MASTERING OPENSTACK Free Download

Mastering OpenStack - Second Edition. Mastering OpenStack -. Mastering OpenStack.

- Omar Khedher
- 400 pages
- 31 Jul 2015
- Packt Publishing Limited
- 9781784395643
- English
- Birmingham, United Kingdom

Mastering OpenStack [Book]

Finally, you will gain experience of running a centralized logging server and monitoring OpenStack services. The book will show you how to carry out performance tuning based on OpenStack service logs. You will be able to master OpenStack benchmarking and performance tuning. By the end of the book, you'll be ready to take steps to deploy and manage an OpenStack cloud with the latest open source technologies. This book will help you understand the flexibility of OpenStack by showcasing integration of several out-of-the-box solutions in order to build a large-scale cloud environment.. It will also cover detailed discussions on the various design and deployment strategies for implementing a fault-tolerant and highly available cloud infrastructure. Leverage the power of Ansible to gain complete control over your systems and automate application deployment …. Dive in to the cutting edge techniques of Linux KVM virtualization, and build the virtualization solutions ….

Finally, you will gain experience of running a centralized logging server and monitoring OpenStack services. The book will show you how to carry out performance tuning based on OpenStack service logs. You will be able to master OpenStack benchmarking and performance tuning. By the end of the book, you'll be ready to take steps to deploy and manage an OpenStack cloud with the latest open source technologies. **All of the code is organized into folders, code files are available for Chapter 1, 2, 3, 4 and project file.**

For example, Chapter 1. Click here if you have any feedback or suggestions. Skip to content. Branches Tags. Could not load branches. Could not load tags. It ensures that the network will not be turned into a bottleneck or limiting factor in a cloud deployment and gives users real self-service, even over their network configurations. Another advantage of Neutron is its ability to provide a way to integrate vendor networking solutions and a flexible way to extend network services. It is designed to provide a plugin and extension mechanism that presents an option for network operators to enable different technologies via the Neutron API. Keep in mind that Neutron allows users to manage and create networks or connect servers and nodes to various networks.

The scalability advantage will be discussed in a later topic in the context of the Software Defined Network SDN and Network Function Virtualization NFV technology, which is attractive to many networks and administrators who seek a high-level network multi-tenancy. Ceilometer provides a metering service in OpenStack. In a shared, multi-tenant environment such as OpenStack, metering resource utilization is of prime importance.

Ceilometer collects data associated with resources. Resources can be any entity in the OpenStack cloud such as VMs, disks, networks, routers, and so on. Resources are associated with meters. The utilization data is stored in the form of samples in units defined by the associated meter. Ceilometer has an inbuilt summarization capability. Ceilometer allows data collection from various sources, such as the message bus, polling resources, centralized agents, and so on. As an additional design change in the Telemetry service in OpenStack since the Liberty release, the Alarming service has been decoupled from the Ceilometer project to make use of a new incubated project code-named Aodh.

The Telemetry Alarming service will be dedicated to managing alarms and triggering them based on collected metering and scheduled events. More Telemetry service enhancements have been proposed to adopt a Time Series Database as a Service project code-named Gnoochi. This architectural change will tackle the challenge of metrics and event storage at scale in the OpenStack Telemetry service and improve its performance. Initial development for Heat was limited to a few OpenStack resources including compute, image, block storage, and network services.

Heat has boosted the emergence of resource management in OpenStack by orchestrating different cloud resources resulting in the creation of stacks to run applications with a few pushes of a button. From simple template engine text files referred to as HOT templates Heat Orchestration Template , users are able to provision the desired resources and run applications in no time. Heat is becoming an attractive OpenStack project due to its maturity and extended support resources catalog within the latest OpenStack releases. Other incubated OpenStack projects such as Sahara Big Data as a Service have been implemented to use the Heat engine to orchestrate the creation of the underlying resources stack. It is becoming a mature component in OpenStack and can be integrated with some system configuration management tools such as Chef for full stack automation and configuration setup.

Horizon is the web dashboard that pulls all the different pieces together from the OpenStack ecosystem. Horizon provides a web frontend for OpenStack services. Currently, it includes all the OpenStack services as well as some incubated projects. It was designed as a stateless and data-less web application. It does not keep any data except the session information in its own data store. It is designed to be a reference implementation that can be customized and extended by operators for a particular cloud. It forms the basis of several public clouds, most notably the HP Public Cloud, and at its heart is its extensible modular approach to construction. Horizon is based on a series of modules called panels that define the interaction of each service. Its modules can be enabled or disabled, depending on the service availability of the particular cloud. Message Queue provides a central hub to pass messages between different components of a service.

This is where information is shared between different daemons by facilitating the communication between discrete processes in an asynchronous way. One major advantage of the queuing system is that it can buffer requests and provide unicast and group-based communication services to subscribers. Its database stores most of the build-time and run-time states for the cloud infrastructure, including instance types that are available for use, instances in use, available networks, and projects. It provides a persistent storage for preserving the state of the cloud infrastructure. It is the second essential piece of sharing information in all OpenStack components. Let's try to see how OpenStack works by chaining all the service cores covered in the previous sections in a series of steps:.

The scheduling process in OpenStack Nova can perform different algorithms such as simple, chance, and zone. An advanced way to do this is by deploying weights and filters by ranking servers as its available resources. It is important to understand how different services in OpenStack work together, leading to a running virtual machine.

The process of launching a virtual machine involves the interaction of the main OpenStack services that form the building blocks of an instance including compute, network, storage, and the base image.

As shown in the previous diagram, OpenStack services interact with each other via a message bus to submit and retrieve RPC calls. The information of each step of the provisioning process is verified and passed by different OpenStack services via the message bus. On the other hand, the inter-communication of APIs within OpenStack requires an authentication mechanism to be trusted, which involves Keystone. Starting with the identity service, the following steps summarize briefly the provisioning workflow based on API calls in OpenStack:. It is important to keep in mind that handling tokens in OpenStack on every API call and service request is a time limited operation.

One of the major causes of a failed provisioning operation in OpenStack is the expiration of the token during subsequent API calls. Additionally, the management of tokens has faced a few changes within different OpenStack releases. This includes two different approaches used in OpenStack prior to the Liberty release including:. Starting from the Kilo release, handling tokens in Keystone has progressed by introducing more sophisticated cryptographic authentication token methods, such as Fernet. Fernet is fully supported in the Mitaka release and the community is pushing to adopt it as the default. Deployment of OpenStack depends on the components were covered previously. It confirms your understanding of how to start designing a complete OpenStack environment. Of course, assuming the versatility and flexibility of such a cloud management platform, OpenStack offers several possibilities that can be considered an advantage. However, owing to such flexibility, it's a challenge to come with the right design decision that suits your needs.

At the end of the day, it all comes down to the use cases that your cloud is designed to service. Many enterprises have successfully designed their OpenStack environments by going through three phases of design: designing a conceptual model, designing a logical model, and finally, realizing the physical design. It's obvious that complexity increases from the conceptual to the logical design and from the logical to the physical design. As the first conceptual phase, we will have our high-level reflection on what we will need from certain generic classes from the OpenStack architecture:. Keep in mind that the illustrated diagram will be refined over and over again since we will aim to integrate more services within our first basic design. In other words, we are following an incremental design approach, within which we should exploit the flexibility of the OpenStack architecture. At this level, we can have a vision and direction of the main goal without worrying about the details.

Based on the conceptual reflection design, most probably you will have a good idea about different OpenStack core components, which will lay the formulation of the logical design. We will start by outlining the relationships and dependencies between the service core of OpenStack. In this section we will look at the deployment architecture of OpenStack. We will start by identifying nodes to run an OpenStack service: the cloud controller, network nodes, and the compute node. You may wonder why such a consideration goes through a physical design classification. However,

seeing the cloud controller and compute nodes as simple packages that encapsulate a bunch of OpenStack services will help you refine your design at an early stage. Furthermore, this approach will help plan in advance further high availability and scalability requirements, and will allow you to introduce them later in more detail.

