Lectures On Cloud Computing with OpenStack and Amazon Web Service
by Prof. Dr. Harald Richter

Foreword

It is the hope that this script will be a good entry into the subsequent subject. However, it is not a book. It shall only free the audience from writing-up by hand, so that listeners can focus better on the matter. Each student has different strong and weak sides. Therefore, he/she will make personal remarks and explanations at different points in the script as needed. The document at hand will help him or her to do this with ease.

Neither the script nor the web can substitute the visit of the lectures. Many times, it has been evident that autodidactic learning of a extensive subject only by means of written documents, by the web or by videos requires much more time and personal engagement than visiting the lectures on a regular basis, especially if a personal contact during the lectures and the practical trainings is established.

Due to German Copyright Law, all rights are with the author. Copying this script in any way or publishing or forwarding it to third parties is not possible. Infringements will be prosecuted by law.

Part I - Introduction into Cloud Computing
1 What is Cloud Computing (CC)?

- Historically, CC evolved -among others roots- from so-called Utility Computing

1.1 Utility Computing First

*Def.: In Utility Computing, a service provider allows its customers to use IT resources in the provider's data centers via the Internet.*

*Example: IT resources are computing power, data storage, networking, operating systems and software applications in a computer cluster*

*Def.: An IT resource is called web resource if it is accessed via http or REST requests. Each web resource is identified by an Uniform Resource Identifier (URI) which is unique for every web resource.*

- Users are renting web resources on a pay-what-you-use basis, similar as gas or electricity

- Utility Computing and thus CC have the following features:
  1.) Renting of web resources is the common business model
  2.) Scalability of web resources is easily possible
      - all provider's services appear to be 'infinite' to the users
      - users can request dynamically via the Internet more or less resources
  3.) Smart metering of web resources is provided
      - consumed resources are fine-grained metered and billed according to their effective utilization
  4.) Automation of management of web resources is reality
      - System administrator tasks are highly automated and accomplished by simple user actions via the Internet

- These technologies are modern and effective means for the provisioning, management and usage of web resources

1.2 The New Features of Cloud Computing (CC)

- On top of Utility Computing, CC has put several new features, such as auto-scaling of web resources, for example
- Over time, CC evolved to many different services
- A comprehensive definition of CC stems from the U.S. National Institute of Standards, NIST:

  *Def.: „Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.“*
This definition is good but not clear enough for a CC novice
For a more intuitive definition, it is better to ask the following question:

• Does the CC user engage another company called Cloud Service Provider (CSP) to satisfy his IT needs?

or

• Is a so-called cloud operating system installed on user computers at his own premise?

The answers to that questions determine the functionalities of CC

Example: Amazon Web Service (AWS) is the most important commercial CSP, while OpenStack is the most important open-source cloud operating-system.

Please note that most of the Internet-accessible services are not CC

Example: No CC services are delivered by Facebook, Twitter, YouTube, Instagram, WhatsApp, Dropbox, BitBucket or GitHub.

Example: Apple's iCloud provides only a few simple CC services

However, many of the services of the providers listed above are implemented internally using CC

1.3 Cloud Service Provider (CSP)

A CSP is a company that has multiple computing or data centers around the world

Comment: We talk explicitly about a data center if the storage of data is the main purpose, such as it is the case in big data. We talk about a computing center if the processing of data is the main purpose, such as via supercomputers. If big data is processed in supercomputers both notions coincide.

A CSP provides IT services to paying customers via the Internet such as storage, databases, computing, data distribution, ...

Example: The big CSPs are Amazon, Microsoft, IBM and Google. Smaller CSPs are SAP and German Telecom.

A CSP's customer is typically a company, but private users exist as well

Amazon is the inventor of CC and the market and technology leader in CC, followed by Microsoft Azure Cloud, IBM Cloud, Google Cloud Platform and SAP HANA cloud e.g.

Example: AWS offers about 65 services from all fields of IT. OpenStack offers about 46 services from many fields of IT.

CSPs use internally proprietary or public cloud OSes for their computing/data centers to implement services for their customers

All of them have thin-provisioning as their business-model

1.3.1 Thin Provisioning

Typically, a big CSP has millions of customers

Thin provisioning means that a CSP has much more registered customers than he can satisfy simultaneously
This is possible because CSP services are world-wide accessible, and because there is always a whole continent where users are sleeping and thus not active in a CSP cloud.

Additionally, not every user needs 24h/7d CC services

\[
\Rightarrow \text{less computers can serve more customers by over-committing the CSPs physical hardware}
\]

Thin provisioning allows for high hardware utilization and is thus financially very attractive for a CSP.

Thin provisioning is useful for both, the CSP and the CC user, because the resulting price for CC is very low.

Example: Dozens of virtual machines (VMs) with Gigaflops of computing power and Terabytes of disk storage per VM, as well as large virtual LANs with many switches connecting these VMs are available in AWS auctions for 10 $ per month only.

Thin Provisioning is one of the reasons why CC is so widespread.

Thin provisioning is implemented in software by multi-tenancy software.

---

### 1.4 CC via a CSP (not via own cloud OS)

**Def.:** CC via a CSP means for a user to have Internet access to a so-called public or hybrid cloud at various levels of service. These levels are Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service.

#### 1.4.1 Public Cloud

**Def.:** A public cloud provides for users general IT resources via the Internet, such as computing capability, storage capacity, networking management, operating system support and sector-specific software-applications.

- Typically, a public cloud is geographically distributed over dozens or even hundreds of data/computing centers of the CSP.
- The exact location where a specific web resource is handled is unknown to the user and normally also not important.

#### 1.4.2 Private Cloud

For the definition of a hybrid cloud we need to define a private cloud first.

**Def.:** A private cloud is a locally concentrated computer cluster running a cloud OS. The cloud OS allows for numerous IT management functions initiated from a central admin console.

