VLANs in non-cloud infrastructures) per tenant and routing between these networks, if that's what the tenant needs.

This network problem is a scalability problem based on technology, as the technology behind VLAN was standardized years before the number of VLANs could become a scalability problem.

Let's continue our journey through OpenStack by explaining the most fundamental subject of cloud environments, which is scaling cloud networking via **software-defined networking (SDN)**. The reason for this is really simple – without SDN concepts, the cloud wouldn't really be scalable enough for customers to be happy, and that would be a complete showstopper. So, buckle up your seatbelts and let's do an SDN primer.

## Software-defined networking

One of the straightforward stories about the cloud – at least on the face of it – should have been the story about cloud networking. In order to understand how simple this story should've been, we only need to look at one number, and that number is the **virtual LAN (VLAN ID)** number. As you might already be aware, by using VLANs, network administrators have a chance to divide a physical network into separate logical networks. Bearing in mind that the VLAN part of

the Ethernet header can have up to 12 bits, the maximum number of these logically isolated networks is 4,096. Usually, the first and last VLANs are reserved (0 and 4095), as is **VLAN 1**.

So, basically, we're left with 4,093 separate logical networks in a real-life scenario, which is probably more than enough for the internal infrastructure of any given company. However, this is nowhere near enough for public cloud providers. The same problem applies to public cloud providers that use hybrid-cloud types of services to – for example – extend their compute power to the cloud.

So, let's focus on this network problem for a bit. Realistically, if we look at this problem from the cloud user perspective, data privacy is of utmost importance to us. If we look at this problem from the cloud provider perspective, then we want our network isolation problem to be a non-issue for our tenants. This is what cloud services are all about at a more basic level – no matter what the background complexity in terms of technology is, users have to be able to access all of the necessary services in as user-friendly a way as possible. Let's explain this by using an example.

What happens if we have 5,000 different clients (tenants) in our public cloud environment? What happens if every tenant needs to have five or more logical networks? We quickly realize that we have a big problem as cloud environments need to be separated, isolated, and fenced. They need to be separated from one another at a network level for security and privacy reasons.

However, they also need to be routable, if a tenant needs that kind of service. On top of that, we need the ability to scale so that situations in which we need more than 5,000 or 50,000 isolated networks don't bother us. And, going back to our previous point – roughly 4,000 VLANs just isn't going to cut it.

There's a reason why we said that this should have been a straightforward story. The engineers among us see these situations in black and white – we focus on a problem and try to come to a solution. And the solution seems rather simple – we need to extend the 12-bit VLAN ID field so that we can have more available logical networks. How difficult can that be?

As it turns out, very difficult. If history teaches us anything, it's that various different interests, companies, and technologies compete for years for that *top dog* status in anything in terms of IT technology. Just think of the good old days of DVD+R, DVD-R, DVD+RW, DVD-RW, DVD-RAM, and so on. To simplify things a bit, the same thing happened here when the initial standards for cloud networking were introduced. We usually call these network technologies cloud overlay network technologies. These technologies are the basis for SDN, the principle that describes the way cloud networking works at a global, centralized management level. There are multiple standards on the market to solve this problem – VXLAN, GRE, STT, NVGRE, NVO3, and more.

Realistically, there's no need to break them all down one by one. We are going to take a simpler



route – we're going to describe one of them that's the most valuable for us in the context of today (**VXLAN**) and then move on to something that's considered to be a *unified* standard of tomorrow (**GENEVE**).

First, let's define what an overlay network is. When we're talking about overlay networks, we're talking about networks that are built on top of another network in the same infrastructure. The idea behind an overlay network is simple – we need to disentangle the physical part of the network from the logical part of the network. If we want to do that in absolute terms (configure everything without spending massive amounts of time in the CLI to configure physical switches, routers, and so on), we can do that as well. If we don't want to do it that way and we still want to work directly with our physical network environment, we need to add a layer of programmability to the overall scheme. Then, if we want to, we can interact with our physical devices and push network configuration to them for a more top-to-bottom approach. If we do things this way, we'll need a bit more support from our hardware devices in terms of capability and compatibility.

Now that we've described what network overlay is, let's talk about VXLAN, one of the most prominent overlay network standards. It also serves as a basis for developing some other network overlay standards (such as GENEVE), so – as you might imagine – it's very important to understand how it works.

## Understanding VXLAN

Let's start with the confusing part. VXLAN (IETF RFC 7348) is an extensible overlay network standard that enables us to aggregate and tunnel multiple Layer 2 networks across Layer 3 networks. How does it do that? By encapsulating a Layer 2 packet inside a Layer 3 packet. In terms of transport protocol, it uses UDP, by default on port **4789** (more about that in just a bit). In terms of special requests for VXLAN implementation – as long as your physical network supports MTU 1600, you can implement VXLAN as a cloud overlay solution easily. Almost all the switches you can buy (except for the cheap home switches, but we're talking about enterprises here) support jumbo frames, which means that we can use MTU 9000 and be done with it.

From the standpoint of encapsulation, let's see what it looks like:

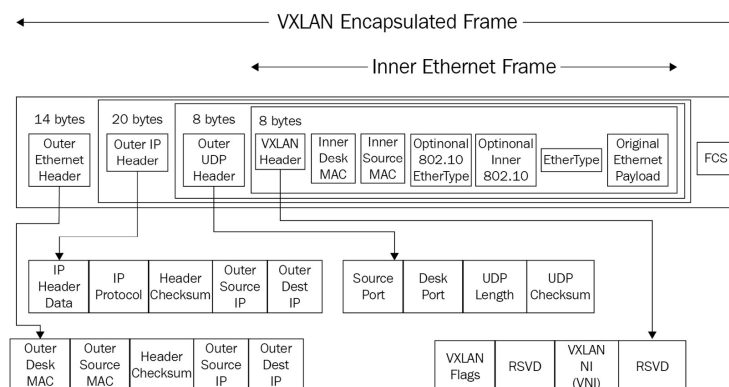


Figure 12.1 – VXLAN frame encapsulation

In more simplistic terms, VXLANs use tunneling between two VXLAN endpoints (called VTEPs; that is, VXLAN tunneling endpoints) that check **VXLAN network identifiers (VNIs)** so that they can decide which packets go where.

If this seems complicated, then don't worry – we can simplify this. From the perspective of VXLAN, a VNI is the same thing as a VLAN ID is to VLAN. It's a unique network identifier. The difference is just the size – the VNI field has 24 bits, compared to VLAN's 12. That means that we have  $2^{24}$  VNIs compared to VLAN's  $2^{12}$ . So, VXLANs – in terms of network isolation – are VLANs squared.

*Why does VXLAN use UDP?*

*When designing overlay networks, what you usually want to do is reduce latency as much as possible. Also, you don't want to introduce any kind of overhead. When you consider these two basic design principles and couple that with the fact that VXLAN tunnels Layer 2 traffic inside Layer 3 (whatever the traffic is – unicast, multicast, broadcast), that literally means we should use UDP. There's no way around the fact that TCP's two methods – three-way handshakes and retransmissions – would get in the way of these basic design principles. In the simplest of terms, TCP would be too complicated for VXLAN as it would mean too much overhead and latency at scale.*

In terms of VTEPs, just imagine them as two interfaces (implemented in software or hardware) that can encapsulate and decapsulate traffic based on VNIs. From a technology standpoint, VTEPs map various tenant's virtual machines and devices to VXLAN segments (VXLAN-backed isolated networks), perform package inspection, and encapsulate/decapsulate network traffic

based on VNIs. Let's describe this communication with the help of the following diagram:

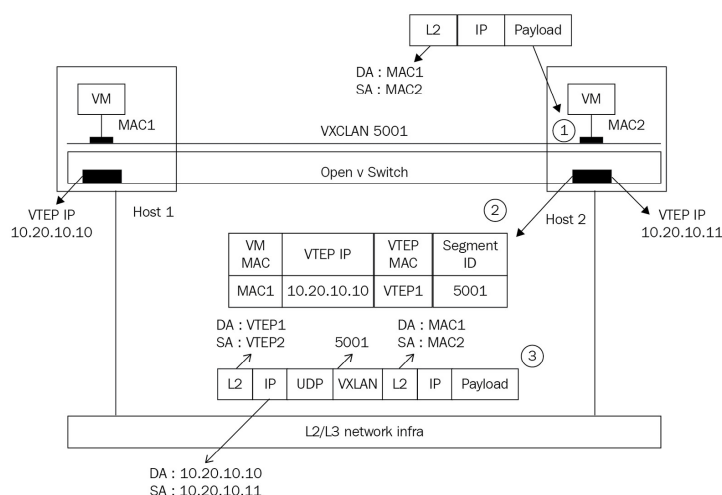


Figure 12.2 – VTEPs in unicast mode

In our open source-based cloud infrastructure, we're going to implement cloud overlay networks by using OpenStack Neutron or Open vSwitch, a free, open source distributed switch that supports almost all network protocols that you could possibly think of, including the already mentioned VXLAN, STT, GENEVE, and GRE overlay networks.

Also, there's a kind of gentleman's agreement in place in cloud networking regarding not using VXLANs from **1-4999** in most use cases. The reason for this is simple – because we still want to have our VLANs with their reserved range of **0-4095** in a way that is simple and not error-prone. In other words, by design, we leave network IDs **0-4095** for VLANs and start VXLANs with VNI 5000 so that it's really easy to differentiate between the two. Not using 5,000 VXLAN-backed networks out of 16.7 million VXLAN-backed networks isn't that much of a sacrifice for good engineering practices.

The simplicity, scalability, and extensibility of VXLAN also means more really useful usage models, such as the following:

- **Stretching Layer 2 across sites:** This is one of the most common problems regarding cloud networking, as we will describe shortly.
- **Layer 2 bridging:** Bridging a VLAN to a cloud overlay network (such as VXLAN) is *very* useful when onboarding our users to our cloud services as they can then just connect to our cloud network directly. Also, this usage model is heavily used when we want to physically insert a hardware device (for example, a physical database server or a physical appliance) into a VXLAN. If we didn't have Layer 2 bridging, imagine all the pain that we would have. All our customers running the Oracle Database Appliance would have no way to connect their physical servers to our cloud-based infrastructure.
- **Various offloading technologies:** These include load balancing, antivirus, vulnerability and antimalware scanning, firewall, IDS, IPS integration, and so on. All of these technologies enable us to have useful, secure environments with simple management concepts.

We mentioned that stretching Layer 2 across sites is a fundamental problem, so it's obvious that we need to discuss it. We'll do that next. Without a solution to this problem, you'd have very little chance of creating multiple data center cloud infrastructures efficiently.

## Stretching Layer 2 across sites

One of the most common sets of problems that cloud providers face is how to stretch their environment across sites or continents. In the past, when we didn't have concepts such as VXLAN, we were forced to use some kind of Layer 2 VPN or MPLS-based technologies. These types of services are really expensive, and sometimes, our service providers aren't exactly happy with our *give me MPLS* or *give me Layer 2 access* requests. They would be even less happy if we mentioned the word *multicast* in the same sentence, and this was a set of technical criteria that was *often* used in the past. So, having the capability to deliver Layer 2 over Layer 3 fundamentally changes that conversation. Basically, if you have the capability to create a Layer 3-based VPN between sites (which you can almost always do), you don't have to be bothered with that discussion at all. Also, that significantly reduces the price of these types of infrastructure connections.