Thus, the physical model design will be elaborated based on the previous theoretical phases by assigning parameters and values to our design. Let's start with our first logical iteration:. Obviously, in a highly available setup, we should achieve a degree of redundancy in each service within OpenStack. You may wonder about the critical OpenStack services claimed in the first part of this chapter: the database and message queue. Why can't they be separately clustered or packaged on their own? This is a pertinent question. Remember that we are still in the second logical phase where we try to dive slowly into the infrastructure without getting into the details.

Besides, we keep on going from a generic and simple design to targeting specific use-cases. What about high availability? The aforementioned technologies will be discussed in more detail in Chapter 9 , Openstack HA and Failover. Compute nodes are relatively simple as they are intended just to run the virtual machine's workload. In order to manage the VMs, the nova-compute service can be assigned for each compute node.

Besides, we should not forget that the compute nodes will not be isolated; a Neutron agent and an optional Ceilometer compute agent may run these nodes. You should now have a deeper understanding of the storage types within Swift, Cinder, and Manila. You will have to ask yourself questions such as: How much data do I need to store? Will my future use cases result in a wide range of applications that run heavy-analysis data? What are my storage requirements for incrementally backing up a virtual machine's snapshots? Do I really need control over the filesystem on the storage or is just a file share enough?

Do I need a shared storage between VMs? To answer this question, you can think about ephemeral storage as the place where the end user will not be able to access the virtual disk associated with its VM when it is terminated. Ephemeral storage should mainly be used in production when the VM state is non-critical, where users or application don't store data on the VM. If you need your data to be persistent, you must plan for a storage service such as Cinder or Manila. Remember that the current design applies for medium to large infrastructures. Ephemeral storage can also be a choice for certain users; for example, when they consider building a test environment.

Considering the same case for Swift, we claimed previously that object storage might be used to store machine images, but when do we use such a solution? Simply put, when you have a sufficient volume of critical data in your cloud environment and start to feel the need for replication and redundancy. OpenStack allows a wide ranging of configurations that variation, and tunneled networks such as GRE, VXLAN, and so on, with Neutron are not intuitively obvious from their appearance to be able to be implemented without fetching their use case in our design. Thus, this important step implies that you may differ between different network topologies because of the reasons behind why every choice was made and why it may work for a given use case. OpenStack has moved from simplistic network features to more complicated ones, but

of course the reason is that it offers more flexibility! This is why OpenStack is here. It brings as much flexibility as it can!

Without taking any random network-related decisions, let's see which network modes are available. We will keep on filtering until we hit the first correct target topology:. The preceding table shows a simple differentiation between two different logical network designs for OpenStack. Every mode shows its own requirements: this is very important and should be taken into consideration before the deployment. Arguing about our example choice, since we aim to deploy a very flexible, large-scale environment we will toggle the Neutron choice for networking management instead of nova-network. The choice was made for Neutron, since we started from a basic network deployment. We will cover more advanced features in the subsequent chapters of this book. We would like to exploit a major advantage of Neutron compared to nova-network, which is the virtualization of Layers 2 and 3 of the OSI network model. Let's see how we can expose our logical network design. For performance reasons; it is highly recommended to implement a topology that can handle different types of traffic by using separated logical networks.

In this way, as your network grows, it will still be manageable in case a sudden bottleneck or an unexpected failure affects a segment. The main feature of a data network that it provides the physical path for the virtual networks created by the OpenStack tenants. It separates the tenant data traffic from the infrastructure communication path required for the communication between the OpenStack component itself. In a smaller deployment, the traffic for management and communication between the OpenStack components can be on the same physical link. For a production environment, the network can be further subdivided to provide better isolation of traffic and contain the load on the individual networks.

The storage network provides physical connectivity and isolation for storage-related traffic between the VMs and the storage servers. As the traffic load for the storage network is quite high, it is a good idea to isolate the storage network load from the management and tenant traffic. It allows traffic from a private network to go out to the Internet. We can start with a limited number of servers just to setup the first deployment of our environment effectively. You have to consider the fact that hardware commodity selection will accomplish the mission of our massive scalable architecture. Since the architecture is being designed to scale horizontally, we can add more servers to the setup.

We will start by using commodity class, cost-effective hardware. In order to expect our infrastructure economy, it would be great to make some basic hardware calculations for the first estimation of our exact requirements. Considering the possibility of experiencing contentions for resources such as CPU, RAM, network, and disk, you cannot wait for a particular physical component to fail before you take corrective action, which might be more complicated.

Let's inspect a real-life example of the impact of underestimating capacity planning. A cloud-hosting company set up two medium servers, one for an e-mail server and the other to host the official website. The company, which is one of our several clients, grew in a few months and eventually ran out of disk space. The expected time to

resolve such an issue is a few hours, but it took days. The problem was that all the parties did not make proper use of the cloud, due to the on demand nature of the service. The cloud provider did not expect this! Incidents like this highlight the importance of proper capacity planning for your cloud infrastructure. Capacity management is considered a day-to-day responsibility where you have to stay updated with regard to software or hardware upgrades.

Through a continuous monitoring process of service consumption, you will be able to reduce the IT risk and provide a quick response to the customer's needs. From your first hardware deployment, keep running your capacity management processes by looping through tuning, monitoring, and analysis. Let's make our first calculation based on certain requirements.

For example, let's say we aim to run VMs in our OpenStack environment. Processor over subscription is defined as the total number of CPUs that are assigned to all the powered-on virtual machines multiplied by the hardware CPU core. If this number is greater than the GHz purchased, the environment is oversubscribed. The number of virtual machines per server with eight dual socket servers is calculated as follows:. Based on the previous example, 25 VMs can be deployed per compute node. Memory sizing is also important to avoid making unreasonable resource allocations. Considering the number of sticks supported by your server, you will need around GB installed. Therefore, the total number of RAM sticks installed can be calculated in the following way:.

To do this, it might be possible to serve our VMs by using a 10 GB link for each server, which will give:. This is a very satisfying value.

Mastering OpenStack - Second Edition | Packt

Then you'll learn about various hypervisors and container technology supported by OpenStack. You'll get an understanding about the segregation of compute nodes based on reliability and availability needs. Next, you'll understand the OpenStack infrastructure from a cloud user point of view. Moving on, you'll develop troubleshooting skills, and get a comprehensive understanding of services such as high availability and failover in OpenStack. Finally, you will gain experience of running a centralized logging server and monitoring OpenStack services. The book will show you how to carry out performance tuning based on OpenStack service logs. You will be able to master OpenStack benchmarking and performance tuning. By the end of the book, you'll be ready to take steps to deploy and manage an OpenStack cloud with the latest open source technologies.

Style and approach This book will help you understand the flexibility of OpenStack by showcasing integration of several out-of-the-box solutions in order to build a large-scale cloud environment.. It will also cover detailed discussions on the various design and deployment strategies for implementing a fault-tolerant and highly available cloud infrastructure.

Omar Khedher is a systems and network engineer who has worked for a few years in cloud computing environment and has been involved in several private cloud projects based on OpenStack. He has also worked on projects targeting public cloud AWS. Leveraging his skills as a system administrator in virtualization, storage, and networking, Omar works as a cloud system engineer for a leading advertising technology company, Fyber, based in Berlin. He is part of a highly skilled team working on several projects which include building and migrating infrastructure to the cloud using latest open source tools and DevOps philosophy. He has also authored a few academic publications based on a new research for cloud performance improvement.

Ltd, working on OpenStack Neutron plugins. The idea behind this service is to prevent direct database access from the compute nodes, thus enhancing database security in case one of the compute nodes gets compromised. By zooming out of the general components of OpenStack, we find that Nova interacts with several services such as Keystone for authentication, Glance for images, and Horizon for the web interface. For example, the Glance interaction is central; the API process can upload any query to Glance, while nova-compute will download images to launch instances. Nova also provides console services that allow end users to access the console of the virtual instance through a proxy such as nova-console , nova-novncproxy, and nova-consoleauth. Neutron provides a real Network as a Service NaaS capability between interface devices that are managed by OpenStack services such as Nova.