Comment: Admin = cloud administrator.
1.4.3 Hybrid Cloud

**Def.:** A hybrid cloud is the combination of a private with a public cloud. Sensitive data are kept on the user premise in a private cloud, while public cloud services are though accessible via the Internet.

- Hybrid clouds extend the user's on-premise features significantly by ‘infinite' CSP resources and their numerous services
- A drawback of hybrid clouds is their lower IT security compared to a private cloud

1.4.4 Infrastructure-as-a Service (IaaS)

- IaaS is the basic level a CSP can provide for a user as service
- With IaaS, the CSP provides PCs, servers, hard drives, switches, routers or even whole data/computing centers for a customer

**Example: Amazon’s EC2 in AWS delivers IaaS to customers**

- Typically, IaaS resources are virtual, i.e. the functionality of PCs or servers is emulated by software
  
  ⇒ Virtual IaaS resources do not really exist, they are just code on the CSP’s hardware
- The CSP bundles a set of virtual resources into a so-called virtual machine (VM)
- The user rents one or many VMs from a CSP

- On special request, also real resources can be obtained as pieces of CSP hardware in a CC center
- Physical resources are rented from a CSP and located in their CC centers
- Alternatively to virtual resources, also so-called Linux Containers (LXCs) can be requested
- While demand for real resources is very rare, LXCs are getting more and more attention in CC
- All resources, i.e. virtualized ones, LXCs, or physical ones are maintained and administered by the user only without the help of the CSP

1.4.4.1 Resource Administration by the User

- The user has to install a so-called guest operating systems on top of his virtual or physical machines or his LXCs, together with the software development frameworks he needs, such as Eclipse or Microsoft Visual Studio for example
- Additionally, he installs application software and cares for all updates and licenses by himself
- Furthermore, the user can increase or decrease the number of virtual and LXC resources easily and quickly via the internet
- After all installations, he has a set of resources for himself and his project
- Finally, he has full control over his virtual and LXC resources and can determine their type and features arbitrarily
Please note, both does not hold in case of physical resources

1.4.4.2 The Role of the CSP

- The CSP only meters and bills the actually consumed computing, storage and network resources according to „pay as you go“
- The CSP ensures also a high resource-availability of more than 99,99 %
- This implies automatic backups made by the CSP, distributed file systems, redundant PCs and servers, as well as redundant power supplies and disjoint geographical locations of computing centers

1.4.4.3 Nested Virtualization

- On top of a virtual infrastructure, a user can install an own cloud OS, thereby creating a cloud inside of a cloud
- This is called nested virtualization
- Nested virtualization is not possible with PaaS or SaaS

1.4.5 Platform-as-a-Service (PaaS)

- PaaS is a more abstract service of the CSP for a customer
- In PaaS, a ready-to-use IT infrastructure is available for customers, with all guest OSes and software development frameworks already installed
- The CSP cares also for all switch and router settings, for public and private IP and MAC addresses, for firewalls and software updates
- The user can focus on installing and running his own sector-specific applications, or he can focus on writing his own software

Example: Microsoft Azure provides PaaS for software developers not for arbitrary users

- Typically, PaaS has auto-scaling of IT resources
- With auto-scaling, the user’s IT resources depend on the number of active applications started by him or on the number of software developers that are currently active
- In PaaS, the user knows nothing about the underlying physical cloud and its software and also nothing about the effectively consumed resources
- At the end of the month, the user is billed on basis of special contracting such as „flat rate“ or actual resource usage

1.4.6 Software-as-a-Service (SaaS)

- SaaS is the most abstract, but also the most expensive service level
- In SaaS, a CSP provides for everything and maintains even the user’s sector-specific applications
- The user has to care for nothing, for example also not for his software licences

Example: Oracle Database Cloud or Google Docs are providing SaaS
Users at the PaaS or SaaS level use a CSP’s cloud without seeing anything from it
Sometimes, these users claim that they are doing CC which is wrong, because they are only using CC services from a CSP
Please note, although Google is a commercial CSP its “Docs” SaaS is free at first sight
However at 2nd sight, it is clear that Google has a glass fibre to the U.S. National Security Agency (NSA) and transmits your data in Google Docs to NSA, as all other U.S.A.-based CSPs do with your data
Only in case that it is assured by a contract that your data are stored and processed in Germany, German IT protection laws will apply

1.4.7 Special CSP Access-Offers

In addition to public or hybrid clouds exist Community Clouds, Virtual Private Clouds and Multi Clouds as special access offers from some CSPs

1.4.7.1 Community Cloud

A community cloud is a semi-public cloud run by a specialized CSP
The community cloud has restricted access, only members are entitled to register

Example: The cloud of the GWDG in Göttingen is a community cloud.

The advantage of a community cloud is the cost-sharing between its members compared to a private cloud for each member alone

1.4.7.2 Virtual Private Cloud

A virtual private cloud (VPC) is the attempt of CSPs to deliver more data security to their customers
The CSPs guarantees that a VPC can only be accessed via a VPN connection that is additionally secured by IPsec

Comment: A VPN connection establishes a virtual private network which is as a tunnel between the user PC and a VPN gateway.
Comment: IPsec allows for data encryption on ISO layer 3. It can be used to encrypt data in a VPN

Furthermore, the CSP adds firewalls around the customer's resources

Example: Amazon’s Virtual Private Cloud is the standard AWS EC2 service with an additional IPsec-based VPN-access, together with Linux security groups and access lists for the virtual machines of the customer.

1.4.7.3 Multi Cloud

A multi cloud allows to access services from multiple CSPs via a single interface
Its API creates a federation of clouds

Example: A multi cloud can bundle the services from AWS and from Google Cloud Platform in one API.