Consider the following multicast-based example:

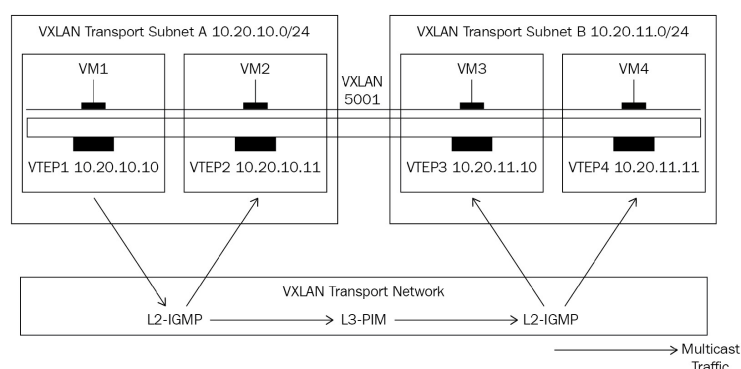


Figure 12.3 – Extending VXLAN segments across sites in multicast mode

Let's say that the left-hand side of this diagram is the first site and that the right-hand side of this

diagram is the second site. From the perspective of **VM1**, it doesn't really matter that **VM4** is in some other remote site as its segment (VXLAN 5001) *spans* across those sites. How? As long as the underlying hosts can communicate with each other over the VXLAN transport network (usually via the management network as well), the VTEPs from the first site can *talk* to the VTEPs from the second site. This means that virtual machines that are backed by VXLAN segments in one site can talk to the same VXLAN segments in the other site by using the aforementioned Layer 2-to-Layer 3 encapsulation. This is a really simple and elegant way to solve a complex and costly problem.

We mentioned that VXLAN, as a technology, served as a basis for developing some other standards, with the most important being GENEVE. As most manufacturers work toward GENEVE compatibility, VXLAN will slowly but surely disappear. Let's discuss what the purpose of the GENEVE protocol is and how it aims to become *the standard* for cloud overlay networking.

## Understanding GENEVE

The basic problem that we touched upon earlier is the fact that history kind of repeated itself in cloud overlay networks, as it did many times before. Different standards, different firmwares, and different manufacturers supporting one standard over another, where all of the standards are incredibly similar but still not compatible with each other. That's why VMware, Microsoft, Red Hat, and Intel proposed GENEVE,

a new cloud overlay standard that only defines the encapsulation data format, without interfering with the control planes of these technologies, which are fundamentally different. For example, VXLAN uses a 24-bit field width for VNI, while STT uses 64-bit. So, the GENEVE standard proposes no fixed field size as you can't possibly know what the future brings. Also, taking a look at the existing user base, we can still happily use our VXLANs as we don't believe that they will be influenced by future GENEVE deployments.

Let's see what the GENEVE header looks like:

GENEVE Header

V	Option Length	O	C	Reserved	Protocol Type
VNI					Reserved
Variable Length Options					

Figure 12.4 – GENEVE cloud overlay network header

The authors of GENEVE learned from some other standards (BGP, IS-IS, and LLDP) and decided that the key to doing things right is extensibility. This is why it was embraced by the Linux community in Open vSwitch and VMware in NSX-T. VXLAN is supported as the network overlay technology for **Hyper-V Network Virtualization (HNV)** since Windows Server 2016 as well. Overall, GENEVE and VXLAN seem to be two technologies that are surely here to stay – and both are supported nicely from the perspective of OpenStack.



Now that we've covered the most basic problem regarding the cloud – cloud networking – we can go back and discuss OpenStack. Specifically, our next subject is related to OpenStack components – from Nova through to Glance and then to Swift, and others. So, let's get started.

## OpenStack components

When OpenStack was first formed as a project, it was designed from two different services:

- A computing service that was designed to manage and run virtual machines themselves
- A storage service that was designed for large-scale object storage

These services are now called OpenStack Compute or *Nova*, and OpenStack Object Store or *Swift*. These services were later joined by *Glance* or the OpenStack Image service, which was designed to simplify working with disk images. Also, after our SDN primer, we need to discuss OpenStack Neutron, the **Network-as-a-Service (NaaS)** component of OpenStack.

The following diagram shows the components of OpenStack:

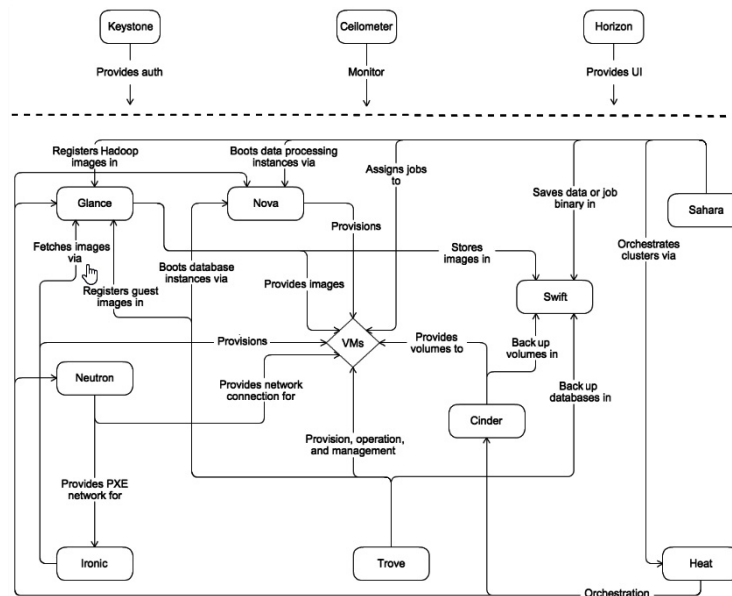


Figure 12.5 – Conceptual architecture of OpenStack (source: <https://docs.openstack.org/>)

We'll go through these in no particular order and will include additional services that are important. Let's start with **Swift**.

## Swift

The first service we need to talk about is Swift. For that purpose, we are going to grab the project's own definition from the OpenStack official documentation and parse it to try and explain what services are fulfilled by this project, **and what is it used for. The Swift website** (<https://docs.openstack.org/swift/latest/>) states the following:

*"Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply. It's built for scale and optimized for durability, availability, and concurrency across the entire dataset. Swift is ideal for storing unstructured data that can grow without bounds."*

Having read that, we need to point out quite a few things that may be completely new to you. First and foremost, we are talking about storing data in a particular way that is not common in computing unless you have used unstructured data stores. Unstructured does not mean that this way of storing data is lacking structure; in this context, it means that we are the ones that are defining the structure of the data, but the service itself does not care about our structure, instead relying on the concept of objects to store our data. One result of this is something that may also sound unusual at first, and that is that the data we store in Swift is not directly accessible through any filesystem, or any other way we are used to manipulating files through our machines. Instead, we are manipulating data as objects and we must use the API that is provided as part of Swift to get the data objects. Our data is stored in *blobs*, or objects, that the system itself just labels and stores to take care of availability and access speed. We are supposed to know what the internal structure of our data is and how to parse it. On the other hand, because of this approach, Swift can be amazingly fast with any amount of data and scales horizontally in a way that is almost impossible to achieve using normal, classic databases.

Another thing worth mentioning is that this service offers highly available, distributed, and *eventually consistent* storage. This means that, first and foremost, the priority is for the data to be distributed and highly available, which are two things that are important in the cloud.

Consistency comes after that but is eventually achieved. Once you come to use this service, you will understand what that means. In almost all usual scenarios where data is read and rarely written, it is nothing to even think about, but there are some cases where this can change the way we need to think about the way we go about delivering the service. The documentation states the following:

*"Because each replica in Object Storage functions independently and clients generally require only a simple majority of nodes to respond to consider an operation successful, transient failures such as network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems and concurrently perform operations in a manner that balances load across physical disks."*

We can roughly translate this. Let's say that you have a three-node Swift cluster. In such a scenario, a Swift object will become available to clients after the **PUT** operation has been confirmed to have been completed on at least two nodes. So, if your goal is to create a low-latency, synchronous storage replication with Swift, there are other solutions available for that.

Having put aside all the abstract promises regarding what Swift offers, let's go into more details. High availability and distribution are the direct result of using a concept of *zones* and having multiple copies of the same data written onto

multiple storage servers. Zones are nothing but a simple way of logically dividing the storage resources we have at our disposal and deciding on what kind of isolation we are ready to provide, as well as what kind of redundancy we need. We can group servers by the server itself, by the rack, by sets of servers across a Datacenter, in groups across different Datacenters, and in any combination of those. Everything really depends on the amount of available resources and the data redundancy and availability we need and want, as well as, of course, the cost that will accompany our configuration.

Based on the resources we have, we are supposed to configure our storage system in terms of how many copies it will hold and how many zones we are prepared to use. A copy of a particular data object in Swift is referred to as a *replica*, and currently, the best practices call for at least three replicas across no less than five zones.

A zone can be a server or a set of servers, and if we configure everything correctly, losing any one zone should have no impact on the availability or distribution of data. Since a zone can be as small as a server and as big as any number of data centers, the way we structure our zones has a huge impact on the way the system reacts to any failures and changes. The same goes for replicas. In the recommended scenario, configuration has a smaller number of replicas than the number of zones, so only some of the zones will hold some of these replicas. This means the sys-

tem must balance the way data is written in order to evenly distribute both the data and the load, including both the writing and the reading load for the data. At the same time, the way we structure the zones will have an enormous impact on the cost – redundancy has a real cost in terms of server and storage hardware, and multiplying replicas and zones creates additional demands in regard to how much storage and computing power we need to allocate for our OpenStack installation. Being able to do this correctly is the biggest problem that a Datacenter architect has to solve.

Now, we need to go back to the concept of eventual consistency. Eventual consistency in this context means that data is going to be written to the Swift store and that objects are going to get updated, but the system will not be able to do a completely simultaneous write of all the data into all the copies (replicas) of the data across all zones. Swift will try to reconcile the differences as soon as possible and will be aware of these changes, so it serves new versions of the objects to whoever tries to read them. Scenarios where data is inconsistent due to a failure of some part of the system exist, but they are to be considered abnormal states of the system and need to be repaired rather than the system being designed to ignore them.

## Swift daemons

Next, we need to talk about the way Swift is designed in regard to its architecture. Data is managed through three separate logical daemons:

- **Swift-account** is used to manage a SQL database that contains all the accounts defined with the object storage service. Its main task is to read and write the data that all the other services need, primarily in order to validate and find appropriate authentication and other data.
- **Swift-container** is another database process, but it is used strictly to map data into containers, a logical structure similar to AWS *buckets*. This can include any number of objects that are grouped together.
- **Swift-object** manages mapping to actual objects, and it keeps track of the location and availability of the objects themselves.

All these daemons are just in charge of data and make sure that everything is both mapped and replicated correctly. Data is used by another layer in the architecture: the presentation layer.

When a user wants to use any data object, it first needs to authenticate via a token that can be either externally provided or created by an authentication system inside Swift. After that, the main process that orchestrates data retrieval is Swift-proxy, which handles communication with three daemons that deal with the data. Provided that the user presented a valid token, it gets the data object delivered to the user request.

This is just the briefest of overviews regarding how Swift works. In order to understand this, you need to not only read the documentation but also use some kind of system that will perform

low-level object retrieval and storage into and out of Swift.

Cloud services can't be scaled or used efficiently if we don't have orchestration services, which is why we need to discuss the next service on our list – **Nova**.