There are various characteristics that should be considered for Neutron:. Neutron has many core network features that are constantly growing and maturing. Some of these features are useful for routers, virtual switches, and SDN networking controllers. Neutron provides additional resources as extensions. The following are some of the commonly used extensions:. Neutron also provides advanced services to rule additional network OpenStack capabilities as follows:. Agents and plugins differ depending on the vendor technology of a particular cloud for the virtual and physical Cisco switches, NEC, OpenFlow, OpenSwitch, Linux bridging, and so on.

Neutron is a service that manages network connectivity between the OpenStack instances. It ensures that the network will not be turned into a bottleneck or limiting factor in a cloud deployment and gives users real self-service, even over their network configurations. Another advantage of Neutron is its ability to provide a way to integrate vendor networking solutions and a flexible way to extend network services.

It is designed to provide a plugin and extension mechanism that presents an option for network operators to enable different technologies via the Neutron API. Keep in mind that Neutron allows users to manage and create networks or connect servers and nodes to various networks. The scalability advantage will be discussed in a later topic in the context of the Software Defined Network SDN and Network Function Virtualization NFV technology, which is attractive to many networks and administrators who seek a high-level network multi-tenancy. Ceilometer provides a metering service in OpenStack. In a shared, multi-tenant environment such as OpenStack, metering resource utilization is of prime importance. Ceilometer collects data associated with resources. Resources can be any entity in the OpenStack cloud such as VMs, disks, networks, routers, and so on. Resources are associated with meters. The utilization data is stored in the form of samples in units defined by the associated meter.

Ceilometer has an inbuilt summarization capability. Ceilometer allows data collection from various sources, such as the message bus, polling resources, centralized agents, and so on. As an additional design change in the Telemetry service in OpenStack since the Liberty release, the Alarming service has been decoupled from the Ceilometer project to make use of a new incubated project code-named Aodh.

The Telemetry Alarming service will be dedicated to managing alarms and triggering them based on collected metering and scheduled events. More Telemetry service enhancements have been proposed to adopt a Time Series Database as a Service project code-named Gnoochi. This architectural change will tackle the challenge of metrics and event storage at scale in the OpenStack Telemetry service and improve its performance. Initial development for Heat was limited to a few OpenStack resources including compute, image, block storage, and network services.

Heat has boosted the emergence of resource management in OpenStack by orchestrating different cloud resources resulting in the creation of stacks to run applications with a few pushes of a button. From simple template engine text files referred to as HOT templates Heat Orchestration Template , users are able to provision the desired resources and run applications in no time. Heat is becoming an attractive OpenStack project due to its maturity and extended support resources catalog within the latest OpenStack releases. Other incubated OpenStack projects such as Sahara Big Data as a Service have been implemented to use the Heat engine to orchestrate the creation of the underlying resources stack.

It is becoming a mature component in OpenStack and can be integrated with some system configuration management tools such as Chef for full stack automation and configuration setup. Horizon is the web dashboard that pulls all the different pieces together from the OpenStack ecosystem. Horizon provides a web frontend for OpenStack services. Currently, it includes all the OpenStack services as well as some

incubated projects. It was designed as a stateless and data-less web application. It does not keep any data except the session information in its own data store. It is designed to be a reference implementation that can be customized and extended by operators for a particular cloud. It forms the basis of several public clouds, most notably the HP Public Cloud, and at its heart is its extensible modular approach to construction.

Horizon is based on a series of modules called panels that define the interaction of each service. Its modules can be enabled or disabled, depending on the service availability of the particular cloud. Message Queue provides a central hub to pass messages between different components of a service. This is where information is shared between different daemons by facilitating the communication between discrete processes in an asynchronous way.

One major advantage of the queuing system is that it can buffer requests and provide unicast and group-based communication services to subscribers. Its database stores most of the build-time and run-time states for the cloud infrastructure, including instance types that are available for use, instances in use, available networks, and projects. It provides a persistent storage for preserving the state of the cloud infrastructure. It is the second essential piece of sharing information in all OpenStack components. Let's try to see how OpenStack works by chaining all the service cores covered in the previous sections in a series of steps:. The scheduling process in OpenStack Nova can perform different algorithms such as simple, chance, and zone. An advanced way to do this is by deploying weights and filters by ranking servers as its available resources. It is important to understand how different services in OpenStack work together, leading to a running virtual machine.

The process of launching a virtual machine involves the interaction of the main OpenStack services that form the building blocks of an instance including compute, network, storage, and the base image. As shown in the previous diagram, OpenStack services interact with each other via a message bus to submit and retrieve RPC calls. The information of each step of the provisioning process is verified and passed by different OpenStack services via the message bus. On the other hand, the inter-communication of APIs within OpenStack requires an authentication mechanism to be trusted, which involves Keystone.

Starting with the identity service, the following steps summarize briefly the provisioning workflow based on API calls in OpenStack:. It is important to keep in mind that handling tokens in OpenStack on every API call and service request is a time limited operation. One of the major causes of a failed provisioning operation in OpenStack is the expiration of the token during subsequent API calls.

Additionally, the management of tokens has faced a few changes within different OpenStack releases. This includes two different approaches used in OpenStack prior to the Liberty release including:. Starting from the Kilo release, handling tokens in Keystone has progressed by introducing more sophisticated cryptographic authentication token methods, such as Fernet. Fernet is fully supported in the Mitaka release and the community is pushing to adopt it as the default. Deployment of OpenStack depends on the components were covered previously. It confirms your understanding of how to start designing a complete OpenStack environment. Of course, assuming the versatility and flexibility of such a cloud management platform, OpenStack offers several possibilities that can be considered an advantage.

However, owing to such flexibility, it's a challenge to come with the right design decision that suits your needs. At the end of the day, it all comes down to the use cases that your cloud is designed to service. Many enterprises have successfully designed their OpenStack environments by going through three phases of design: designing a conceptual model, designing a logical model, and finally, realizing the physical design. It's obvious that complexity increases from the conceptual to the logical design and from the logical to the physical design. As the first conceptual phase, we will have our high-level reflection on what we will need from certain generic classes from the OpenStack architecture:. Keep in mind that the illustrated diagram will be refined over and over again since we will aim to integrate more services within our first basic design. In other words, we are following an incremental design approach, within which we should exploit the flexibility of the OpenStack architecture.

At this level, we can have a vision and direction of the main goal without worrying about the details. Based on the conceptual reflection design, most probably you will have a good idea about different OpenStack core components, which will lay the formulation of the logical design. We will start by outlining the relationships and dependencies between the service core of OpenStack. In this section we will look at the deployment

architecture of OpenStack. We will start by identifying nodes to run an OpenStack service: the cloud controller, network nodes, and the compute node.

You may wonder why such a consideration goes through a physical design classification. However, seeing the cloud controller and compute nodes as simple packages that encapsulate a bunch of OpenStack services will help you refine your design at an early stage. Furthermore, this approach will help plan in advance further high availability and scalability requirements, and will allow you to introduce them later in more detail. Thus, the physical model design will be elaborated based on the previous theoretical phases by assigning parameters and values to our design.

Let's start with our first logical iteration:. Obviously, in a highly available setup, we should achieve a degree of redundancy in each service within OpenStack. You may wonder about the critical OpenStack services claimed in the first part of this chapter: the database and message queue. Why can't they be separately clustered or packaged on their own? This is a pertinent question.

Remember that we are still in the second logical phase where we try to dive slowly into the infrastructure without getting into the details. Besides, we keep on going from a generic and simple design to targeting specific use-cases. What about high availability? The aforementioned technologies will be discussed in more detail in Chapter 9 , Openstack HA and Failover. Compute nodes are relatively simple as they are intended just to run the virtual machine's workload. In order to manage the VMs, the nova-compute service can be assigned for each compute node. Besides, we should not forget that the compute nodes will not be isolated; a Neutron agent and an optional Ceilometer compute agent may run these nodes. You should now have a deeper understanding of the storage types within Swift, Cinder, and Manila.