The advantages of a multi cloud are for a user:
• He is not fixed to one CSP
• He has a broader offer of services
• He has 100% availability of at least some services because no two CSPs will collapse at the same time

1.5 CC with own Cloud OS (not via CSP)

- If a cloud OS is installed on a propriety computer cluster, an own cloud is achieved
- The cloud-OS transforms an ordinary computer cluster into a fully-fledged cloud
- The service level an own cloud OS typically provides is IaaS
- The cloud OS can be either open source or commercial

Example: While OpenStack is the dominant open-source cloud OS, IBM Cloud has the same role for a commercial cloud OS. The private cloud versions of Microsoft Azure and the SAP cloud platform are less important.

- A private cloud is devoted to its owning organization only
- Typically, a private cloud is also on the customer's premise
- Users who are running a cloud OS on their premise can truly claim that they are doing CC
- However, hosting and management of a private cloud can also be out-sourced to an IT specialists, i.e. a company which is somewhere else

2 What are the Pros and Cons of CC?

- There are common advantages for both ways of CC (CSP and own cloud OS) which are:

  1.) The usage of CC resources and services is independant of the user's Internet device or physical location
     - all devices with browser are possible
     - all locations in the world with Internet access are allowed

     ⇒ the user has a much higher flexibility and agility than with standard server equipment

  2.) Total costs of ownership are lower

  3.) Administration of a cloud is easier than for a standard computer cluster

  4.) Isolation between VMs is perfect, even on the same server

     ⇒ Data security between multiple tenants in the same cloud is very high

Comment: A tenant is a group of users who share cloud resources and access privileges. Own chapters about tenants and multi-tenancy will follow.

- There are also a common disadvantages:

  1.) A cloud is a distributed system whose VMs must communicate even in the same project only via TCP/IP and although the VMs may be physically hosted by the same server or the same cabinet or the same computing center
Efficient data exchange with low latency between VMs is normally not possible

2.) Because of that, clouds are typically not used for High-Performance Computing

3.) If VMs are engaged instead of LXC containers then high software-overhead results

4.) If LXC containers are used instead of VMs then data security between multiple tenants is significantly lower

The other pros and cons of CC depend on the chosen model

2.1 CC via CSP

In case of CC via a CSP, a customer has numerous advantages compared to own hardware and software

2.1.1 Pros

A company can change its location without changing its IT and without interruption of its business

In AWS, for example, users have much more services than they could ever provide by themselves

Users have a much higher IT reliability and robustness than with own installations

Users have much less costs because:

• no expensive hardware or no own data/computing center has to be provisioned
• only simple PCs with Internet access are needed to profit from CSP services
• no own personnel is required for the company's IT in case of PaaS or SaaS
• prices of IT resources in public clouds are very moderate because of thin provisioning that results in high CSP hardware utilization
• no capital-intense investments for own IT, only smaller running costs for rented IT

deployment, operating and maintenance of company IT is much cheaper and much more simplified with CC

The CSP bills costs precisely and transparently for consumed resources

Users have much higher flexibility with respect to growing or shrinking IT needs because:

• resources can be requested by the user without CSP intervention = On-Demand Self-service
• resources can be obtained or returned within minutes or seconds = Rapid Elasticity
• resources are available on all layers of abstraction, from virtual IaaS hardware to sector-specific SaaS applications
• resources can be automatically adapted to the demand = Auto-Scaling
• resource demand can grow ‘unlimited’ without problems

CSP centers are much more energy-efficient than user clusters because so-called server consolidation via VM migration is possible
2.1.1.1 Server Consolidation via VM Migration

- VM migration means that a whole VM is moved to another computer
- VM migration encompasses the following steps:
  1.) VM's execution is stopped on a physical server
  2.) the whole VM including its data is stored in one file ("Snapshot")
  3.) the file is copied to another physical server, and the VM is activated again

- The VM does not notice its migration
- After having migrated all VMs from those servers that are under-utilized to servers that are better utilized, unused servers can be switched off, thus saving power
- Server consolidation works the better the more VMs can be moved
- CSPs can always consolidate servers because of many VMs, users in a private cloud cannot because of lack of VMs

2.1.2 Cons

- IT security is a critical issue if the underlying data/computing centers are not located on the territory of the Federal Republic of Germany
- Nearly no interoperability between services of different CSPs exists
- To port the own software and applications to another CSP is a challenge

- Therefore, a so-called vendor-lock-in may result in many use cases
- If a CSP gets bankrupt the customers will probably also go bankrupt if company-proprietary data cannot be accessed any more
- If a CSP cloud fails, the user must stop his own business as well until the CSP has restored services

2.1.2.1 Vendor-Lock-in

- If a customer cannot change its CSP any more because of the resulting high complexity or high costs then he is bound to his CSP forever which is called Vendor-Lock-in

2.2 CC with own Cloud OS

*Def.:* A cloud OS is a software to realize a private cloud or the private part of a hybrid cloud on the own premise by means of an own computer cluster

*Comment:* OpenStack is sometimes used by smaller CSPs to create public clouds.

2.2.1 Pros for own Cloud OS

- In case of CC with own cloud OS, a user has also many advantages, but they are different to the case with CSP
  - Benefits are:
    1.) Single Sign-On
• the same user name and password are valid on all computers of the user without extra effort

2.) Single Identity Management
• all user names, passwords and access rights are administered centrally from one console only

3.) Single Update
• the update of the guest OS in the VMs of cloud can be automated by updating only one “master“ VM and replicating it many times

4.) Single Network Management
• the cluster's network management including switches, routers, and MAC and IP adress allocation is easily accomplished via a given cloud OS tool from a central console

5.) Easy Integration
• the integration of new servers into the cloud is easy because of the central console and semi-automated cloud OS tools

6.) Easy Performance Analysis and Tuning
• the analysis and tuning of the performance of a cloud is eased because of given cloud OS tools executed from a central console

7.) Easy Accounting
• accounting and billing in the cloud is fully automated because of cloud OS services