## Nova

Another important service or project is Nova – an orchestration service that is used for providing both provisioning and management for computing instances at a large scale. What it basically does is allow us to use an API structure to directly allocate, create, reconfigure, and delete or *destroy* virtual servers. The following is a diagram of a logical Nova service structure:

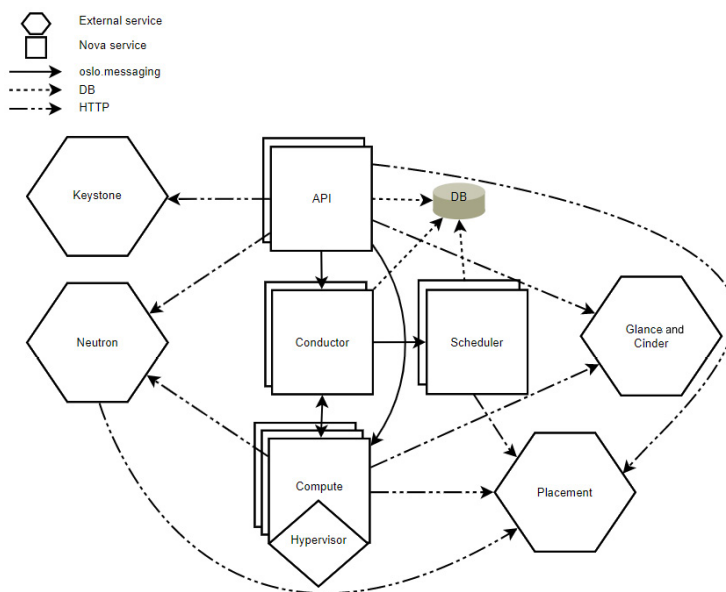


Figure 12.6 – Logical structure of the Nova service (openstack.org)

Most of Nova is a very complex distributed system written almost entirely in Python that consists of a number of working scripts that do the orchestration part and a gateway service that re-



ceives and carries through API calls. The API is also based on Python; it's a **Web Server Gateway Interface (WSGI)**-compatible application that handles calls. WSGI, in turn, is a standard that defines how a web application and a server should exchange data and commands. This means that, in theory, any system capable of using the WSGI standard can also establish communication with this service.

Aside from this multifaceted orchestration solution, there are two more services that are at the heart of Nova – the database and messaging queue. Neither of these is Python-based. We'll talk about messaging and databases first.

Almost all distributed systems must rely on queues to be able to perform their tasks. Messages need to be forwarded to a central place that will enable all daemons to do their tasks, and using the right messaging and queueing system is crucial for system speed and reliance. Nova currently uses RabbitMQ, a highly scalable and available system by itself. Using a production-ready system like this means that not only are there tools to debug the system itself, but there are a lot of reporting tools available for directly querying the messaging queue.

The main purpose of using a messaging queue is to completely decouple any clients from servers, and to provide asynchronous communication between different clients. There is a lot to be said on how the actual messaging works, but for this chapter, we will just refer you to the official documentation at

<https://docs.openstack.org/nova/latest/>, since we are not talking about a couple of functions on a server but an entirely independent software stack.

The database is in charge of holding all the state data for the tasks currently being performed, as well as enabling the API to return information about the current state of different parts of Nova.

All in all, the system consists of the following:

- **nova-api**: The daemon that is directly facing the user and is responsible for accepting, parsing, and working through all the user API requests. Almost all the documentation that refers to **nova-api** is actually referring to this daemon, sometimes calling it just *API*, *controller*, or *cloud controller*. We need to explain a little bit more about Nova in order to understand that calling nova-api a controller is wrong, but since there exists a class inside a daemon named **CloudController**, a lot of users confuse this daemon for the whole distributed system.  
nova-api is a powerful system since it can, by itself, process and sort out some API calls, getting the data from the database and working out what needs to be done. In a more common case, nova-api will just initiate a task and forward it in the form of messages to other daemons inside Nova.
- Another important daemon is the **scheduler**. Its main function is to go through the queue and determine when and where a particular

request should run. This sounds simple enough, but given the possible complexity of the system, this *where and when* can lead to extreme gains or losses in performance. In order to solve this, we can choose how the scheduler makes decisions regarding choosing the right place to perform requests. Users can choose either to write their own request or to use one of the predetermined ones.

If we are choosing the ones provided by Nova, we have three choices:

- a) **Simple scheduler** determines where the request will be run based on the load on the hosts – it will monitor all the hosts and try to allocate the one that has the least load in a particular slice of time.
- b) **Chance** is the default way of scheduling. As its name suggests, it's the simplest algorithm – a host is randomly chosen from the list and given the request.
- c) **Zone scheduling** will also randomly choose a host but will do so from within a zone.

Now, we will look at *workers*, daemons that actually perform requests. There are three of these – network, volume, and compute:

- **nova-network** is in charge of the network. It will perform whatever is given to it from the queue that is related to anything on the network and will create interfaces and rules as needed. It is also in charge of IP address allocation; it will allocate both fixed and dynamically assigned addresses and take care of both the external and internal networks. Instances

usually use one or more fixed IPs to enable management and connectivity, and these are usually local addresses. There are also floating addresses to enable connecting from the outside. This service has been obsolete since the OpenStack Newton release from 2016, although you can still use it in some legacy configurations.

- **nova-volume** handles storage volumes or, to be more precise, all the ways data storage can be connected to any instance. This includes standards such as iSCSI and AoE, which are targeted at encapsulating known common protocols, and providers such as Sheepdog, LeftHand, and RBD, which cover connections to open source and closed source storage systems such as CEPH or HP LeftHand.
- **nova-compute** is probably the easiest to describe – it is used to create and destroy new instances of virtual machines, as well as to update information about them in the database. Since this is a heavily distributed system, this also means that **nova-compute** must adapt itself to using different virtualization technologies and to completely different platforms. It also needs to be able to dynamically allocate and free resources. Primarily, it uses libvirt for its VM management, directly supporting KVM for creating and deleting new instances. This is the reason this chapter exists, since nova-compute using libvirt to start KVM machines is by far the most common way of configuring OpenStack, but support for different technologies extends a lot further. The libvirt interface

also supports Xen, QEMU, LXC, and **user mode Linux (UML)**, and through different APIs, nova-compute can support Citrix, XCP, VMware ESX/ESXi vSphere, and Microsoft Hyper-V. This enables Nova to control all the currently used enterprise virtualization solutions from one central API.

As a side note, **nova-conductor** is there to process requests that require any conversion regarding objects, resizing, and database/proxy access.

The next service on our list is **Glance** – a service that is very important for virtual machine deployment as we want to do this from images. Let's discuss Glance now.

## Glance

At first, having a separate service for cloud disk image management makes little sense, but when scaling any infrastructure, image management will become a problem that needs an API to be solved. Glance basically has this dual identity – it can be used to directly manipulate VM images and store them inside blobs of data, but at the same time it can be used to completely automatically orchestrate a lot of tasks when dealing with a huge number of images.

Glance is relatively simple in terms of its internal structure as it consists of an image information database, an image store that uses Swift (or a similar service), and an API that glues everything together. Database is sometimes called Registry,

and it basically gives information about a given image. Images themselves can be stored on different types of stores, either from Swift (as blobs) on HTTP servers or on a filesystem (such as NFS).

Glance is completely nonspecific about the type of image store it uses, so NFS is perfectly okay and makes implementing OpenStack a little bit easier, but when scaling OpenStack, both Swift and Amazon S3 can be used.

When thinking about the place in the big OpenStack puzzle that Glance belongs to, we could describe it as being the service that Nova uses to find and instantiate images. Glance itself uses Swift (or any other storage) to store images. Since we are dealing with multiple architectures, we need a lot of different supported file formats for images, and Glance does not disappoint. Every disk format that is supported by different virtualization engines is supported by Glance. This includes both unstructured formats such as **raw** and structured formats such as VHD, VMDK, **qcow2**, VDI ISO, and AMI. OVF – as an example of an image container – is also supported.

Glance probably has the simplest API of them all, enabling it to be used even from the command line using curl to query the server and JSON as the format of the messages.

We'll finish this section with a small note directly from the Nova documentation: it explicitly states that everything in OpenStack is designed to be horizontally scalable but that, at any time, there should be significantly more computing nodes

than any other type. This actually makes a lot of sense – computing nodes are the ones in charge of actually accepting and working on requests. The amount of storage nodes you'll need will depend on your usage scenario, and Glance's will inevitably depend on the capabilities and resources available to Swift.

The next service in line is **Horizon** – a *human-readable* GUI dashboard of OpenStack where we *consume* a lot of OpenStack visual information.

## Horizon

Having explained the core services that enable OpenStack to do what it does the way it does in some detail, we need to address the user interaction. In almost every paragraph in this chapter, we refer to APIs and scripting interfaces as a way to communicate and orchestrate OpenStack. While this is completely true and is the usual way of managing large-scale deployments, OpenStack also has a pretty useful interface that is available as a web service in a browser. The name of this project is Horizon, and its sole purpose is to provide a user with a way of interacting with all the services from one place, called the dashboard. Users can also reconfigure most, if not all, the things in the OpenStack installation, including security, networking, access rights, users, containers, volumes, and everything else that exists in the OpenStack installation.

Horizon also supports plugins and *pluggable panels*. There is an active plugin marketplace for Horizon that aims at extending its functionality

even further than it already has. If that's still not enough for your particular scenario, you can create your own plugins in Angular and get them to run in Horizon.

Pluggable panels are also a nice idea – without changing any defaults, a user or a group of users can change the way the dashboard looks and get more (or less) information presented to them. All of this requires a little bit of coding; changes are made in the config files, but the main thing is that the Horizon system itself supports such a customization model. You can find out more about the interface itself and the functions that are available to the user when we cover installing OpenStack and creating OpenStack instances in the *Provisioning the OpenStack environment* section.

As you are aware, networks don't really work all that well without name resolution, which is why OpenStack has a service called **Designate**. We'll briefly discuss Designate next.

## Designate

Every system that uses any kind of network must have at least some kind of name resolution service in the form of a local or remote DNS or a similar mechanism.

Designate is a service that tries to integrate the *DNSaaS* concept in OpenStack in one place.

When connected to Nova and Neutron, it will try to keep up-to-date records in regards to all the hosts and infrastructure details.



Another very important aspect of the cloud is how we manage identities. For that specific purpose, OpenStack has a service called **Keystone**. We'll discuss what it does next.

## Keystone

Identity management is a big thing in cloud computing, simply because when deploying a large-scale infrastructure, not only do you need a way to scale your resources, but you also need a way to scale user management. A simple list of users that can access a resource is not an option anymore, mainly because we are not talking about simple users anymore. Instead, we are talking about domains containing thousands of users separated by groups and by roles – we are talking about multiple ways of logging in and providing authentication and authorization. Of course, this also can span multiple standards for authentication, as well as multiple specialized systems.

For these reasons, user management is a separate project/service in OpenStack named Keystone.

Keystone supports simple user management and the creation of users, groups, and roles, but it also supports LDAP, OAuth, OpenID Connect, SAML, and SQL database authentication and has its own API that can support every possible scenario for user management. Keystone is in a world by itself, and in this book, we will treat it as a simple user provider. However, it can be much more and can require a lot of configuration, depending on the case. The good thing is

that, once installed, you will rarely need to think about this part of OpenStack.

The next service on our list is **Neutron**, the API/backend for (cloud) networking in OpenStack.

## Neutron

OpenStack Neutron is an API-based service that aims to provide a simple and extensible cloud network concept as a development of what used to be called a *Quantum* service <sup>[1]</sup><sub>SEP</sub> in older releases of OpenStack. Before this service, networking was managed by nova-network, which, as we mentioned, is a solution that's obsolete, with Neutron being the reason for this. Neutron integrates with some of the services that we've already discussed – Nova, Horizon, and Keystone. As a standalone concept, we can deploy Neutron to a separate server, which will then give us the ability to use the Neutron API. This is reminiscent of what VMware does in NSX with the NSX Controller concept.

When we deploy neutron-server, a web-based service that hosts the API connects to the Neutron plugin in the background so that we can introduce networking changes to our Neutron-managed cloud network. In terms of architecture, it has the following services:

- Database for persistent storage
- neutron-server
- External agents (plugins) and drivers

In terms of plugins, it has a *lot* of them, but here's a short list:

- Open vSwitch
- Cisco UCS/Nexus
- The Brocade Neutron plugin
- IBM SDN-VE
- VMware NSX
- Juniper OpenContrail
- Linux bridging
- ML2
- Many others

Most of these plugin names are logical, so you won't have any problems understanding what they do. But we'd like to mention one of these plugins specifically, which is the **Modular Layer 2 (ML2)** plugin.