You will have to ask yourself questions such as: How much data do I need to store? Will my future use cases result in a wide range of applications that run heavy-analysis data? What are my storage requirements for incrementally backing up a virtual machine's snapshots? Do I really need control over the filesystem on the storage or is just a file share enough? Do I need a shared storage between VMs? To answer this question, you can think about ephemeral storage as the place where the end user will not be able to access the virtual disk associated with its VM when it is terminated.

Ephemeral storage should mainly be used in production when the VM state is non-critical, where users or application don't store data on the VM. If you need your data to be persistent, you must plan for a storage service such as Cinder or Manila. Remember that the current design applies for medium to large infrastructures. Ephemeral storage can also be a choice for certain users; for example, when they consider building a test environment.

Considering the same case for Swift, we claimed previously that object storage might be used to store machine images, but when do we use such a solution? Simply put, when you have a sufficient volume of critical data in your cloud environment and start to feel the need for replication and redundancy. OpenStack allows a wide ranging of configurations that variation, and tunneled networks such as GRE, VXLAN, and so on, with Neutron are not intuitively obvious from their appearance to be able to be implemented without fetching their use case in our design.

Thus, this important step implies that you may differ between different network topologies because of the reasons behind why every choice was made and why it may work for a given use case. OpenStack has moved from simplistic network features to more complicated ones, but of course the reason is that it offers more flexibility! This is why OpenStack is here. It brings as much flexibility as it can! Without taking any random network-related decisions, let's see which network modes are available. We will keep on filtering until we hit the first correct target topology:. The preceding table shows a simple differentiation between two different logical network designs for OpenStack. Every mode shows its own requirements: this is very important and should be taken into consideration before the deployment. Arguing about our example choice, since we aim to deploy a very flexible, large-scale environment we will toggle the Neutron choice for networking management instead of nova-network.

The choice was made for Neutron, since we started from a basic network deployment. We will cover more advanced features in the subsequent chapters of this book. We would like to exploit a major advantage of Neutron compared to nova-network, which is the virtualization of Layers 2 and 3 of the OSI network model.

Let's see how we can expose our logical network design.

For performance reasons; it is highly recommended to implement a topology that can handle different types of traffic by using separated logical networks. In this way, as your network grows, it will still be manageable in case a sudden bottleneck or an unexpected failure affects a segment. The main feature of a data network that it provides the physical path for the virtual networks created by the OpenStack tenants. It separates the tenant data traffic from the infrastructure communication path required for the communication between the OpenStack component itself. In a smaller deployment, the traffic for management and communication between the OpenStack components can be on the same physical link.

For a production environment, the network can be further subdivided to provide better isolation of traffic and contain the load on the individual networks. The storage network provides physical connectivity and isolation for storage-related traffic between the VMs and the storage servers. As the traffic load for the storage network is quite high, it is a good idea to isolate the storage network load from the management and tenant traffic. It allows traffic from a private network to go out to the Internet. We can start with a limited number of servers just to setup the first deployment of our environment effectively. You have to consider the fact that hardware commodity selection will accomplish the mission of our massive scalable architecture.

Since the architecture is being designed to scale horizontally, we can add more servers to the setup. We will start by using commodity class, cost-effective hardware. In order to expect our infrastructure economy, it would be great to make some basic hardware calculations for the first estimation of our exact requirements. Considering the possibility of experiencing contentions for resources such as CPU, RAM, network, and disk, you cannot wait for a particular physical component to fail before you take corrective action, which might be more complicated.

Let's inspect a real-life example of the impact of underestimating capacity planning. A cloud-hosting company set up two medium servers, one for an e-mail server and the other to host the official website. The company, which is one of our several clients, grew in a few months and eventually ran out of disk space. The expected time to resolve such an issue is a few hours, but it took days.

The problem was that all the parties did not make proper use of the cloud, due to the on demand nature of the service. The cloud provider did not expect this! Incidents like this highlight the importance of proper capacity planning for your cloud infrastructure. Capacity management is considered a day-to-day responsibility where you have to stay updated with regard to software or hardware upgrades. Through a continuous monitoring process of service consumption, you will be able to reduce the IT risk and provide a quick response to the customer's needs. From your first hardware deployment, keep running your capacity management processes by looping through tuning, monitoring, and analysis. Let's make our first calculation based on certain requirements. For example, let's say we aim to run VMs in our OpenStack environment. Processor over subscription is defined as the total number of CPUs that are assigned to all the powered-on virtual machines multiplied by the hardware CPU core.

If this number is greater than the GHz purchased, the environment is oversubscribed. The number of virtual machines per server with eight dual socket servers is calculated as follows:. Based on the previous example, 25 VMs can be deployed per compute node. Memory sizing is also important to avoid making unreasonable resource allocations. Considering the number of sticks supported by your server, you will need around GB installed.

Therefore, the total number of RAM sticks installed can be calculated in the following way:. To do this, it might be possible to serve our VMs by using a 10 GB link for each server, which will give:. This is a very satisfying value. We need to consider another factor: highly available network architecture. Thus, an alternative is using two data switches with a minimum of 24 ports for data. What about the growth of the rack size? This feature allows each server rack to divide its links between the pair of switches to achieve a powerful active-active forwarding while using the full bandwidth capability with no requirement for a spanning tree. MCLAG is a Layer 2 link aggregation protocol between the servers that are connected to the switches, offering a redundant, load-balancing connection to the core network and replacing the spanning-tree protocol.

The network configuration also depends heavily on the chosen network topology. As shown in the previous example network diagram, you should be aware that all nodes in the OpenStack environment must communicate with each other. Based on this requirement, administrators will need to standardize the units will be planned to use and count the needed number of public and floating IP addresses. This calculation depends on which network type the OpenStack environment will run including the usage of Neutron or former nova-network service. Our first basic example assumes the usage of the Public IPs for the following units:. In this case, we will initially need at least 18 public IP addresses. Moreover, when implementing a high available setup using virtual IPs fronted by load balancers, these will be considered as additional public IP addresses.

The use of Neutron for our OpenStack network design will involve a preparation for the number of virtual devices and interfaces interacting with the network node and the rest of the private cloud environment including:. In this case, we will initially need at least floating IP addresses given that every virtual router is capable of connecting to the public network. Additionally, increasing the available bandwidth should be taken into consideration in advance.

Bonding will empower cloud network high availability and achieve boosted bandwidth performance. Considering the previous example, you need to plan for an initial storage capacity per server that will serve 25 VMs each. Most probably, you have an idea about the replication of object storage in OpenStack, which implies the usage of three times the required space for replication. In other words, if you are planning for X TB for object storage, your storage requirement will be 3X. Other considerations, such as the best storage performance using SSD, can be useful for a better throughput where you can invest more boxes to get an increased IOPS. Well, let's bring some best practices under the microscope by exposing the OpenStack design flavor. Assuming that your OpenStack environment continues to grow, you may decide to purchase more hardware: large, and at an incredible price! A second compute instance is placed to scale up.

Shortly after this, you may find out that demand is increasing. You may start splitting requests into different compute nodes but keep on continuing scaling up with the hardware. At some point, you will be alerted about reaching your budget limit! There are certainly times when the best practices aren't in fact the best for your design. The previous example illustrated a commonly overlooked requirement for the OpenStack deployment.

If the minimal hardware requirement is strictly followed, it may result in an exponential cost with regards to hardware expenses, especially for new project starters. Thus, you should choose exactly what works for you and consider the constraints that exist in your environment. Keep in mind that best practices are a guideline; apply them when you find what you need to be deployed and how it should be set up. On the other hand, do not stick to values, but stick to the spirit of the rules.