8.) Single Data Storage and Access
• sharing of data in files within the same project is simple because of ready-to-use shared and distributed file systems with automated data replication

9.) Load Balancing
• load balancing between servers is possible and simple because of a cloud-wide meta-scheduler and because of easy manual VM migration between servers

10.) Easy Overall Administration
• The multitude of cloud OS services and the central console simplify the overall handling of any computer cluster without extra software required

2.2.2 Cons for own Cloud OS

CC with own cloud OS has also drawbacks:

1.) Difficult Installation
• Although diverse helper tools exist, the installation of a cloud OS such as OpenStack is a challenge and normally only possible for experts

2.) High Complexity
• Cloud OSes are mostly highly complex

Example: The OpenStack project has more than 80 Tsd. registered members and comprises > 20 M lines of code.
Management services can be highly complex as well

Example: OpenStack’s “Neutron” networking service has an API with about 244 REST-based request “calls”.

3.) Slow APIs

- Because of REST, high-speed data transfers between management services or between user apps and management services are not possible

4.) Inefficient Inter-VM coupling

- A cloud is a distributed system which implies inter-VM coupling via TCP/IP
- An efficient inter-VM data-exchange is not possible because of the high latency each TCP/IP protocol stack has
- TCP/IP would not be needed because the cloud is in the same cabinet or the same computing center

5.) Normally No High-Performance Computing

- Clouds are for general-purpose computing but not for high-performance computing because of the high latency in inter-VM data-exchange
- Only if a virtual shared memory between VMs is engaged, HPC is also possible

Example: lvshmém is the only existing virtual shared-memory between VMs on the same server. It is based on a virtual PCI device in each guest OS and a common Linux SHM in the host OS of a single server.

3 Technologies of CC

- All CC technologies were created to serve users and so-called tenants, and to maximize profit for the CSP

3.1 Common Technologies for CSP Clouds and own Cloud OS

- There are some common software technologies for both, CC via CSP and via own cloud OS which are:
  1.) Tenants and multi-tenancy for resource pooling
  2.) Linux containers (LXCs) or alternatively virtualization
  3.) Auto-scaling for resource allocation to users

3.2 Tenant and Multi-Tenancy

Tenants and multi-tenancy are key concepts of CC

3.2.1 Tenant

- The tenant concept is derived from so-called multi-user jobs in mainframes of the 1960s
Example: The code of a text editor existed only once in the main memory of a 1960 mainframe and was shared by many users at the same time exhibiting to each of them only his file

- From that, the definition of the term tenant is:

  **Def.:** A tenant is a group of users who share cloud resources and access privileges.

  **Example:** Users in a tenant are working in a joint project for which they are sharing the cloud’s disk storage for data exchange.

- In OpenStack, the notion of “tenant“ evolved to the simpler term “project“ which has become identical by definition of the OpenStack developers

### 3.2.2 Multi-Tenancy

**Def.:** Multi-tenancy is a feature of software that runs on a single server and serves multiple isolated tenants at the same time.

- Multi-tenant software has to provide shared resources and privileges within each tenant and isolated resources between tenants
- Multi-tenancy leads on the same server to a set of user groups with cooperation between users inside of the same group and full isolation between groups

Example: Each application in Google docs is a SaaS service that allows for multi-tenancy because it allows to share documents within the same collaboration group and to separate documents from different collaboration groups.

Example: Neutron is an OpenStack multi-tenancy IaaS-service that allows for a shared vLAN for each user in the same tenant. It provides separate vLANs for different tenants which are all implemented by the same physical LAN.

Example: Linux Control groups (Cgroups) provide for every user group shared resources and privileges and isolate different Cgroups from each other.

Example: Hypervisors such as KVM or Xen are Multi-tenancy software because they create and maintain isolated VMs while each VM can serve a user group by time sharing.

- Multi-tenancy implements thin provisioning
- Multi-tenancy is very efficient because it profits from resource pooling

### 3.2.3 Resource Pooling

**Def.:** Resources such as computing power, disk storage or networking are shared between the users of a tenant and between multiple tenants

- Resource pooling reduces the costs of a CSP because the price of a resource is divided by the number of users for it
- However, shared resources have the potential of a single-point-of-failure
3.3 Linux Containers (LXCs)

- LXCs are based on Cgroups and namespace isolation which are Linux kernel functions

3.3.1 Cgroups

- A Cgroup is a set of processes that are owned by a group of users
- All processes of a Cgroup can be frozen for later checkpointing and restart
- A Cgroup can be set by a Linux command as follows:
  1.) Not to exceed a prescribed limit with respect to main memory
  2.) To get a larger CPU share or more disk I/O bandwidth than other Cgroups
     • This implements Cgroup priorities
- A Cgroup's resource usage is measured by the Linux kernel for accounting and billing

3.3.2 Namespace Isolation

- In Linux, namespaces exist for:
  1.) process IDs (PIDs), which results in different PID namespaces
  2.) user IDs (UIDs) in order to establish different UID namespaces
  3.) disk mounts to obtain different virtual file systems
  4.) the own server name to allow for different hostnames for the same physical server
  5.) inter-process communications (IPCs) to get different sets of IPCs
  6.) networking to have different network controllers, IP tables and routing tables to create different networks
- Namespace isolation allows to separate process groups such that they cannot access resources from other groups but such that they can access resources easily from the same process group
- The access isolation is accomplished by the fact that only the own namespaces are visible to each process group

3.3.3 LXC Functionalities

- The LXC kernel functionality splits the set of all processes into groups that are isolated from each other by means of Cgroups and namespace isolation
- Each process group sees only its “own“ Linux kernel because other groups are invisible
- The “own“ Linux kernel is also named “Linux container“ or LXC
- On top of each LXC, standard Linux libraries and tools are executed which establish together with the common kernel a whole Linux operating system for every process group individually
Inside of each process group, the standard Linux access rights for resources can be set such that group members can cooperate easily

⇒ LXC makes Linux to a multi-tenancy operating-system

- The first advantage of LXC is the higher data security between process groups compared to that without LXC
- However, LXC containers are not as secure as VMs
- The 2nd advantage is its low overhead compared to VMs

Example: “Docker” is the most widespread Container-Software that is based on LXC.