By using the ML2 plugin, OpenStack Neutron can connect to various Layer 2 backends – VLAN, GRE, VXLAN, and so on. It also enables Neutron to go away from the Open vSwitch and Linux bridge plugins as its basic plugins (which are now obsolete). These plugins are considered to be too monolithic for Neutron's modular architecture, and ML2 has replaced them completely since the release of Havana (2013). ML2 today has many vendor-based plugins for integration. As shown by the preceding list, Arista, Cisco, Avaya, HP, IBM, Mellanox, and VMware all have ML2-based plugins for OpenStack.

In terms of network categories, Neutron supports two:

- **Provider networks:** Created by an OpenStack administrator, these are used for external connections on a physical level, which are usually backed by flat (untagged) or VLAN (802.1q tagged) concepts. These networks are shared since tenants use them to access their private infrastructure in hybrid cloud models or to access the internet. Also, these networks describe the way underlay and overlay networks interact, as well as their mappings.
- **Tenant networks, self-service networks, project networks:** These networks are created by users/tenants and their administrators so that they can connect their virtual resources and networks in whatever shape or form they need. These networks are isolated and usually backed by a network overlay such as GRE or VXLAN, as that's the whole purpose of tenant networks.

Tenant networks usually use some kind of SNAT mechanism to access external networks, and this service is usually implemented via virtual routers. The same concept is used in other cloud technologies such as VMware NSX-v and NSX-t, as well as Microsoft Hyper-V SDN technologies backed by Network Controller.

In terms of network types, Neutron supports multiple types:

- **Local:** Allows us to communicate within the same host.
- **Flat:** An untagged virtual network.
- **VLAN:** An 802.1Q VLAN tagged virtual network.

- **GRE, VXLAN, GENEVE:** Depending on the network overlay technologies, we select these network backends.

Now that we've covered OpenStack's usage models, ideas, and services, let's discuss additional ways in which OpenStack can be used. As you might imagine, OpenStack – being what it is – is highly capable of being used in many non-standard scenarios. We'll discuss these non-obvious scenarios next.

## Additional OpenStack use cases

OpenStack has a lot of really detailed documentation available at <https://docs.openstack.org>. One of the more useful topics is the architecture and design examples, which both explain the usage scenarios and the ideas behind how a particular scenario can be solved using the OpenStack infrastructure. We are going to talk a lot about two different edge cases when we deploy our test OpenStack, but some things need to be said about configuring and running an OpenStack installation.

OpenStack is a complex system that encompasses not only computing and storage but also a lot of networking and supporting infrastructure. You will first notice that when you realize that even the documentation is neatly divided into an administration, architecture, operations, security, and virtual machine image guide. Each of these subjects is practically a topic for a single book,

and a lot of things that guides cover are part experience, part best practice advice, and part assumptions based on best guesses.

There are a couple of things that are more or less common to all these use cases. First, when designing a cloud, you must try and get all the information about possible loads and your clients as soon as possible, even before a first server is booted. This will enable you to plan not only how many servers you need, but their location, the ratio of computing to storage nodes, the network topology, energy requirements, and all the other things that need to be thought through in order to create a working solution.

When deploying OpenStack, we are talking about a large-scale enterprise solution that is usually deployed for one of three reasons:

- *Testing and learning*: Maybe we need to learn how to configure a new installation, or we need to test a new computing node before we even go near production systems. For that reason, we need a small OpenStack environment, perhaps a single server that we can expand if there is a need for that. In practice, this system should be able to support probably a single user with a couple of instances. Those instances will usually not be the focus of your attention; they are going to be there just to enable you to explore all the other functionalities of the system. Deploying such a system is usually done the way we described in this chapter – using a readymade script that installs and

configures everything so that we can focus on the part we are actually working on.

- *We have a need for a staging or pre-production environment:* Usually, this means that we need to either support the production team so they have a safe environment to work in, or we are trying to keep a separate test environment for storing and running instances before they are pushed into production.

Having such an environment is definitively recommended, even if you haven't had it yet, since it enables you and your team to experiment without fear of breaking the production environment. The downside is that this installation requires an environment that has to have some resources available for the users and their instances. This means we are not going to be able to get away with using a single server. Instead, we will have to create a cloud that will be, at least in some parts, as powerful as the production environment. Deploying such an installation is basically the same as production deployment since once it comes online, this environment will, from your perspective, be just another system in production.

Even if we are calling it pre-production or test, if the system goes down, your users will inevitably call and complain. This is the same as what happens with the production environment; you will have to plan downtime, schedule upgrades, and try to keep it running as best as you can.

- *For production:* This one is demanding in another way – maintenance. When creating an actual production cloud environment, you will

need to design it well, and then carefully monitor the system to be able to respond to problems. Clouds are a flexible thing from the user's perspective since they offer scaling and easy configuration, but being a cloud administrator means that you need to enable these configuration changes by having spare resources ready. At the same time, you need to pay attention to your equipment, servers, storage, networking, and everything else to be able to spot problems before the users see them. Has a switch failed over? Are the computing nodes all running correctly? Have the disks degraded in performance due to a failure? Each of these things, in a carefully configured system, will have minimal to no impact on the users, but if we are not proactive in our approach, compounding errors can quickly bring the system down.

Having distinguished between a single server and a full install in two different scenarios, we are going to go through both. The single server will be done manually using scripts, while the multi-server will be done using Ansible playbooks.

Now that we've covered OpenStack in quite a bit of detail, it's time to start using it. Let's start with some small things (a small environment to test) in order to provision a regular OpenStack environment for production, and then discuss integrating OpenStack with Ansible. We'll revisit OpenStack in the next chapter, when we start discussing scaling out KVM to Amazon AWS.

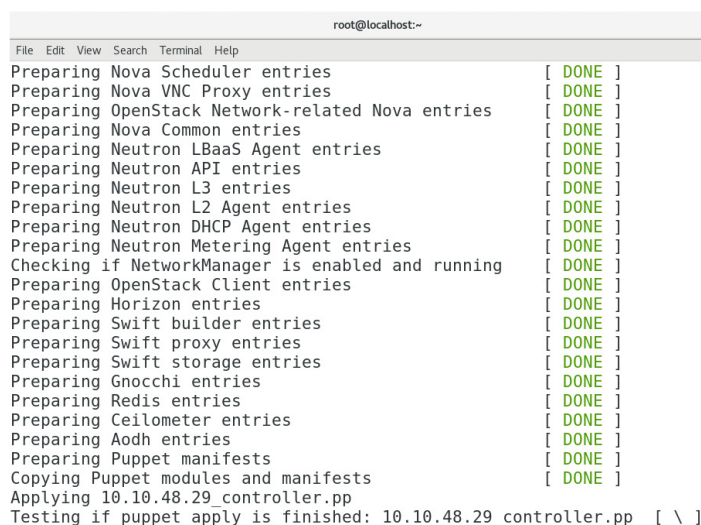


## Creating a Packstack demo environment for OpenStack

If you just need a **Proof of Concept (POC)**, there's a very easy way to install OpenStack. We are going to use **Packstack** as it's the simplest way to do this. By using Packstack installation on CentOS 7, you'll be able to configure OpenStack in 15 minutes or so. It all starts with a simple sequence of commands:

```
yum update -y
yum install -y centos-release-
openstack-train
yum update -y
yum install -y openstack-packstack
packstack --allinone
```

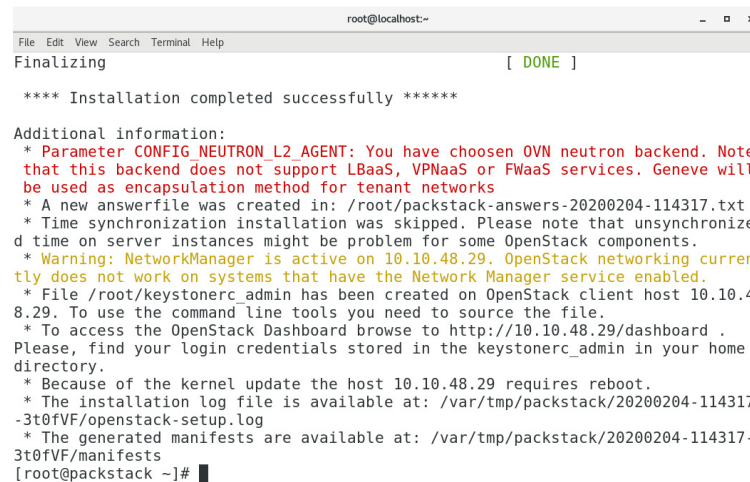
As the process goes through its various phases, you'll see various messages, such as the following, which are quite nice as you get to see what's happening in real time with a decent verbosity level:



```
root@localhost:~
File Edit View Search Terminal Help
Preparing Nova Scheduler entries [ DONE ]
Preparing Nova VNC Proxy entries [ DONE ]
Preparing OpenStack Network-related Nova entries [ DONE ]
Preparing Nova Common entries [ DONE ]
Preparing Neutron LBaaS Agent entries [ DONE ]
Preparing Neutron API entries [ DONE ]
Preparing Neutron L3 entries [ DONE ]
Preparing Neutron L2 Agent entries [ DONE ]
Preparing Neutron DHCP Agent entries [ DONE ]
Preparing Neutron Metering Agent entries [ DONE ]
Checking if NetworkManager is enabled and running [ DONE ]
Preparing OpenStack Client entries [ DONE ]
Preparing Horizon entries [ DONE ]
Preparing Swift builder entries [ DONE ]
Preparing Swift proxy entries [ DONE ]
Preparing Swift storage entries [ DONE ]
Preparing Gnocchi entries [ DONE ]
Preparing Redis entries [ DONE ]
Preparing Ceilometer entries [ DONE ]
Preparing Aodh entries [ DONE ]
Preparing Puppet manifests [ DONE ]
Copying Puppet modules and manifests [ DONE ]
Applying 10.10.48.29_controller.pp
Testing if puppet apply is finished: 10.10.48.29_controller.pp [ \ ]
```

Figure 12.7 – Appreciating Packstack's installation verbosity

After the installation is finished, you will get a report screen that looks similar to this:



```

root@localhost:~
File Edit View Search Terminal Help
Finalizing [ DONE ]

**** Installation completed successfully ****

Additional information:
* Parameter CONFIG_NEUTRON_L2_AGENT: You have chosen OVN neutron backend. Note
  that this backend does not support LBaaS, VPNaaS or FWaaS services. Geneve will
  be used as encapsulation method for tenant networks
* A new answerfile was created in: /root/packstack-answers-20200204-114317.txt
* Time synchronization installation was skipped. Please note that unsynchroniz-
  ed time on server instances might be problem for some OpenStack components.
* Warning: NetworkManager is active on 10.10.48.29. OpenStack networking curren-
  tly does not work on systems that have the Network Manager service enabled.
* File /root/keystonerc_admin has been created on OpenStack client host 10.10.4
  8.29. To use the command line tools you need to source the file.
* To access the OpenStack Dashboard browse to http://10.10.48.29/dashboard .
  Please, find your login credentials stored in the keystonerc_admin in your home
  directory.
* Because of the kernel update the host 10.10.48.29 requires reboot.
* The installation log file is available at: /var/tmp/packstack/20200204-114317
  -3t0fVF/openstack-setup.log
* The generated manifests are available at: /var/tmp/packstack/20200204-114317-
  3t0fVF/manifests
[root@packstack ~]#
  
```

Figure 12.8 – Successful Packstack installation

The installer has finished successfully, and it gives us a warning about **NetworkManager** and a kernel update, which means we need to restart our system. After the restart and checking the `/root/keystonerc_admin` file for our username and password, Packstack is alive and kicking and we can log in by using the URL mentioned in the previous screen's output (`http://IP_or_hostname_where_PackStack_is_deployed/dashboard`):

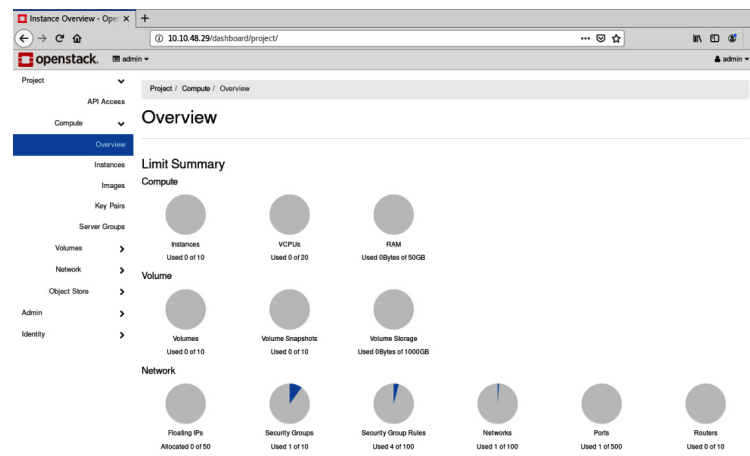


Figure 12.9 – Packstack UI

There's a bit of additional configuration that needs to be done, as noted in the Packstack docu-

mentation at

<https://wiki.openstack.org/wiki/Packstack>. If

you're going to use an external network, you need a static IP address without **NetworkManager**, and you probably want to either configure **firewalld** or stop it altogether. Other than that, you can start using this as your demo environment.