Let's bring the previous example under the microscope again: scaling up shows more risk and may lead to failure than scaling out or horizontally. The reason behind such a design is to allow for a fast scale of transactions at the cost of duplicated compute functionality and smaller systems at a lower cost. That is how OpenStack was designed: degraded units can be discarded and failed workloads can be replaced. Transactions and requests in the compute node may grow tremendously in a short time to a point where a single big compute node with 16 core CPUs starts failing performance-wise, while a few small compute nodes with 4 core CPUs can proceed to complete the job successfully. As we have shown in the previous section, planning for capacity is a quite intensive exercise but very crucial to setting up an initial, successful OpenStack cloud strategy. Planning for growth should be driven by the natural design of OpenStack and how it is implemented. We should consider that growth is based on demand where workloads in OpenStack take an elastic form and not a linear one.

Although the previous resource's computation example can be helpful to estimate a few initial requirements for our designed OpenStack layout, reaching acceptable capacity planning still needs more action. This includes a detailed analysis of cloud performance in terms of growth of workload. In addition, by using more sophisticated monitoring tools, operators should be consistent in tracking the usage of each unit running in the OpenStack environment, which includes, for example, its overall resource consumption over time and cases of unit overutilization resulting in performance degradation. As we have conducted a rough estimation of our future hardware capabilities, this calculation model can be hardened by sizing the instance flavor for each compute host after first deployment and can be adjusted on demand if resources are carefully

monitored. This chapter has revisited the basic components of OpenStack and exposed new features such as Telemetry, Orchestration, and File Share projects.

We continued refining our logical design for future deployment by completing a first design layout. As an introductory chapter, we have rekindled the flames on each OpenStack component by discussing briefly each use case and role in its ecosystem. We have also covered a few tactical tips to plan and mitigate the future growth of the OpenStack setup in a production environment.

As a main reference for the rest of the book, we will be breaking down each component and new functionality in OpenStack by extending the basic layout covered in this chapter. We will continue the OpenStack journey to deploy what was planned in a robust and effective way: the DevOps style. Omar Khedher is a systems and network engineer. He spent few years as cloud system engineer with talented teams to architect infrastructure in the public cloud at Fyber in Berlin. Ltd, working on OpenStack Neutron plugins. He has over 11 years of experience in the deployment of Linux-based solutions. In the past, he has been involved in developing Linux-based clustering and deployment solutions. He has contributed to setting up and maintaining a private cloud solution in Juniper Networks.

He was a speaker at the OpenStack Tokyo summit, where he presented the idea of adding firewall logs and other Neutron enhancements. He is speaker at the Austin summit where he talks about making enhancements to the Nova scheduler. He loves to explore technology. Discover the basics of virtual networking in OpenStack to implement various cloud network architectures. About this book In this second edition, you will get to grips with the latest features of OpenStack.

Mastering OpenStack - - Omar Khedher - Paperback () » Bokklubben

Whatever cloud category is used, this trend was felt by many organizations, which needs to introduce an orchestration engine to their infrastructure to embrace elasticity, scalability, and achieve a unique user experience to a certain extent. Nowadays, a remarkable orchestration solution, which falls into the private cloud category, has brought thousands of enterprises to the next era of data center generation: OpenStack. At the time of writing, OpenStack has been deployed in several large to medium enterprise infrastructures, running different types of production workload. The maturity of this cloud platform has been boosted due to the joint effort of several large organizations and its vast developer community around the globe.

Within every new release, OpenStack brings more great features, which makes it a glorious solution for organizations seeking to invest in it, with returns in operational workloads and flexible infrastructure. In this edition, we will keep explaining the novelties of OpenStack within the latest releases and discuss the great opportunities, which OpenStack can offer for an amazing cloud experience. Deploying OpenStack is still a challenging step, which needs a good understanding of its beneficial returns to a given organization in terms of automation, orchestration, and flexibility. If expectations are set properly, this challenge will turn into a valuable opportunity, which deserves an investment. After collecting infrastructure requirements, starting an OpenStack journey will need a good design and consistent deployment plan with different architectural assets.

The challenge, which has been set by the public cloud is about agility, speed, and self-service. Most companies have expensive IT systems, which they have developed and deployed over the years, but they are siloed and need human intervention. In many cases, IT systems are struggling to respond to the agility and speed of the public cloud services. The traditional data center model and siloed infrastructure might become unsustainable in today's agile service delivery environment. In fact, today's enterprise data center must focus on speed, flexibility, and automation for delivering services to get to the level of next-generation data center efficiency.

The big move to a software infrastructure has allowed administrators and operators to deliver a fully automated infrastructure within a minute. The next-generation data center reduces the infrastructure to a single, big, agile, scalable, and automated unit. The end result is a programmable, scalable, and multi-tenant-aware infrastructure. This is where OpenStack comes into the picture: it promises the features of a next-generation data center operating system.

Today, many of them are running a very large scalable private cloud based on OpenStack in their production environment. If you intend to be a part of a winning, innovative cloud enterprise, you should jump to the next-generation data center and gain valuable experience by adopting OpenStack in your IT infrastructure. Before delving into the OpenStack architecture , we need to refresh or fill gaps and learn more about the basic concepts and usage of each core component. In order to get a better understanding on how it works, it will be beneficial to first briefly parse the things, which make it work. In the following sections, we will look at various OpenStack services, which work together to provide the cloud experience to the end user. Despite the different services catering to different needs, they follow a common theme in their design that can be summarized as follows:.

With this common theme in mind, let's now put the essential core components under the microscope and go a bit further by asking the question: What is the purpose of such component? From an architectural perspective, Keystone presents the simplest service in the OpenStack composition. It is the core component and provides an identity service comprising authentication and authorization of

tenants in OpenStack. Communications between different OpenStack services are authorized by Keystone to ensure that the right user or service is able to utilize the requested OpenStack service. With the evolution of Keystone, many features have been implemented within recent OpenStack releases leveraging a centralized and federated identity solution.

This will allow users to use their credentials in an existing, centralized, sign-on backend and decouples the authentication mechanism from Keystone. Swift is one of the storage services available to OpenStack users. Compared to traditional storage solutions, file shares, or block-based access, an Object-Storage takes the approach of dealing with stored data as objects that can be stored and retrieved from the Object-Store. A very high-level overview of Object Storage goes like this. To store the data, the Object-Store splits it into smaller chunks and stores it in separate containers. These containers are maintained in redundant copies spread across a cluster of storage nodes to provide high availability, auto-recovery, and horizontal scalability. We will leave the details of the Swift architecture for later. Briefly, it has a number of benefits:. You may wonder whether there is another way to provide storage to OpenStack users.

Indeed, the management of the persistent block storage is available in OpenStack by using the Cinder service. Its main capability is to provide block-level storage to the virtual machine. Cinder provides raw volumes that can be used as hard disks in virtual machines. It is very important to keep in mind that like Keystone services, Cinder features can be delivered by orchestrating various backend volume providers through configurable drivers for the vendor's storage products such as from IBM, NetApp, Nexenta, and VMware.

Cinder is proven as an ideal solution or a replacement of the old nova-volume service that existed before the Folsom release on an architectural level. It is important to know that Cinder has organized and created a catalog of block-based storage devices with several differing characteristics. However, we must obviously consider the limitation of commodity storage such as redundancy and auto-scaling. When Cinder was introduced in the OpenStack Grizzly release, a joint feature was implemented to allow creating backups for Cinder volumes.

A common use case has seen Swift evolves as a storage backup solution. This great backup extensible feature is defined by the means of Cinder backup drivers that have become richer in every new release. Within the OpenStack Mitaka release, Cinder has shown its vast number of backup options by marrying two different cloud computing environments, bringing an additional backup driver targeting Google Cloud Platform. This exciting opportunity allows OpenStack operators to leverage an hybrid cloud backup solution that empowers , a disaster recovery strategy for persistent data. What about security? This latent issue has been resolved since the Kilo release so Cinder volumes can be encrypted before starting any backup operations. Apart from the block and object we discussed in the previous section, since the Juno release, OpenStack has also had a file-share-based storage service called Manila.