3.4 Virtualization by means of VMs

- Virtualization means to have VMs and virtual networks connecting them to the Internet instead of LXC
- It means also to engage complex software technologies that incur additional overhead compared to LXC and that need explanations (see Chapter 4 "Virtual Machines (VMs)"
- Finally, virtualization means also to have complete isolation between VMs which is highly secure

3.5 Auto-scaling

- In AWS:
  - auto-scaling is possible with respect to the number of VMs in the Elastic Compute Cloud (EC2)
  - thus also the capacities for storage and networking can be auto-scaled, as in a few other AWS services
  - the throughput of the DynamoDB No-SQL Database can be auto-scaled as well according to DB’s input traffic
  - most of the other AWS services are not auto-scalable
- In MS Azure:
  - the number of active PaaS applications can be auto-scaled on basis of any metric
  - advanced auto-scaling according to time and date is also possible
- In OpenStack:
  - various auto-scaling policies are possible by means of the “Senlin” service
  - the AWS auto-scaling API will also be available in Senlin

Example: The OpenStack “Heat” service uses the OpenStack “Senlin” service for auto-scaling the number of VMs in a cloud.
3.6 Disjoint Software Technologies

Beside the software technologies which are common to both CC models, there are also other technologies where a distinction has to be made between CC via a CSP and CC via own cloud OS.

3.7 Specific Technologies for CC via CSPs

- AWS has:
  - language and platform-specific APIs, together with classical http(s) requests and SOAP
  
  Comment: The SOAP protocol allows for data exchange and remote procedure calls (RPCs) in distributed systems via the Internet.
  
  - only some services have also REST-based APIs, such as the AWS object store “S3“ or the content delivery service “Cloud Front“

- MS Azure has:
  - language and platform-specific APIs as AWS has and HTTP(s) and REST

- Both CSPs provide for on-demand self-service via their APIs

3.7.1 Language and Platform-specific APIs

- The AWS API Gateway allows to define any API for the compute cloud EC2, for the serverless “Lambda“ execution environment and for web applications

- AWS API calls are made by manual user requests, or automatically by content changes in S3 or in DynamoDB, or by state changes in the workflow software “Step Functions“

  Comment: “Step Functions“ allows to define arbitrary actions of AWS services in arbitrary sequences. This is called workflow.

- MS Azure has also an API Gateway as AWS does

- Additionally, Azure Active Directory AD allows to secure REST-requests by registering at the AD with credentials

  Comment: Credentials are username/password or SSH key.

3.8 Specific Technologies for CC via own Cloud OS

- CC via own cloud OS has typically only REST-based APIs

- Furthermore, the own cloud OS is visible to the owner of the private cluster in his software stack

- Its location is described subsequently

3.8.1 Location of own Cloud OS in the Software Stack

- The cloud OS is logically located above the “host OS“ of the physical servers and the hypervisor, but below the user apps in the VMs or Cgroups

- The resulting block diagram is shown below
The cloud OS is always executed in the host OS of all physical servers and never in a "guest OS" of a VM.

In contrast to classical OSes, a cloud OS is only a collection of management services that fully rely on classical hosts OSes and hypervisors or Linux container services.

Typically, the cloud OS configures only this classical software, but is not active by itself during run time.

Each management service is accessible via a REST-based API.

Comment: See the subsequent chapter about REST protocol for an explanation what REST is.

Example: OpenStack has about 24 different APIs for its roughly 32 management services. Many of them use JSON data structures inside of REST requests and responses.

Among each other, the management services are either not coupled or only loosely coupled.

3.8.2 Basic Services in own Cloud OS

A few basic services have to be installed on all servers of a own cloud.

Example: OpenStack requires some components of the "Nova" and "Neutron" services on every node.

Def.: In Openstack, a node is a physical computer.

Most of the other management services are optional.

Additionally, some nodes have a special role in a cloud OS.

Example: OpenStack has one or more controller nodes, one or more network nodes and many compute nodes.

In contrast to classical OSes, a cloud OS allows to administer only virtual resources.

This requires frequent interventions from the hypervisor on each server.

Furthermore, the management services are coupled to a central admin console.

This console allows to monitor and control the whole cloud which is very convenient.
4 Virtual Machines (VMs)

- The most important software technology for private and public clouds is virtualization.
- Virtualization has three aspects that have to be fulfilled all together:
  1.) Virtualization of main memory in physical servers and PCs by means of VMs
  2.) Virtualization of CPUs
  3.) Virtualization of peripheral devices, such as network interfaces and cables, switches and routers for example
- These aspects are implemented by a so-called hypervisor host-OS kernel-module
- In some cases the hypervisor has a user space emulator as companion that technically mimics a PC or server

Comment: The host OS is the operating system of the real server or PC. The guest OS is the operating system in a VM.

- If existent, emulator and hypervisor cooperate closely

Example: KVM is the hypervisor for a Linux host OS, and QEMU is its emulator that creates a virtual PC. Hyper-V is the hypervisor for a Windows Server host OS. It has the emulator included.

4.1 What is a VM?

Def.: A VM is a collection of virtual IT resources created by software in a real computer. Each VM is created, maintained, optionally moved and terminated by a hypervisor (virtual machine monitor), in some cases with the help of an emulator.

- In the context of CC, a VM is also called “instance“
- On every VM, a guest OS and user applications are running as on real hardware
- A single real PC or server can accommodate multiple VMs/instances simultaneously
Example: 3 VMs are emulated by one PC as shown in subsequent block diagram.