## Provisioning the OpenStack environment

One of the tasks that is going to be the simplest and, at the same time, the hardest when you need to create your first OpenStack configuration is going to be provisioning. There are basically two ways you can go with this: one is to install services one at a time in a carefully prepared hardware configuration, while the other is to just use a *single server install* guide from the OpenStack site and create a single machine that will serve as your test bed. In this chapter, everything we do is created in such an instance, but before we learn how to install the system, we need to understand the differences.

OpenStack is a cloud operating system, and its main idea is to enable us to use multiple servers and other devices to create a coherent, easily configured cloud that can be managed from a central point, either through an API or through a web server. The size and type of the OpenStack deployment can be from one server running everything, to thousands of servers and storage units integrated across several Datacenters.

OpenStack does not have a problem with large-scale deployment; the only real limiting factor is usually the cost and other requirements for the environment we are trying to create.

We mentioned scalability a few times, and this is where OpenStack shines in both ways. The amazing thing is that not only does it scale up easily but that it also scales down. An installation that will work perfectly fine for a single user can be done on a single machine – even on a single VM inside a single machine – so you will be able to have your own cloud within a virtual environment on your laptop. This is great for testing things but nothing else.

Having a bare-metal install that will follow the guidelines and recommended configuration requirements for particular roles and services is the only way to go forward when creating a working, scalable cloud, and obviously this is the way to go if you need to create a production environment. Having said that, between a single machine and a thousand server installs, there are a lot of ways that your infrastructure can be shaped and redesigned to support your particular use case scenario.

Let's first quickly go through an installation inside another VM, a task that can be accomplished in under 10 minutes on a faster host machine. For our platform, we decided on installing Ubuntu 18.04.3 LTS in order to be able to keep the host system to a minimum. The entire guide for Ubuntu regarding what we are trying to do is available at

<https://docs.openstack.org/devstack/latest/guides/single-machine.html>.

One thing that we must point out is that the OpenStack site has a guide for a number of different install scenarios, both on virtual and bare-metal hardware, and they are all extremely easy to follow, simply because the documentation is straight to the point. There's also a simple install script that takes care of everything once a few steps are done manually by you.

Be careful with hardware requirements. There are some good sources available to cover this subject. Start here:

<https://docs.openstack.org/newton/install-guide-rdo/overview.html#figure-hwreqs>.

## Installing OpenStack step by step

The first thing we need to do is create a user that is going to install the entire system. This user needs to have **sudo** privileges since a lot of things require system-wide permissions.

Create a user either as root or through **sudo**:

```
useradd -s /bin/bash -d /opt/stack -m  
stack  
chmod 755 /opt/stack
```

The next thing we need to do is allow this user to use **sudo**:

```
echo "stack ALL=(ALL) NOPASSWD: ALL"  
>> /etc/sudoers
```

We also need to install **git** and switch to our newly created user:

```

stack@openstack:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man libasn1-8-heimdal libcurl3-gnutls liberror-perl libgdbm-compat4 libgssapi3-heimdal
  libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libkrb5-26-heimdal libldap-2.4-2 libldap-common libnghttp2-14 libperl5.26 libroken18-heimdal
  librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db libwind0-heimdal patch perl
  perl-modules-5.26
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs
  git-mediawiki git-svn libsasl2-modules-gssapi-mit | libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql diffutils-doc perl-doc
  libterm-readline-gnu-perl | libterm-readline-perl-perl make
The following NEW packages will be installed:
  git git-man libasn1-8-heimdal libcurl3-gnutls liberror-perl libgdbm-compat4 libgssapi3-heimdal
  libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libkrb5-26-heimdal libldap-2.4-2 libldap-common libnghttp2-14 libperl5.26 libroken18-heimdal
  librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db libwind0-heimdal patch perl
  perl-modules-5.26
0 upgraded, 25 newly installed, 0 to remove and 0 not upgraded.
Need to get 12.8 MB of archives.
After this operation, 80.6 MB of additional disk space will be used.
Do you want to continue? [Y/n] y

```

Figure 12.10 – Installing git, the first step in deploying OpenStack

Now for the fun part. We are going to clone (copy the latest version of) **devstack**, the installation script that will provide everything we need to be able to run and use OpenStack on this machine:

```

stack@openstack:~$ git clone https://opendev.org/openstack/devstack
Cloning into 'devstack'...
remote: Enumerating objects: 44764, done.
remote: Counting objects: 100% (44764/44764), done.
remote: Compressing objects: 100% (20257/20257), done.
remote: Total 44764 (delta 31643), reused 36536 (delta 23819)
Receiving objects: 100% (44764/44764), 9.10 MiB | 900.00 KiB/s, done.
Resolving deltas: 100% (31643/31643), done.
stack@openstack:~$ cd devstack/
stack@openstack:~/devstack$ _

```

Figure 12.11 – Cloning devstack by using git

A little bit of configuration is now needed. Inside the **samples** directory, in the directory we just cloned, there is a file called **local.conf**. Use it to configure all the things the installer needs. Networking is one thing that has to be configured manually – not just the local network, which is the one that connects you to the rest of the internet, but also the internal network address space, which is going to get used for everything OpenStack needs to do between instances. Different passwords for different services also need to be set. All of this can be read in the sample file. Directions on how to exactly configure this are both on the web at the address we gave you earlier, and inside the file itself:

```
# Sample ``local.conf`` for user-configurable variables in ``stack.sh``
# NOTE: Copy this file to the root DevStack directory for it to work properly.
# ``local.conf`` is a user-maintained settings file that is sourced from ``stackrc``
# This gives it the ability to override any variables set in ``stackrc``.
# Also, most of the settings in ``stack.sh`` are written to only be set if no
# value has already been set; this lets ``local.conf`` effectively override the
# default values.
# This is a collection of some of the settings we have found to be useful
# in our DevStack development environments. Additional settings are described
# in https://docs.openstack.org/devstack/latest/configuration.html#local-conf
# These should be considered as samples and are unsupported DevStack code.
# The ``localrc`` section replaces the old ``localrc`` configuration file.
# Note that if ``localrc`` is present it will be used in favor of this section.
[[local|localrc]]
# Minimal Contents
# -----
# While ``stack.sh`` is happy to run without ``localrc``, devlife is better when
# there are a few minimal variables set:
# If the ``*_PASSWORD`` variables are not set here you will be prompted to enter
# values for them by ``stack.sh`` and they will be added to ``local.conf``.
ADMIN_PASSWORD=nomoresecret
DATABASE_PASSWORD=stackdb
RABBIT_PASSWORD=stackqueue
SERVICE_PASSWORD=admin_password
# ``HOST_IP`` and ``HOST_IPV6`` should be set manually for best results if
# the NIC configuration of the host is unusual, i.e. ``eth1`` has the default
# route but ``eth0`` is the public interface. They are auto-detected in
# ``stack.sh`` but often is indeterminate on later runs due to the IP moving
;
```

Figure 12.12 – Installer configuration

There will be some issues with this installation process, and as a result, installation might break twice because of the following reasons:

- Ownership of **/opt/stack/.cache** is **root:root**, instead of **stack:stack**. Please correct this ownership before running the installer;
- An installer problem (a known one), as it fails to install a component and then fails. Solution is rather simple - there's a line that needs to be changed in a file in inc directory, called python. At time of writing, line 192 of that file needs to be changed from **\$cmd\_pip \$upgrade** \ to **\$cmd\_pip \$upgrade --ignore-installed** \

In the end, after we collected all the data and modified the file, we settled on this configuration:

```
[[local|localrc]]
FLOATING_RANGE=192.168.61.222/24
FIXED_RANGE=10.11.10.0/24
ADMIN_PASSWORD=secretpass
DATABASE_PASSWORD=dbpass
RABBIT_PASSWORD=rabbitpass
SERVICE_PASSWORD=admin_password
LOGFILE=$DEST/logs/stack.sh.log
LOGDAYS=2
SWIFT_REPLICAS=1
SWIFT_DATA_DIR=$DEST/data
~
```

### Figure 12.13 – Example configuration

Most of these parameters are understandable, but let's cover two of them first: **FLOATING\_RANGE** and **FIXED\_RANGE**. The **FLOATING\_RANGE** parameter tells our OpenStack installation which network scope will be used for *private* networks. On the other hand, **FIXED\_RANGE** is the network scope that will be used by OpenStack-provisioned virtual machines. Basically, virtual machines provisioned in OpenStack environments will be given internal addresses from **FIXED\_RANGE**. If a virtual machine needs to be available from the outside world as well, we will assign a network address from **FLOATING\_RANGE**. Be careful with **FIXED\_RANGE** as it shouldn't match an existing network range in your environment.

One thing we changed from what is given in the guide is that we reduced the number of replicas in the Swift installation to one. This gives us no redundancy, but reduces the space used for storage and speeds things up a little. Do not do this in the production environment.

Depending on your configuration, you may also need to set the **HOST\_IP** address variable in the file. Here, set it to your current IP address.

Then, run **./stack.sh**.

Once you've run the script, a really verbose installation should start and dump a lot of lines on your screen. Wait for it to finish – it is going to take a while and download a lot of files from the



internet. At the end, it is going to give you an installation summary that looks something like this:

```
This is your host IP address: 192.168.61.129
This is your host IPv6 address: ::1
Horizon is now available at http://192.168.61.129/dashboard
Keystone is serving at http://192.168.61.129/identity/
The default users are: admin and demo
The password: secretpass

WARNING:
Using lib/neutron-legacy is deprecated, and it will be removed in the future

Services are running under systemd unit files.
For more information see:
https://docs.openstack.org/devstack/latest/systemd.html

DevStack Version: ussuri
Change: 455ba66098953b08dabf38ec7256998de89ac755 Merge "Remove conflicting packages in Ubuntu" 2020-01-30 00:01:06 +0000
OS Version: Ubuntu 18.04 bionic
stack@openstack:~/devstack$ _
```

Figure 12.14 – Installation summary

Once this is done, if everything is okay, you should have a complete running version of OpenStack on your local machine. In order to verify that, connect to your machine using a web browser; a welcome screen should appear:



openstack.

Log in

---

User Name

Password

Sign in

Figure 12.15 – OpenStack login screen

After logging in with the credentials that are written on your machine, after the installation (the default administrator name is **admin** and the password is the one you set in **local.conf** when installing the service), you are going to be welcomed by a screen showing you the stats for your cloud. The screen you are looking at is actually a Horizon dashboard and is the main screen

that provides you with all you need to know about your cloud at a glance.