It provides storage as a remote file system. The Manila service provides the orchestration of shares on the share servers. Each storage solution in OpenStack has been designed for a specific set of purposes and implemented for different targets. Before taking any architectural design decisions, it is crucial to understand the difference between existing storage options in OpenStack today, as outlined in the following table:. The Glance service provides a registry of images and metadata that the OpenStack user can launch as a virtual machine. Various image formats are supported and can be used based on the choice of hypervisor. Both handle storage.

What is the difference between them? Why do I need to integrate such a solution? Swift is a storage system, whereas Glance is an image registry. The difference between the two is that Glance is a service that keeps track of virtual machine images and metadata associated with the images. Metadata can be information such as a kernel, disk images, disk format, and so on. Glance can use a variety of backends for storing images. The default is to use directories, but in a massive production environment it can use other approaches such as NFS and even Swift. Swift, on the other hand, is a storage system. It is designed for object-storage where you can keep data such as virtual disks, images, backup archiving, and so on. The mission of Glance is to be an image registry. From an architectural point of view, the goal of Glance is to focus on advanced ways to store and query image information via the Image Service API. A typical use case for Glance is to allow a client which can be a user or an external service to register a new virtual disk image, while a storage system focuses on providing a highly scalable and redundant data store.

At this level, as a technical operator, your challenge is to provide the right storage solution to meet cost and performance requirements. This will be discussed at the end of the book. As you may already know, Nova is the original core component of OpenStack. From an architectural level, it is considered one of the most complicated components of OpenStack. Nova provides the compute service in OpenStack and manages virtual machines in response to service requests made by OpenStack users. What makes Nova complex is its interaction with a large number of other OpenStack services and internal components, which it must collaborate with to respond to user requests for running a VM.

Let's break down the Nova service itself and look at its architecture as a distributed application that needs orchestration between different components to carry out tasks. The nova-api component accepts and responds to the end user and computes API calls. The nova-api initiates most orchestrating activities such as the running of an instance or the enforcement of some particular policies. The nova-network component accepts networking tasks from the queue and then performs these tasks to manipulate the network such as setting up bridging interfaces or changing IP table rules. Neutron is a replacement for the nova-network service. The nova-scheduler component takes a VM instance's request from the queue and determines where it should run specifically which compute host it should run on. At an application architecture level, the term scheduling or scheduler invokes a systematic search for the best outfit for a given infrastructure to improve its performance.

The nova-conductor service provides database access to compute nodes. The idea behind this service is to prevent direct database

access from the compute nodes, thus enhancing database security in case one of the compute nodes gets compromised. By zooming out of the general components of OpenStack, we find that Nova interacts with several services such as Keystone for authentication, Glance for images, and Horizon for the web interface. For example, the Glance interaction is central; the API process can upload any query to Glance, while nova-compute will download images to launch instances.

Nova also provides console services that allow end users to access the console of the virtual instance through a proxy such as nova-console , nova-novncproxy, and nova-consoleauth. Neutron provides a real Network as a Service NaaS capability between interface devices that are managed by OpenStack services such as Nova. There are various characteristics that should be considered for Neutron:. Neutron has many core network features that are constantly growing and maturing. Some of these features are useful for routers, virtual switches, and SDN networking controllers. Neutron provides additional resources as extensions. The following are some of the commonly used extensions:. Neutron also provides advanced services to rule additional network OpenStack capabilities as follows:. Agents and plugins differ depending on the vendor technology of a particular cloud for the virtual and physical Cisco switches, NEC, OpenFlow, OpenSwitch, Linux bridging, and so on.

Neutron is a service that manages network connectivity between the OpenStack instances. It ensures that the network will not be turned into a bottleneck or limiting factor in a cloud deployment and gives users real self-service, even over their network configurations. Another advantage of Neutron is its ability to provide a way to integrate vendor networking solutions and a flexible way to extend network services. It is designed to provide a plugin and extension mechanism that presents an option for network operators to enable different technologies via the Neutron API.

Keep in mind that Neutron allows users to manage and create networks or connect servers and nodes to various networks. The scalability advantage will be discussed in a later topic in the context of the Software Defined Network SDN and Network Function Virtualization NFV technology, which is attractive to many networks and administrators who seek a high-level network multi-tenancy. Ceilometer provides a metering service in OpenStack. In a shared, multi-tenant environment such as OpenStack, metering resource utilization is of prime importance. Ceilometer collects data associated with resources. Resources can be any entity in the OpenStack cloud such as VMs, disks, networks, routers, and so on. Resources are associated with meters. The utilization data is stored in the form of samples in units defined by the associated meter.

Ceilometer has an inbuilt summarization capability. Ceilometer allows data collection from various sources, such as the message bus, polling resources, centralized agents, and so on. As an additional design change in the Telemetry service in OpenStack since the Liberty release, the Alarming service has been decoupled from the Ceilometer project to make use of a new incubated project code-named Aodh. The Telemetry Alarming service will be dedicated to managing alarms and triggering them based on collected metering and scheduled events. More Telemetry service enhancements have been proposed to adopt a Time Series Database as a Service project code-named Gnoochi.

This architectural change will tackle the challenge of metrics and event storage at scale in the OpenStack Telemetry service and improve its performance. Initial development for Heat was limited to a few OpenStack resources including compute, image, block storage, and network services. Heat has boosted the emergence of resource management in OpenStack by orchestrating different cloud resources resulting in the creation of stacks to run applications with a few pushes of a button. From simple template engine text files referred to as HOT templates Heat Orchestration Template , users are able to provision the desired resources and run applications in no time. Heat is becoming an attractive OpenStack project due to its maturity and extended support resources catalog within the latest OpenStack releases. Other incubated OpenStack projects such as Sahara Big Data as a Service have been implemented to use the Heat engine to orchestrate the creation of the underlying resources stack.

It is becoming a mature component in OpenStack and can be integrated with some system configuration management tools such as Chef for full stack automation and configuration setup. Horizon is the web dashboard that pulls all the different pieces together from the OpenStack ecosystem. Horizon provides a web frontend for OpenStack services. Currently, it includes all the OpenStack services as well as some incubated projects.

It was designed as a stateless and data-less web application. It does not keep any data except the session information in its own data store. It is designed to be a reference implementation that can be customized and extended by operators for a particular cloud. It forms the basis of several public clouds, most notably the HP Public Cloud, and at its heart is its extensible modular approach to construction. Horizon is based on a series of modules called panels that define the interaction of each service. Its modules can be enabled or disabled, depending on the service availability of the particular cloud. Message Queue provides a central hub to pass messages between different components of a service. This is where information is shared between different daemons by facilitating the communication between discrete processes in an asynchronous way.

One major advantage of the queuing system is that it can buffer requests and provide unicast and group-based communication services to subscribers. Its database stores most of the build-time and run-time states for the cloud infrastructure, including instance types that are available for use, instances in use, available networks, and projects. It provides a persistent storage for preserving the state of the cloud infrastructure. It is the second essential piece of sharing information in all OpenStack components. Let's try to see how OpenStack works by chaining all the service cores covered in the previous sections in a series of steps:.

The scheduling process in OpenStack Nova can perform different algorithms such as simple, chance, and zone. An advanced way to do this is by deploying weights and filters by ranking servers as its available resources. It is important to understand how different services in OpenStack work together, leading to a running virtual machine. The process of launching a virtual machine involves the interaction of the main OpenStack services that form the building blocks of an instance including compute, network, storage, and the

base image. As shown in the previous diagram, OpenStack services interact with each other via a message bus to submit and retrieve RPC calls.

The information of each step of the provisioning process is verified and passed by different OpenStack services via the message bus. On the other hand, the inter-communication of APIs within OpenStack requires an authentication mechanism to be trusted, which involves Keystone. Starting with the identity service, the following steps summarize briefly the provisioning workflow based on API calls in OpenStack:.