- In the example above, there is no over-commitment of resources because the number of virtual cores and the added sizes of virtual main memory and hard drives is not higher than that of the real hardware.

Comment: To over-commit = to over-book

⇒ thin provisioning does not exist in this example

Example: A virtual multicore CPU can be created inside of a VM by adding a so-called vCPU to a VM.

Def.: A vCPU is the virtual counterpart of a real core.

Example: In case of an Intel i7, i5 or i3 CPU and a so-called hyperthreading factor of 2, each real core can emulate 2 vCPUs without being over-committed.

- Although being on top of the same host OS, each guest OS can be different, as well as each VM setup.
- Each VM has its own guest OS => there are as many guest OS as VMs exist.

4.2 Advantages of Virtualization

- The advantages of virtualization lie mostly at the side of the CSP’s computing centers, but the user has also benefits.
- A CSP and its system administrators have the following advantages:

1.) There is no visible change for CSP customers before and after virtualization of a computer center’s hardware

   • Normally, the user can find out only by run time measurements that his applications have become a little bit slower.
2.) User software licenses are valid in the VMs as they are on the real hardware

3.) A reduction of the electricity bill results
   • the reason for that is the server consolidation:
     • low loaded servers get rid of their VMs by VM migration and can enter sleep mode
     • permanent server consolidation is important for CSPs because their costs are mainly due to electricity and not due to hardware or personnel
     • energy saving alone already justifies virtualization for a computing center

4.) System stability is higher
   • If a VM crashes this does not affect other VMs
   • If a VM tries to intrude an other VM this will be not possible
   • The reason is the perfect isolation of VMs between each other, even on the same computer

5.) Backups are simplified
   • a whole VM including all user files can be backed up in one “image” file only:
     • the VM image is a complete snapshot of the VM and all of its content

6.) Restarts are simplified
   • periodic snapshots can be made easily and help as follows:
     • in case of an error, a VM snapshot which was taken before the error can be run which is called Checkpoint and Restart
     • restarting a whole VM is as easy as starting an .exe file
     • it is not needed to restart the physical hardware, instead a VM launch is sufficient

7.) Migration, server consolidation and maintenance are easily accomplished:
   • from every VM, a snapshot can be taken by one command from the operator console
   • this VM image is copied to a 2nd physical machine and run again
   • after the 1st physical machine is free of VMs, it can be easily maintained or shut down

8.) Existing VMs can be made visible also to a cloud OS
Comment: A cloud OS is the software that makes a cloud out of a computer cluster.
   • the cloud OS can create, administer, maintain and terminate virtual resources by means of its powerful services which are implemented in part by a hypervisor and its emulator, for example
   • all advantages of virtualization and of cloud services are available to the admin and the user as soon as the hardware is virtualized

9.) Cyber security is improved
   • attacks are much more difficult because login into a VM from outside is accomplished via an SSH key such as RSA for example
     • a RSA key is safer than user name/password
   • isolation between VMs and between different virtual network equipment is good
   • if combined with a cloud OS, VMs and and virtual network equipment can be completely isolated from the Internet by not donating a public IP address to the VM => the hacker does not see the VM
10.) VMs can be created freely according to user wishes
   • every VM setup and configuration wish is possible because nothing really exists
     • every user request w.r.t. to a specific VM can be fulfilled, even a virtual shared memory between VMs is possible
Comment: W.r.t. = with respect to

4.2.1 Setup of a VM

- Typically, a VM consists of the following parts:
  1.) one or more virtual sockets for virtual CPU chips
  2.) one or more virtual-cores per virtual CPU chip which are called vCPUs
Comment: Also real server have several sockets per server and several cores per socket. Real PCs have only one socket for one CPU.
    • a vCPU is also called “logical processing unit”
  3.) virtual main memory including a virtual memory management unit (MMU)
  4.) virtual periphery including hard drives and network interface cards, so-called vNICs
- These virtual components have indirect access to a real window on a real screen, to a real keyboard and a real mouse, to a real network interface card and to a real hard drive
Comment: Indirect access means by means of a hypervisor and its companion emulator if existent.

4.3 Disadvantages of Virtualization

- Virtualization, however, has some disadvantages as well:
  1.) Enormous software overhead for its realization
      ⇒ very high structural complexity results and thus increased execution times
        • Not that many PCs or servers can be emulated by a single server
        • in order to reduce performance degradation, additional hardware support exist on the motherboard, the periphery and in the cores which are called hardware accelerators for virtualization
  2.) Accelerators for virtualization are nowadays indispensable but increase price and complexity of the hardware
- For these reasons, virtualization is under market pressure by LXC and successors, such as “LXD“ from Canonical Ltd.
  • This happens although they are not as safe as VMs, but much easier to realize

4.4 Host OS and Guest OS

- On every real server or PC, only one host OS exists
- Typically, the host OS is a standard OS such as Linux or Windows
- The host OS is indispensable as lower layer for the cloud OS because it contains the hypervisor in its kernel or is even host OS and hypervisor together

*Example: Linux is the host OS for OpenStack. It has KVM as hypervisor kernel module and QEMU in user space for VM emulation. XEN is another host OS that is its own hypervisor at the same time. Windows is a third host OS that has Hyper-V as hypervisor.*

- On any emulated PC or server (=VM or instance), multiple user applications are runnable under the same guest OS as on the real hardware with the host OS

Comment: There is a difference between simulation and emulation. Simulation happens solely by software, while emulation is with hardware assistance or fully in hardware. As a consequence, Simulation is never real-time capable, while emulation can be.

- Technically, each VM is a process in host OS which means it has its own address space and is isolated from other processes by the memory management unit of the CPU core that executes the host OS

- Together with the emulator (if existent), the hypervisor ensures that the VM knows nothing about the host OS, nothing about to be a host OS process only and nothing about the real hardware

- Typically, communication between VMs on the same server is possible only via the Internet, i.e. via TCP/IP, provided the VM has a vNIC with indirect Internet access

Comment: Internet access is possible for a VM if its vNIC has a valid MAC address, a public IP address and if its QEMU can access a real NIC that is connected to an Internet gateway.