## OpenStack administration

Looking at the top-left corner of Horizon, we can see that there are three distinct sections that are configured by default. The first one – **Project** – covers everything about our default instance and its performance. This is where you can create new instances, manage images, and work on server groups. Our cloud is just a core installation, so we only have one server and two defined zones, which means that we have no server groups installed:

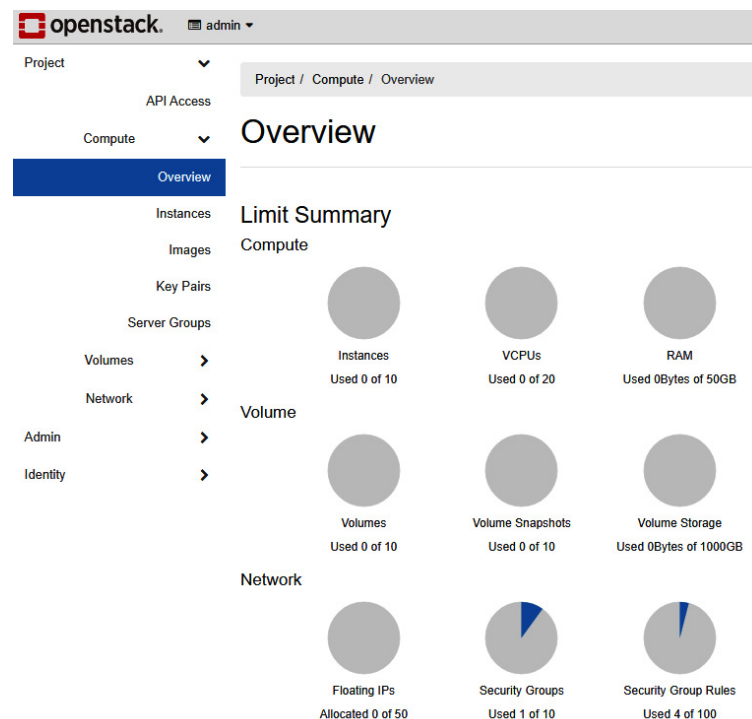


Figure 12.16 – Basic Horizon dashboard

First, let's create a quick instance to show how this is done:

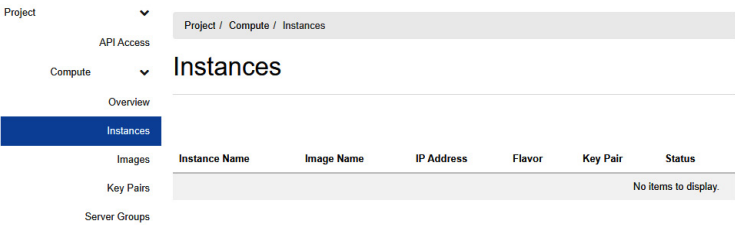


Figure 12.17 – Creating an instance

Follow these steps to create an instance:

1. Go to **Launch Instance** in the far-right part of the screen. A window will open that will enable you to give OpenStack all the information it needs to create a new VM instance:

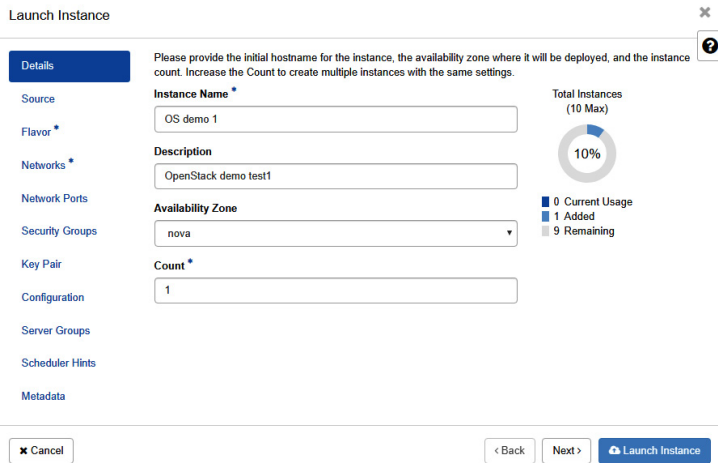


Figure 12.18 – Launch Instance wizard

2. On the next screen, you need to supply the system with the image source. We already mentioned glances – these images are taken from the Glance store and can be either an image snapshot, a ready-made volume, or a volume snapshot. We can also create a persistent image if we want to. One thing that you'll notice is that there are two differences when comparing this process to almost any other deployment. The first is that we are using a ready-made image by default as one was provided for us. Another big thing is the ability to create a new persistent volume to store our data in,

or to have it deleted when we are done with the image, or have it not be created at all. Choose the one image you have allocated in the public repository; it should be called something similar to the one shown in the following screenshot. CirrOS is a test image provided with OpenStack. It's a minimal Linux distribution that is designed to be as small as possible and enable easy testing of the whole cloud infrastructure but to be as unobtrusive as possible. CirrOS is basically an OS placeholder. Of course, we need to click on the **Launch Instance** button to go to the next step:

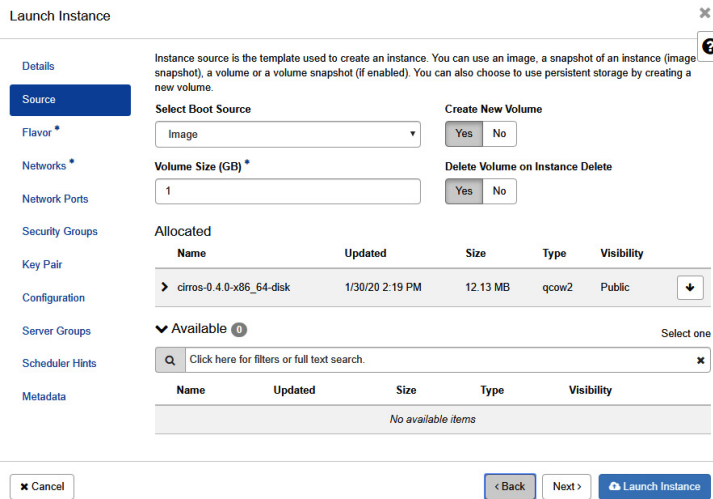


Figure 12.19 – Selecting an instance source

3. The next important part of creating a new image is choosing a flavor. This is another one of those peculiarly named things in OpenStack. A flavor is a combination of certain resources that basically creates a computing, memory, and storage template for new instances. We can choose from instances that have as little as 64 MB of RAM and 1 vCPU and go as far as our infrastructure can provide:

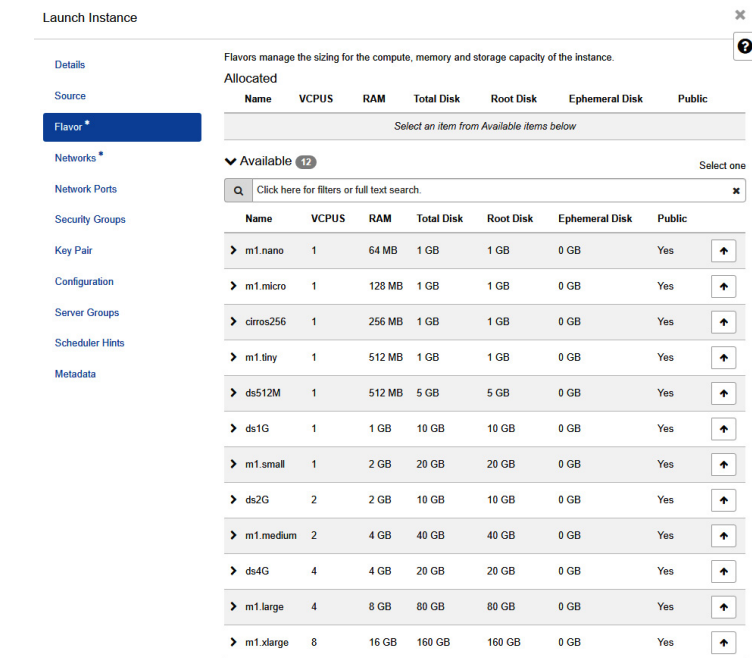


Figure 12.20 – Selecting an instance flavor

In this particular example, we are going to choose **cirros256**, a flavor that is basically designed to provide our test system with as few resources as is feasible.

4. The last thing we actually need to choose is the network connectivity. We need to set all the adapters our instance will be able to use while running. Since this is a simple test, we are going to use both adapters we have, both the internal and external one. They are called **public** and **shared**:

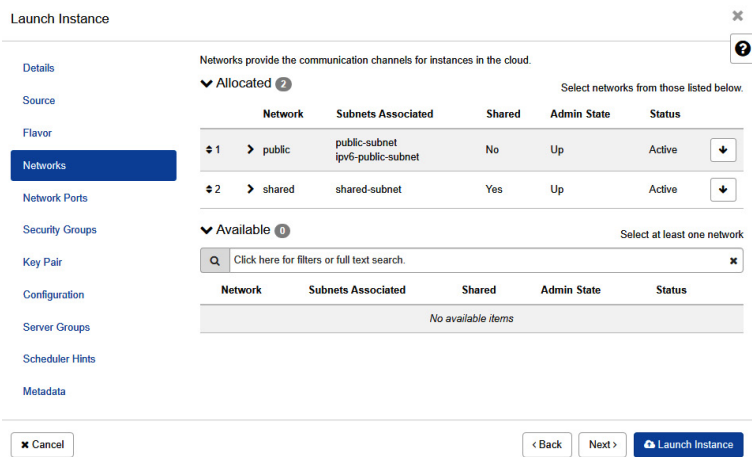


Figure 12.21 – Instance network configuration

Now, we can launch our instance and it will be able to boot. Once you click on the **Launch Instance** button, it is going to take probably under a minute to create a new instance. The screen showing its current progress and instance status will auto update while the instance is being deployed.

Once this is done, our instance will be ready:

<input type="checkbox"/>	OS demo 1	cirros-0.4.0-x86_64-disk	shared 192.168.233.155 public 192.168.61.24, 2001:db8::1b9
--------------------------	-----------	--------------------------	---

Figure 12.22 – The instance is ready

We'll quickly create another instance, and then create a snapshot so that we can show you how image management works. If you click on the **Create snapshot** button on the right-hand side of the instance list, Horizon will create a snapshot and immediately put you in the interface meant for image administration:

Project

API Access

Compute

Overview

Instances

Images

Key Pairs

Server Groups

Volumes

Network

Admin

Identity

Project / Compute / Images

Images

Click here for filters or full text search.

Displaying 2 items

<input type="checkbox"/>	Owner	Name
<input type="checkbox"/>	admin	cirros-0.4.0-x86_64-disk
<input type="checkbox"/>	admin	Test snap 1

Displaying 2 items

Figure 12.23 – Images

Now, we have two different snapshots: one that is the start image and another that is an actual snapshot of the image that is running. So far, ev-

everything has been simple. What about creating an instance out of a snapshot? It's just a click away! What you need to do is just click on the **Launch Instance** button on the right and go through the wizard for creating new instances.

The end result of our short example of instance creation should be something like this:

<input type="checkbox"/>	OS test 3	Test snap 1	public 192.168.61.221, 2001:db8::263 shared 192.168.233.191	cirros256
<input type="checkbox"/>	OS demo 2	cirros-0.4.0-x86_64-disk	shared 192.168.233.121 public 192.168.61.98, 2001:db8::1b6	cirros256
<input type="checkbox"/>	OS demo 1	cirros-0.4.0-x86_64-disk	shared 192.168.233.155 public 192.168.61.24, 2001:db8::1b9	cirros256

Figure 12.24 – New instance creation finished

What we can see is all the information we need on our instances, what their IP addresses are, their flavor (which translates into what amount of resources are allocated for a particular instance), the availability zone that the image is running in, and information on the current instance state. The next thing we are going to check out is the **Volumes** tab on the left. When we created our instances, we told OpenStack to create one permanent volume for the first instance. If we now click on **Volumes**, we should see the volume under a numeric name:

Project

API Access

Compute

Volumes

Volumes

Snapshots

Groups

Group Snapshots

Network

Admin

Identity

Project / Volumes / Volumes

Volumes

Displaying 1 item

<input type="checkbox"/>	Name	Description	Size	Status
<input type="checkbox"/>	73238817-a806-4af7-82fe-511f5108c6d4	-	1GiB	In-use

Displaying 1 item

Figure 12.25 – Volumes

From this screen, we can now snapshot the volume, reattach it to a different instance, and even upload it as an image to the repository.