It is important to keep in mind that handling tokens in OpenStack on every API call and service request is a time limited operation. Dive in to the cutting edge techniques of Linux KVM virtualization, and build the virtualization solutions …. Skip to main content. Start your free trial. Book description Discover your complete guide to designing, deploying, and managing OpenStack-based clouds in mid-to-large IT infrastructures with best practices, expert understanding, and more About This Book Design and deploy an OpenStack-based cloud in your mid-to-large IT infrastructure using automation tools and best practices Keep yourself up-to-date with valuable insights into OpenStack components and new services in the latest OpenStack release Discover how the new features in the latest OpenStack release can help your enterprise and infrastructure Who This Book Is For This book is for system administrators, cloud engineers, and system architects who would like to deploy an OpenStack-based cloud in a mid-to-large IT infrastructure.

What You Will Learn Explore the main architecture design of OpenStack components and core-by-core services, and how they work together Design different high availability scenarios and plan for a no-single-point-of-failure environment Set up a multinode environment in production using orchestration tools Boost OpenStack's performance with advanced configuration Delve into various hypervisors and container technology supported by OpenStack Get familiar with deployment methods and discover use cases in a real production environment Adopt the DevOps style of automation while deploying and operating in an OpenStack environment Monitor the cloud infrastructure and make decisions on maintenance and performance improvement In Detail In this second edition, you will get to grips with the latest features of OpenStack.

Style and approach This book will help you understand the flexibility of OpenStack by showcasing integration of several out-of-the-box solutions in order to build a large-scale cloud environment.. Show and hide more. Table of contents Product information. What makes OpenStack unique is its exposure; it is widely open to other open source solutions along with being a shining example of a multiport-integrated solution with great flexibility. All that you really need is a good design to fulfill most of your requirements and the right decisions on how and what to deploy. If you browse the pages of this book, you might wonder what makes a laminated cover entitled Mastering , such a great deal to you as a system administrator, cloud architect, DevOps engineer, or any technical personnel operating on the Linux platform.

Basically, you may be working on a project, going on a vacation, building a house, or redesigning your fancy apartment. In each of these cases, you will always need a strategy. Ultimately, based on what you learned from the OpenStack literature, and what you have deployed, or practiced, you will probably ask the famous key question: How does OpenStack work? Well, the OpenStack community is very rich in terms of topics and tutorials—some of which you may have already tried out. It is time to go ahead and raise the curtain on the OpenStack design and architecture. Basically, the goal of this chapter is to get you from where you are today to the point where you can confidently build a private cloud based on OpenStack with your own design choice. At the end of this chapter, you will have a good perspective on ways to design your project by putting the details under the microscope.

You will also learn about how OpenStack services work together and be ready for the next stage of our adventure by starting the deployment of an OpenStack environment with best practices. Getting acquainted with the logical architecture of the OpenStack ecosystem and the way its different core components interact with each other. Learning how to design an OpenStack environment by choosing the right core services for the right environment. Designing the first OpenStack architecture for a large-scale environment while bearing in mind that OpenStack can be designed in numerous ways.

Learning some best practices and the process of capacity planning for a robust OpenStack environment. Let's start the mission by putting the spot light on the place where the core OpenStack components come in the first place. The challenge that has been set by the public cloud is about agility, speed, and service efficiency. Most companies have expensive IT systems they have developed and deployed over the years, but they are siloed. In many cases, the IT systems are struggling to respond to the agility and speed of the public cloud services that are offered within their own private silos in their own private data center.

The traditional data center model and siloed infrastructure might lead to unsustainability. In fact, today's enterprise data center focuses on what it takes to become a next-generation data center. The shift to the new data center generation has evolved the adoption of a model for the management and provision of software. This has been accompanied by a shift from workload isolation in the traditional model to a mixed model.

With an increasing number of users utilizing cloud services, the next-generation data centers are able to handle multitenancy. The traditional one was limited to a single tenancy. Moreover, enterprises today look for scaling down next to scaling up. It is a huge step in the data center technology to shift the way of handling an entire infrastructure.

The big move to a software infrastructure has allowed administrators and operators to deliver a fully automated infrastructure within a minute. The next-generation data center reduces the infrastructure to a single, big, agile, scalable, and automated unit. The end result is that the administrators will have to program the infrastructure. This is where OpenStack comes into the picture—the next-generation data center operating system. Today, many of them are running a very large scalable private cloud based on OpenStack

in their production environment. If you intend to be a part of a winning, innovative cloud enterprise, you should jump to the next-generation data center and gain a valuable experience by adopting OpenStack in your IT infrastructure. Before delving into the architecture of OpenStack, we need to refresh or fill gaps, if they do exist, to learn more about the basic concepts and usage of each core component.

In order to get a better understanding on how it works , it will be beneficial to first briefly parse the things that make it work. Assuming that you have already installed OpenStack or even deployed it in a small or medium-sized environment, let's put the essential core components under the microscope and go a bit further by taking the use cases and asking the question: What is the purpose of such a component? From an architectural perspective, Keystone presents the simplest service in the OpenStack composition. It is the core component that provides identity service and it integrates functions for authentication, catalog services, and policies to register and manage different tenants and users in the OpenStack projects.

The API requests between OpenStack services are being processed by Keystone to ensure that the right user or service is able to utilize the requested OpenStack service. A similar real-life example is a city game. You can purchase a gaming day card and profit by playing a certain number of games during a certain period of time. Before you start gaming, you have to ask for the card to get an access to the city at the main entrance of the city game. Every time you would like to try a new game, you must check in at the game stage machine. This will generate a request, which is mapped to a central authentication system to check the validity of the card and its warranty, to profit the requested game.

By analogy, the token in Keystone can be compared to the gaming day card except that it does not diminish anything from your request. The identity service is being considered as a central and common authentication system that provides access to the users. Although it was briefly claimed that Swift would be made available to the users along with the OpenStack components, it is interesting to see how Swift has empowered what is referred to as cloud storage. Most of the enterprises in the last decade did not hide their fears about a critical part of the IT infrastructure—the storage where the data is held. Thus, the purchasing of expensive hardware to be in the safe zone had become a habit. There are always certain challenges that are faced by storage engineers and no doubt, one of these challenges include the task of minimizing downtime while increasing the data availability.

Despite the rise of many smart solutions for storage systems during the last few years, we still need to make changes to the traditional way. Make it cloudy! Swift was introduced to fulfill this mission. We will leave the details pertaining to the Swift architecture for later, but you should keep in mind that Swift is an object storage software, which has a number of benefits:. When I had my first presentation on the core components and architecture of OpenStack with my first cloud software company, I was surprised by a question raised by the CTO: What is the difference between Glance and Swift? Both handle storage. Well, despite my deployment of OpenStack Cacti and Diablo were released at the time and familiarity with the majority of the component's services, I found the question quite tough to answer! As a system architect or technical designer, you may come across the following questions: What is the difference between them? Why do I need to integrate such a solution? On one hand, it is important to distinguish the system interaction components so that it will be easier to troubleshoot and operate within the production environments.

On the other hand, it is important to satisfy the needs and conditions that go beyond your IT infrastructure limits. To alleviate any confusion, we keep it simple. Swift and Glance are storage systems. However, the difference between the two is in what they store. Swift is designed to be an object storage where you can keep data such as virtual disks, images, backup archiving, and so forth, while Glance stores metadata of images. Metadata can be information such as kernel, disk images, disk format, and so forth. Do not be surprised that Glance was originally designed to store images. Since the first release of OpenStack included only Nova and Swift Austin code name October 21, , Glance was integrated with the second release Bexar code name February 23, The mission of Glance is to be an image registry.

From this point, we can conclude how OpenStack has paved the way to being more modular and loosely coupled core component model. Using Glance to store virtual disk images is a possibility. From an architectural level, including more advanced ways to query image information via the Image Service API provided by Glance through an independent image storage backend such as Swift brings more valuable performance and well-organized system core services.

In this way, a client can be a user or an external service will be able to register a new virtual disk image, for example, to stream it from a highly scalable and redundant store. At this level, as a technical operator, you may face another challenge—performance. This will be discussed at the end of the book. You may wonder whether there is another way to have storage in OpenStack. Indeed, the management of the persistent block storage is being integrated into OpenStack by using Cinder. Its main capability to present block-level storage provides raw volumes that can be built into logical volumes in the filesystem and mounted in the virtual machine.