### 4.5 Hypervisor

- The hypervisor software is located always below the VMs but above the hardware

- An emulator contains a single VM in its address space (if existent)

- One has to differentiate between type 1 and type 2 hypervisors

*Example: XEN and Hyper-V are type 1 hypervisors, while KVM is a type 2 hypervisor.*

- Both hypervisor types allow to create, to configure, to start, to stop and to terminate VMs on a single physical machine
Def.: A type 1 hypervisor does not need a host OS, instead it is the host OS. It is located
directly on top of a physical machine and has all device drivers needed for its hardware.
It does not have an emulator companion.

- If a type 1 hypervisor is present, then the device drivers of the guest OSes are stubs
  only which is called paravirtualization of the guest OS

Comment: An own chapter about paravirtualization will follow.

- In that case, the guest-OS device-drivers are modified such that they call the device-
drivers of the type 1 hypervisor instead of doing own IO

- A type 1 hypervisor typically implies the paravirtualization of the guest OS

**Example:** XEN is a type 1 hypervisor and micro OS kernel. It provides for its guest OSes
important functionalities, such as IO via device drivers, process scheduling via time sha-
ring, memory management via own pages tables, and some other important kernel func-
tions.

- In contrast to a type 1 hypervisor stands the type 2 hypervisor

Def.: Type 2 hypervisor: it is not a micro kernel, instead it needs a host OS and its device
drivers. It is executed inside of the host OS kernel. It may need also a software emulator
as companion to mimic a computer on which a guest OS is executed.

**Example:** KVM is usually a type 2 hypervisor. KVM is used by Linux and is part of it. XEN
as type 1 hypervisor is used by Amazon AWS.

- Both hypervisor types provide so-called server Virtualization

Comment: Server virtualization will be explained later.

- The device drivers of the guest OSes remain unmodified for a type 2 hypervisor
- Additionally, all guest-OS device-driver calls are intercepted by the hypervisor as soon
  as the guest OS tries to perform them
- Type 1 hypervisors are executing guest-OS IO-calls as proxies via their own drivers
- Type 2 hypervisors are doing guest-OS IO-calls as proxies via the host OS
- Both types need and have system privileges to execute the complete instruction set of
  the underlying CPU core
- Both types can thus access all files and devices, including main memory
- To summarize:
  - type 1 hypervisors do not need an emulator or a host OS and enforce typically paravirtua-
    lization of the guest OSes
  - type 2 hypervisors need a host OS (and sometimes also a emulator), but do not use para-
    virtualization
  - both hypervisor types are responsible for their VMs, but not for what happens in and bet-
    ween physical machines in a cluster
  - this is the task of the various host OSes and a cloud OS (if present)
4.5.1 ESXi, Hyper-V and KVM/QEMU

ESXi is the latest type 1 hypervisor of VMware Inc.

Comment: VMware is the technology pioneer and market leader in commercial virtualization solutions and therefore important.

- ESXi is used in their “vSphere“ product for server virtualization
- It mimics a virtual server not a PC

Comment: The difference between server and PC is the higher performance. A server typically has multiple CPUs with a shared memory between.

Hyper-V is the standard hypervisor for MS Windows

- It is of type 1 because it sits directly on top of the hardware
- However, it is not a microkernel and does not imply paravirtualization
- Instead, it follows a more complex virtualization concept by creating a so-called parent partition
- In the parent partition, Windows Server is executed as host OS of a physical server
  - Windows Server is a fully-fletches host OS and not a microkernel
  - this is the same as with type 2 hypervisors, although being type 1
- The parent partition spawns multiple child partitions, each with an own guest OS
- Consequently, one VM is located in each child partition
- Guest-OS device-driver calls are forward via a so-called VMBus to the host OS and executed there

- this is the same as as with type 2 hypervisors
- The VMBus connects host OS with guest OS
- Its endpoint in host OS is coupled to a “Virtualization Service Provider“, i.e. to Windows Server
- Its endpoint in guest OS is coupled to a “Virtualization Service Consumer“, i.e. to a VM

KVM is the standard type 2 hypervisor for Linux and has QEMU as emulator companion

- KVM/QEMU mimic together an old PC that has an IDE bus as hard drive interface and an PCI bus for its other periphery

Comment: IDE is the prerunner of parallel ATA, which is in turn the prerunner of SATA. PCI is the prerunner of PCI-X, which is the prerunner of PCIe. Because of these old technologies the emulated PC is old.

- KVM is part of the Linux kernel, while QEMU is running in user space
  - QEMU is also included in any Linux distribution as indispensible user module
  - A separate QEMU instance is needed for every VM because the VM is incorporated into QEMU's address space in the host OS
  - Each QEMU, i.e. VM is a host OS process
  - However, there existe only 1 KVM per physical server for all QEMUs/VMs

4.6 Paravirtualization

For paravirtualization, all original guest-OS device-drivers must be replaced by stubs
Example: KVM/virtio provide alternative device drivers for Linux as guest OS. XEN provides the same for Linux and Windows.

- These device drivers have the same API as the original drivers but they do not drive anything since they run in guest OS only
- In contrast to the original device drivers, they know that they run in a guest OS and cooperate therefore actively with that QEMU that is responsible for the guest OS
- They are stubs only that hand-over their call parameters to their QEMU

⇒ intercepting interrupts or privileged instructions is not needed since they do not want to control real hardware

- This saves time which is why paravirtualization is an efficient software solution

Example: If a TCP/IP packet has to be sent from a VM, its data is forward via the chain: Guest-OS application -> Guest-OS device-driver stub -> QEMU -> KVM -> host OS device driver -> NIC. This means: QEMU hands over the call parameters to KVM, and KVM calls the proper host OS device driver and returns their results via QEMU to the driver stub.