The third tab on the left-hand side, named **Network**, contains even more information about our currently configured setup.

If we click on the **Network Topology** tab, we will get the whole network topology of our currently running network, shown in a simple graphical display. We can choose from **Topology** and **Graph**, both of which basically represent the same thing:

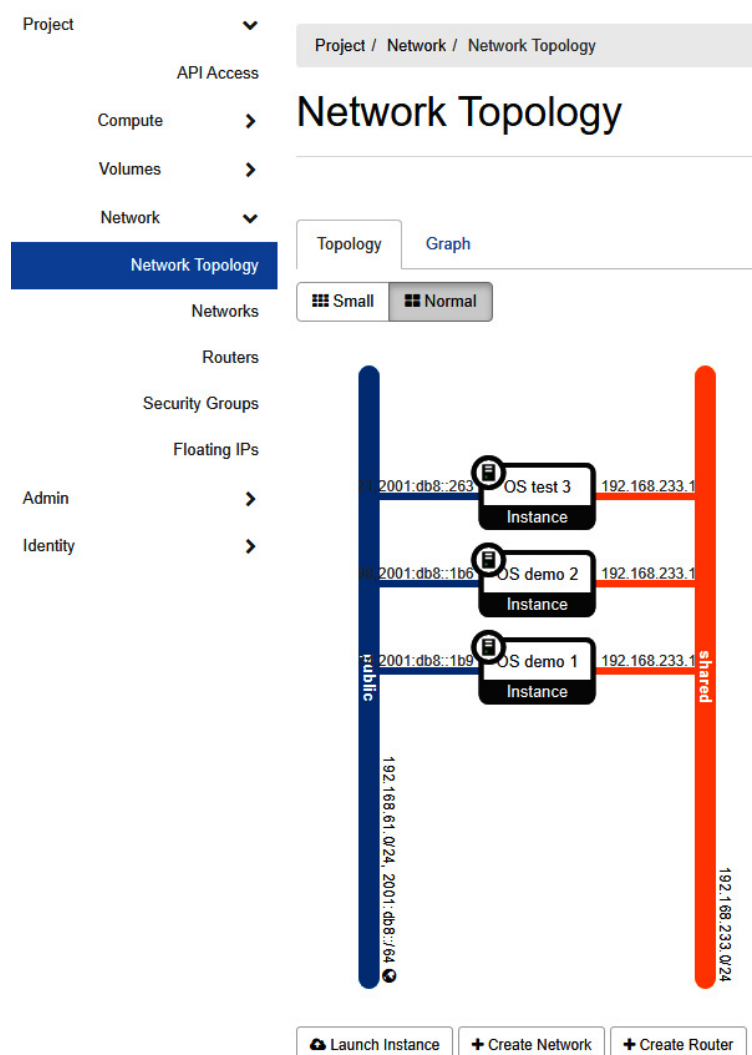


Figure 12.26 – Network topology



If we need to create another network or change anything in the network matrix, we can do so here. We consider this to be really administrator-friendly, on top of being documentation-friendly. Both of these points make our next topic – day-to-day administration – much easier.

## Day-to-day administration

We are more or less finished with the most important options that are in any way connected to the administration of our day-to-day tasks in the **Project** Datacenter. If we click on the tab named **Admin**, we will notice that the menu structure we've opened looks a lot like the one under **Project**. This is because, now, we are looking at administration tasks that have something to do with the infrastructure of the cloud, not the infrastructure of our particular logical Datacenter, but the same building blocks exist in both of these. However, if we – for example – open **Compute**, a completely different set of options exist:

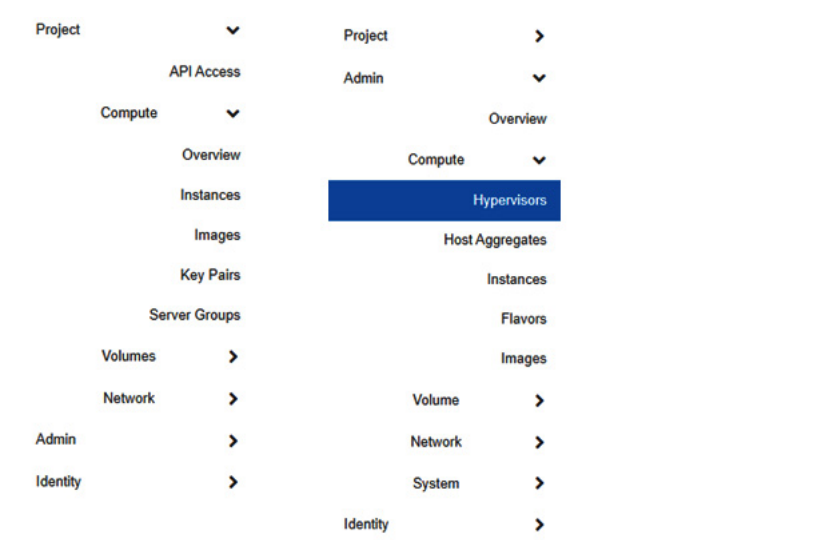


Figure 12.27 – Different available configuration options

This part of the interface is used to completely administer parts that form our infrastructure and those that define different things we can use while working in our *Datacenter*. When logged in as a user, we can add and remove virtual machines, configure networks, and use resources, but to put resources online, add new hypervisors, define flavors, and do these kinds of tasks that completely change the infrastructure, we need to be assigned the administrative role. Some of the functions overlap, such as both the administrative part of the interface and the user-specific part, which have control over instances. However, the administrative part has all these functions, and users can have their set of commands tweaked so that they are, for instance, unable to delete or create new instances.

The administritative view enables us to monitor our nodes on a more direct level, not only through the services they provide, but also through raw data about a particular host and the resources utilized on it:

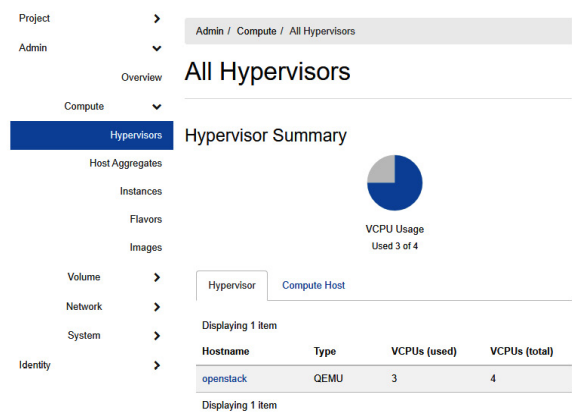


Figure 12.28 – Available hypervisors in our Datacenter

Our Datacenter has only one hypervisor, but we can see the amount of resources physically available on it, and the share of those resources the current setup is using at this particular moment.

Flavors are also one important part of the whole of OpenStack. We already mentioned them as a predefined sets of resource presets that form a platform that the instance is going to run on. Our test setup has a few of them defined, but we can delete the ones that are shipped in this setup and create new ones tailored to our needs.

Since the point of the cloud is to optimize resource management, flavors play a big part in this concept. Creating flavors is not an easy task in terms of planning and design. First and foremost, it requires deep knowledge of what is possible on a given hardware platform, how much and what computing resources even exist, and how to utilize it to the full extent possible. So, it is essential that we plan and design things properly. The other thing is that we actually need to understand what kind of load we are preparing for. Is it memory-intensive? Do we have many small services that require a lot of nodes with a simple configuration? Are we going to need a lot of computing power and/or a lot of storage? The answers to those questions are something that will not only enable us to create what our clients want, but also create flavors that will have users utilizing our infrastructure in full.


The basic idea is to create flavors that will give individual users just enough resources to get their job done in a satisfactory way. This is not

obvious in a deployment that has 10 instances, but once we run into thousands, a flavor that always leaves 10 percent of the storage unused is quickly going to eat into our resources and limit our ability to serve more users. Striking this balance between what we have and what we give users to use in a particular way is probably the hardest task in planning and designing our environments:

Create Flavor

Flavor Information \* Flavor Access

Name \*

ID 

VCPUs \*

RAM (MB) \*

Root Disk (GB) \*

Ephemeral Disk (GB)

Swap Disk (MB)

RX/TX Factor

Flavors define the sizes for RAM, disk, number of cores, and other resources and can be selected when users deploy instances.

Cancel Create Flavor

Figure 12.29 – Create Flavor wizard

Creating a flavor is a simple task. We need to do the following:

1. Give it a name; an ID will be assigned automatically.
2. Set the number of vCPUs and RAM for our flavor.
3. Select the size of a base disk, and an ephemeral disk that doesn't get included in any

- of the snapshots and gets deleted when a virtual machine is terminated.
4. Select the amount of swap space.
  5. Select the RX/TX factor so that we can create a bit of QoS on the network level. Some flavors will need to have more network traffic priority than others.

OpenStack allows a particular project to have more than one flavor, and for a single flavor to belong to different projects. Now that we've learned that, let's work with our user identities and assign them some objects.

## Identity management

The last tab on the left-hand side is **Identity**, which is responsible for handling users, roles, and projects. This is where we are going to configure not only our usernames, but the user roles, groups, and projects a particular user can use:

Project

Admin

Identity

Identity / Users

Users

Groups	User Name	Description	Email	User ID	Enabled	Domain Name	Actions
<input type="checkbox"/>	admin	-		1db138b939a449c7d704d8e15e7b4	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	demo	-	demo@example.com	6e85dc8b8a1e41d8b37a8b2d91084c	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	all_demo	-	all_demo@example.com	d92d4d7a7cc4db384c4d81c0e9127f	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	nova	-		4863227bea0448d033899440310fdb	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	glance	-		132e3be0f0b442c19d14c0f01a0d612	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	cinder	-		d935c96aeb49c20993020c4ac027	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	neutron	-		54bc33f4b442c2975d4955e6c8b05	Yes	Default	<a href="#">Edit</a>
<input type="checkbox"/>	placement	-		4db78d672b4e11570d46d74c31080	Yes	Default	<a href="#">Edit</a>

Figure 12.30 – Users, Groups, and Roles management

We are not going to go too much into how users are managed and installed, but just cover the basics of user management. As always, the original documentation on the OpenStack site is the place to go to learn more. Make sure that you check

out this link:

<https://docs.openstack.org/keystone/pike/admin/cli-manage-projects-users-and-roles.html>.

In short, once you create a project, you need to define which users are going to be able to see and work on a particular project. In order to ease administration, users can also be part of groups, and you can then assign whole groups to a project.

The point of this structure is to enable the administrator to limit the users not only to what they can administer, but also to how many of the available resources are available for a particular project. Let's use an example for this. If we go to **Projects** and edit an existing project (or create a new one), we'll see a tab called **Quota** in the configuration menu, which looks like this:

## Edit Quotas



**Compute \*****Volume \*****Network \***

Instances \*

10

VCPUs \*

20

RAM (MB) \*

51200

Metadata Items \*

128

Key Pairs \*

100

Server Groups \*

10

Server Group Members \*

10

Injected Files \*

5

Injected File Content (Bytes) \*

10240

Length of Injected File Path \*

255

Cancel

Save

Figure 12.31 – Quotas on the default project

Once you create a project, you can assign all the resources in the form of quotas. This assignment limits the maximum available resources for a particular group of instances. The user has no overview of the whole system; they can only *see* and utilize resources available through the project. If a user is part of multiple projects, they can create, delete, and manage instances based on their role in the project, and the resources available to them are specific to a project.