Volume management : This allows the creation or deletion of a volume. Snapshot management : This allows the creation or deletion of a snapshot of volumes. Several storage options have been proposed in the OpenStack core. Without a doubt, you may be asked this question: What kind of storage will be the best for you? With a decision-making process, a list of pros and cons should be made. The following is a very simplistic table that describes the difference between the storage types in OpenStack to avoid any confusion when choosing the storage management option for your future architecture design:. It is very important to keep in mind that unlike Glance and Keystone services, Cinder features are delivered by orchestrating volume providers through the configurable setup driver's architectures such as IBM, NetApp, Nexenta, and VMware. Whatever choice you have made, it is always considered good advice since nothing is perfect.

If Cinder is proven as an ideal solution or a replacement of the old nova-volume service that existed before the Folsom release on an architectural level, it is important to know that Cinder has organized and created a catalog of block-based storage devices with

several differing characteristics. However, it is obvious if we consider the limitation of commodity storage redundancy and autoscaling. Eventually, the block storage service as the main core of OpenStack can be improved if a few gaps are filled, such as the addition of values:. The aforementioned Cinder specification reveals its Non-vendor-lock-in characteristic, where it is possible to change the backend easily or perform data migration between two different storage backends.

Therefore, a better storage design architecture in a Cinder use case will bring a third party into the scalability game. For instance, you can keep in mind that Cinder is essential for our private cloud design, but it misses some capacity scaling features. As you may already know, Nova is the most original core component of OpenStack. From an architectural level, it is considered one of the most complicated components of OpenStack. In a nutshell, Nova runs a large number of requests, which are collaborated to respond to a user request into running VM. Let's break down the blob image of nova by assuming that its architecture as a distributed application needs orchestration to carry out tasks between different components.

The nova-api component accepts and responds to the end user and computes the API calls. Nova-api initiates most of the orchestrating activities such as the running of an instance or the enforcement of some particular policies. Accept actions from the queue and perform system commands such as the launching of the KVM instances to take them out when updating the state in the database. Ceph is an open source storage software platform for object, block, and file storage in a highly available storage environment.

The nova-volume component manages the creation, attaching, and detaching of N volumes to compute instances similar to Amazon's EBS. The nova-network component accepts networking tasks from the queue and then performs these tasks to manipulate the network such as setting up bridging interfaces or changing the IP table rules. The nova-scheduler component takes a VM instance's request from the queue and determines where it should run specifically which compute server host it should run on. At an application architecture level, the term scheduling or scheduler invokes a systematic search for the best outfit for a given infrastructure to improve its performance. Nova also provides console services that allow end users to access the console of the virtual instance through a proxy such as nova-console, nova-novncproxy, and nova-consoleauth. By zooming out the general components of OpenStack, we find that Nova interacts with several services such as Keystone for authentication, Glance for images, and Horizon for the web interface.

For example, the Glance interaction is central; the API process can upload any query to Glance, while nova-compute will download images to launch instances. Queue provides a central hub to pass messages between daemons. This is where information is shared between different Nova daemons by facilitating the communication between discrete processes in an asynchronous way. Any service can easily communicate with any other service via the APIs and queue a service. One major advantage of the queuing system is that it can buffer a large buffer workload. Rather than using an RPC service, a queue system can queue a large workload and give an eventual consistency. A database stores most of the build-time and runtime state for the cloud infrastructure, including instance types that are available for use, instances in use, available networks, and projects.

It is the second essential piece of sharing information in all OpenStack components. There are various characteristics that should be considered for Neutron:. Its pluggable backend architecture lets users take advantage of the commodity gear or vendor-supported equipment. Neutron has many core network features that are constantly growing and maturing. Some of these features are useful for routers, virtual switches, and the SDN networking controllers. Starting from the Folsom release, the Quantum network service has been replaced by a project named Neutron , which was incorporated into the mainline project in the subsequent releases. The examples elaborated in this book are based on the Havana release and later. Port : Ports in Neutron refer to the virtual switch connections. These connections are where instances and network services attached to networks. When attached to the subnets, the defined MAC and IP addresses of the interfaces are plugged into them. Networks : Neutron defines networks as isolated Layer 2 network segments.

Operators will see networks as logical switches that are implemented by the Linux bridging tools, Open vSwitch, or some other software. Unlike physical networks, this can be defined by either the operators or users in OpenStack. Subnets in Neutron represent a block of IP addresses associated with a network. They will be assigned to instances in an associated network. Routers : Routers provide gateways between various networks. Private IP addresses are visible within the instance and are usually a part of a private network dedicated to a tenant. This network allows the tenant's instances to communicate when isolated from the other tenants. Floating IP addresses are assigned to an instance so that they can connect to external networks and access the Internet. They are exposed as public IPs, but the guest's operating system has completely no idea that it was assigned an IP address. In Neutron's low-level orchestration of Layer 1 through Layer 3, components such as IP addressing, subnetting, and routing can also manage high-level services.

There are three main components of Neutron architecture that you ought to know in order to validate your decision later with regard to the use case for a component within the new releases of OpenStack:. Neutron-server : It accepts the API requests and routes them to the appropriate neutron-plugin for its action. Neutron plugins and agents : They perform the actual work such as the plugging in or unplugging of ports, creating networks and subnets, or IP addressing. Agents and plugins differ depending on the vendor technology of a particular cloud for the virtual and physical Cisco switches, NEC, OpenFlow, OpenSwitch, Linux bridging, and so on.

Queue : This routes messages between the neutron-server and various agents as well as the database to store the plugin state for a particular queue. Neutron is a system that manages networks and IP addresses. OpenStack networking ensures that the network will not be turned into a bottleneck or limiting factor in a cloud deployment and gives users real self-service, even over their network configurations. Another advantage of Neutron is its capability to provide a way for organizations to relieve stress within the network

of cloud environments and to make it easier to deliver NaaS in the cloud. It is designed to provide a plugin mechanism that will provide an option for the network operators to enable different technologies via the Neutron API. As a result of the API extensions, organizations have additional control over security and compliance policies, quality of service, monitoring, and troubleshooting, in addition to paving the way to deploying advanced network services such as firewalls, intrusion detection systems, or VPNs.

Keep in mind that Neutron allows users to manage and create networks or connect servers and nodes to various networks. The scalability advantage will be discussed in a later topic in the context of the Software Defined Network SDN technology, which is an attraction to many networks and administrators who seek a high-level network multitenancy. Horizon is the web dashboard that pools all the different pieces together from your OpenStack ecosystem. Horizon provides a web frontend for OpenStack services.

Currently, it includes all the OpenStack services as well as some incubated projects. It was designed as a stateless and data-less web application—it does nothing more than initiating actions in the OpenStack services via API calls and displaying information that OpenStack returns to the Horizon. It does not keep any data except the session information in its own data store. It is designed to be a reference implementation that can be customized and extended by operators for a particular cloud. It forms the basis for several public clouds—most notably the HP Public Cloud and at its heart, is its extensible modular approach to construction. Horizon is based on a series of modules called panels that define the interaction of each service.

Its modules can be enabled or disabled, depending on the service availability of the particular cloud. Most cloud provider distributions provide a company's specific theme for their dashboard implementation. Let's try to see how OpenStack works by chaining all the service cores covered in the previous sections in a series of steps:. A conversation is started with Keystone—"Hey, I would like to authenticate and here are my credentials". Keystone responds "OK, then you may authenticate and give the token" once the credentials have been accepted. You may remember that the service catalog comes with the token as a piece of code, which will allow you to access resources. Now you have it! The service catalog, during its turn, will incorporate the code by responding "Here are the resources available, so you can go through and get what you need from your accessible list".

The service catalog is a JSON structure that exposes the resources available upon a token request. Typically, once authenticated, you can talk to an API node. Once we authenticate and request access, we have the following services that will do the homework under the hood:. Network services that make all the connections between VLANs and virtual network interfaces that work and talk to each other. However, how do we get these services to talk?