We'll discuss OpenStack/Ansible integration next, as well as some potential use cases for these two concepts to work together. Keep in mind that the larger the OpenStack environment is, the more use cases we will find for them.

# Integrating OpenStack with Ansible

Dealing with any large-scale application is not easy, and not having the right tool can make it impossible. OpenStack provides a lot of ways for us to directly orchestrate and manage a huge horizontal deployment, but sometimes, this is not enough. Luckily, in our arsenal of tools, we have another one – **Ansible**. In *[Chapter 11](#)*, *Ansible for Orchestration and Automation*, we covered some other, smaller ways to use Ansible to deploy and configure individual machines, so we are not going to go back to that. Instead, we are going to focus on things that Ansible is good for in the OpenStack environment.

One thing that we must make clear, though, is that using Ansible in an OpenStack environment can be based on two very distinct scenarios. One is using Ansible to handle deployed instances, in a way that would pretty much look the same across all the other cloud or bare-metal deployments. You, as an administrator of a large number of instances, create a management node that is nothing more than a Python-enabled server with added Ansible packages and playbooks. After that, you sort out the inventory for your deployment and are ready to manage your instances. This scenario is not what this part of this chapter is about.

What we are talking about here is using Ansible to manage the cloud itself. This means we are



not deploying instances inside the OpenStack cloud; we are deploying compute and storage nodes for OpenStack itself.

The environment we are talking about is sometimes referred to as **OpenStack-Ansible (OSA)** and is common enough to have its own deployment guide, located at the following URL:

<https://docs.openstack.org/project-deploy-guide/openstack-ansible/latest/>.

The requirements for a minimal installation in OpenStack-Ansible are considerably greater than those in a single VM or on a single machine. The reason for this is not just that the system needs all the resources; it's the tools that need to be used and the philosophy behind it all.

Let's quickly go through what Ansible means in terms of OpenStack:

- Once configured, it enables the quick deployment of any kind of resource, be it storage or computing.
- It makes sure you are not forgetting to configure something in the process. When deploying a single server, you will have to make sure that everything works and that errors in configuration are easy to spot, but when deploying multiple nodes, errors can creep in and degrade the performance of part of the system without anyone noticing. The normal deployment practice to avoid this is an installation checklist, but Ansible is a much better solution than that.
- More streamlined configuration changes.

Sometimes, we need to apply a **configuration**

change across the whole system or some part of it. This can be frustrating if not scripted.

So, having said all that, let's quickly go through <https://docs.openstack.org/openstack-ansible/latest/> and see what the official documentation says about how to deploy and use Ansible and OpenStack.

What exactly does OpenStack offer to the administrator in regard to Ansible? The simplest answer is playbooks and roles.

To use Ansible to deploy OpenStack, you basically need to create a deployment host and then use Ansible to deploy the whole OpenStack system. The whole workflow goes something like this:

1. Prepare the deployment host
2. Prepare the target hosts
3. Configure Ansible for deployment
4. Run playbooks and let them install everything
5. Check whether OpenStack is correctly installed

When we are talking about deployment and target hosts, we need to make a clear distinction: the deployment host is a single entity holding Ansible, scripts, playbooks, roles, and all the supporting bits. The target hosts are the actual servers that are going to be part of the OpenStack cloud.

The requirements for installation are straightforward:

- The operating system should be a minimal installation of Debian, Ubuntu CentOS, or openSUSE (experimental) with the latest kernel and full updates applied.
- Systems should also run Python 2.7, have SSH access enabled with public key authentication, and have NTP time sync enabled. This covers the deployment host.
- There are also usual recommendations for different types of nodes. Computing nodes must support hardware-assisted virtualization, but that's an obvious requirement.
- There is a requirement that should go without saying, and that is to use multicore processors, with as many cores as possible, to enable some services to run much faster.

Disk requirements are really up to you.

OpenStack suggests using fast disks if possible, recommending SSD drives in a RAID, and large pools of disks available for block storage.

- Infrastructure nodes have requirements that are different than the other types of nodes since they are running a few databases that grow over time and need at least 100 GB of space. The infrastructure also runs its services as containers, so it will consume resources in a particular way that will be different than the other compute nodes.

The deployment guide also suggests running a logging host since all the services create logs. The recommended disk space is at least 50 GB for logs, but in production, this will quickly grow in orders of magnitude.

OpenStack needs a fast, stable network to work with. Since everything in OpenStack will depend on the network, every possible solution that will speed up network access is recommended, including using 10G and bonded interfaces. Installing a deployment server is the first step in the overall process, which is why we'll do that next.

## Installing an Ansible deployment server

Our deployment server needs to be up to date with all the upgrades and have Python, **git**, **ntp**, **sudo**, and **ssh** support installed. After you've installed the required packages, you need to configure the **ssh** keys to be able to log into the target hosts. This is an Ansible requirement and is also a best practice that leverages security and ease of access.

The network is simple – our deployment host must have connectivity to all the other hosts. The deployment host should also be installed on the L2 of the network, which is designed for container management.

Then, the repository should be cloned:

```
# git clone -b 20.0.0  
https://opendev.org/openstack/openstack-ansible /opt/openstack-ansible
```

Next, an Ansible bootstrap script needs to be run:

```
# scripts/bootstrap-ansible.sh
```

This concludes preparing the Ansible deployment server. Now, we need to prepare the target computers we are going to use for OpenStack. Target computers are currently supported on Ubuntu Server (18.04) LTS, CentOS 7, and openSUSE 42.x (at the time of writing, there still isn't CentOS 8 support). You can use any of these systems. For each of them, there is a helpful guide that will get you up and running quickly:

<https://docs.openstack.org/project-deploy-guide/openstack-ansible/latest/deploymenthost.html>. We'll just explain the general steps to ease you into installing it, but in all truth, just copy and paste the commands that have been published for your operating system from <https://www.openstack.org/>.

No matter which system you decide to run on, you have to be completely up to date with system updates. After that, install the **linux-image-extra** package (if it exists for your kernel) and install the **bridge-utils**, **debootstrap**, **ifenslave**, **lsuf**, **lvm2**, **chrony**, **openssh-server**, **sudo**, **tcpdump**, **vlan**, and Python packages. Also, enable bonding and VLAN interfacing. All these things may or may not be available for your system, so if something is already installed or configured, just skip over it.

Configure the NTP time sync in **chrony.conf** to synchronize time across the whole deployment. You can use any time source you like, but for the system to work, time must be in sync.

Now, configure the **ssh** keys. Ansible is going to deploy using **ssh** and key-based authentication. Just copy the public keys from the appropriate user on your deployment machine to **/root/.ssh/authorized\_keys**. Test this setup by simply logging in from the deployment host to the target machine. If everything is okay, you should be able to log in without any password or any other prompt. Also, note that the root user on the deployment host is the default user for managing everything and that they have to have their **ssh** keys generated in advance since they are used not only on the target hosts but also in all the containers for different services running across the system. These keys must exist when you start to configure the system.

For storage nodes, please note that LVM volumes will be created on the local disks, thus overwriting any existing configuration. Network configuration is going to be done automatically; you just need to ensure that Ansible is able to connect to the target machines.

The next step is configuring our Ansible inventory so that we can use it. Let's do that now.

## Configuring the Ansible inventory

Before we can run the Ansible playbooks, we need to finish configuring the Ansible inventory so that it points the system to the hosts it should install on. We are going to quote the verbatim, available at <https://docs.openstack.org/project-deploy-guide/openstack-ansible/queens/configure.html>:

- 1. Copy the contents of the `/opt/openstack-ansible/etc/openstack_deploy` directory to the `/etc/openstack_deploy` directory.*
- 2. Change to the `/etc/openstack_deploy` directory.*
- 3. Copy the `openstack_user_config.yml.example` file to `/etc/openstack_deploy/openstack_user_config.yml`.*
- 4. Review the `openstack_user_config.yml` file and make changes to the deployment of your OpenStack environment.*

Once inside the configuration file, review all the options. **Openstack\_user\_config.yml** defines which hosts run which services and nodes. Before committing to installing, please review the documentation mentioned in the previous paragraph.

One thing that stands out on the web is **install\_method**. You can choose either source or distro. Each has its pros and cons:

- Source is the simplest installation as it's done directly from the sources on the OpenStack official site and contains an environment that's compatible with all systems.
- The distro method is customized for the particular distribution you are installing on by using specific packages known to work and known as being stable. The major drawback of this is that updates are going to be much slower since not only OpenStack needs to be deployed but

also information about all the packages on distributions, and that setup needs to be verified. As a result, expect long waits between when the upgrade reaches the *source* and gets to your *distro* installation. After installing, you must go with your primary choice; there is no mechanism for switching from one to the other.

The last thing you need to do is open the **user\_secrets.yml** file and assign passwords for all the services. You can either create your own passwords or use a script provided just for this purpose.

## Running Ansible playbooks

As we go through the deployment process, we will need to start a couple of Ansible playbooks. We need to use these three provided playbooks in this order:

- **setup-hosts.yml** : The initial Ansible playbook that we use to provision the necessary services on our OpenStack hosts.
- **setup-infrastructure.yml**: The Ansible playbook that deploys some more services, such as RabbitMQ, repository server, Memcached, and so on.
- **setup-openstack.yml**: The Ansible playbook that deploys the remaining services – Glance, Cinder, Nova, Keystone, Heat, Neutron, Horizon, and so on.

All of these Ansible playbooks need to be finished successfully so that we can integrate



Ansible with Openstack. So, the only thing left is to run the Ansible playbooks. We need to start with the following command:

```
# openstack-ansible setup-hosts.yml
```

You can find the appropriate files in **/opt/openstack-ansible/playbooks**. Now, run the remaining setups:

```
# openstack-ansible setup-  
infrastructure.yml  
# openstack-ansible setup-  
openstack.yml
```

All the playbooks should finish without unreachable or failed plays. And with that – congratulations! You have just installed OpenStack.

## Summary

In this chapter, we spent a lot of time describing the architecture and inner workings of OpenStack. We discussed software-defined networking and its challenges, as well as different OpenStack services such as Nova, Swift, Glance, and so on. Then, we moved on to practical issues, such as deploying Packstack (let's just call that OpenStack for proof of concept), and full OpenStack. In the last part of this chapter, we discussed OpenStack-Ansible integration and what it might mean for us in larger environments.

Now that we've covered the *private* cloud aspect, it's time to grow our environment and expand it to a more *public* or *hybrid*-based approach. In KVM-based infrastructures, this usually means

connecting to AWS to convert your workloads and transfer them there (public cloud). If we're discussing the hybrid type of cloud functionality, then we have to introduce an application called Eucalyptus. For the hows and whys, check out the next chapter.

## Questions

1. What is the main problem with VLAN as a cloud overlay technology?
2. Which types of cloud overlay networks are being used on the cloud market today?
3. How does VXLAN work?
4. What are some of the most common problems with stretching Layer 2 networks across multiple sites?
5. What is OpenStack?
6. What are the architectural components of OpenStack?
7. What is OpenStack Nova?
8. What is OpenStack Swift?
9. What is OpenStack Glance?
10. What is OpenStack Horizon?
11. What are OpenStack flavors?
12. What is OpenStack Neutron?

## Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- OpenStack documentation:  
<https://docs.openstack.org>

- Arista VXLAN overview:  
[https://www.arista.com/assets/data/pdf/Whitepapers/Arista Networks](https://www.arista.com/assets/data/pdf/Whitepapers/Arista_Networks)
- Red Hat – What is GENEVE?:  
<https://www.redhat.com/en/blog/what-geneve>
- Cisco – Configuring Virtual Networks Using OpenStack:  
<https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus1000/>
- Packstack: <http://rdoproject.org>