- Hypervisors have standardized APIs for paravirtualization

Example: XEN provides a “VMI API“ for the driver stubs in guest OS. KVM provides its virtio API for the same.

- With the advent of more and better hardware accelerators for server virtualization paravirtualization has lost importance

4.6.1 VirtIo for Paravirtualization

Def.: Virtio is a Linux kernel module for the Virtualization of peripheral devices (so-called “IO virtualization)“. Virtio contains own device drivers that are only stubs. It is used for paravirtualization.

- There exist Virtio stub drivers for block devices such as hard drives, for network devices such as NICs, and for other PCI devices
- Virtio can cooperate with KVM on the kernel level
- KVM in turn has two APIs, one for all QEMUs in user space and one for virtio in system space
- Surprisingly, as soon as virtio is called by KVM, KVM mutates into a type 1 hypervisor
- As a consequence, KVM can also realize paravirtualization by means of virtio
- The difference between KVM+virtio and XEN is that the Linux kernel is not a micro kernel as XEN but a fully-fletched OS
- The advantage of KVM+virtio+QEMU compared to the KVM+QEMU is its speed which is much higher because of paravirtualization

5 Server Virtualization

- A PC is capable to accommodate about 1-5 VMs because of lack of resources
- Servers instead can host about 10-20 VMs which is called “server virtualization“
A server that was virtualized typically does not provide physical resources to users but only virtual ones.

If VMs are replaced by LXC then about 100-200 containers can be hosted by one server because of much less overhead compared to VMs.

### 5.1 Comparison between Classical Computer and Virtualized Server

Without server virtualization, a classical computer results:

- In a classical computer, a single host OS supports \( n \) user applications, and host-OS device-drivers control real hardware.

Virtualized servers are more complex:

- In a virtualized server, for example by means of KVM/QEMU, more software units exist.

As a summary, KVM/QEMU interact with each other as follows:

1. KVM is executed as a loadable kernel-module in system space.

2. Multiple QEMUs are executed in user space.

3. Each QEMU is a host OS process in which a complete PC is emulated as VM.

4. In each VM, one guest OS is running, together with user applications in time-sharing.

5. In the standard configuration, each QEMU emulates an old PC with PCI and IDE buses:
   - in this PC, QEMU can be configured to emulate a multi core CPU, RAM and periphery.
   - it is also possible that QEMU emulates multiple CPUs with shared memory between them.
   - this allows not only to emulate an PC but also an server.
• for technical reasons, it is not possible to emulate hard drive capacities that are in sum bigger than the real drive, because each VM is just a executable file
• if too many virtual resources have to be emulated, the server becomes over-booked, and its performance drastically drops because of paging in and out hard drive records
• this paging will not stop any more in case of over-booking
6.) KVM communicates with all QEMUs via its QEMU API
7.) However, each VM is isolated from other VMs as any process is separated in the host OS from other processes
8.) Typically, VMs can communicate only via TCP/IP with each other, although being on the same physical machine
9.) Additionally, KVM intercepts all attempts of guest OSes to access real hardware
10.) For example, KVM blocks attempted IO accesses in guest OSes and executes them by itself by means of a proper host-OS device-driver call
11.) KVM returns the result of the host-OS device-driver to that QEMU whose VM has initiated the IO access attempt by its guest-OS device-driver
12.) The host-OS device-driver result is forward by QEMU to the guest-OS device-driver
13.) The same holds also for guest-OS access-attempts to real main memory

⇒ Each guest OS and its applications have the impression that they are accessing real hardware

5.2 How Host OS sees Guest OSes, QEMUs and User Applications

- For Linux as host OS, the situation is as follows:
  • All user applications of a guest OS are host OS threads of their guest OS process
  • Each guest OS is a host OS thread of its QEMU process
- In Linux are threads scheduled as processes and similarly treated which means:
  1.) A user application has a host-OS process-ID and the parent process ID of the guest OS
  2.) The guest OS has also a host-OS process-ID and the parent process ID of its QEMU
  3.) Each QEMU has a host-OS process-ID too

5.3 Privileged CPU instructions in Guest OS

- KVM intercepts privileged CPU instructions of the guest OSes as follows:
  1.) KVM intercepts all attempts of a guest-OS device-driver to execute a privileged CPU instruction which is needed to perform IO or to access physical memory
2.) Additionally, KVM intercepts all privileged CPU instructions of a guest OS that deal with the real memory management unit (MMU) and with page tables.

3.) Finally, KVM intercepts all privileged CPU instructions of a guest OS that have to do with system traps, timers and interrupts.

4.) Instead KVM executes all privileged instructions by itself and by the help of the host-OS device-drivers and forwards the result to the respective QEMU.

⇒ KVM is using corresponding host OS calls to replace the guest OS calls that are blocked by it.

5.) QEMU forwards the result to its VM and thus to the guest OS or application which has made the attempt.

⇒ The effect of the host OS call is the same as the guest OS call would have had. No guest OS understands that it is running only inside of a QEMU host-OS process.

- In case of an IO attempt by means of a guest-OS device-driver call, the notion of “IO virtualization“ is used.
- IO virtualization is part of server virtualization.

5.4 Server Virtualization in a Cloud

- In case that a cloud OS such as OpenStack is present, the situation in the computer changes again.

- In this case, a software called libvirt is the mediator between OpenStack and the virtualized server:
  - Libvirt is API and library for virtualization that helps OpenStack in the management of VMs.
  - With libvirt, VMs can be created, started, stopped and changed more easily than with KVM/QEMU alone.
  - libvirt in turn uses KVM/QEMU as underlying actuator, among other hypervisors.
  - libvirt has a comfortable CLI named “virsh“ and a GUI named “virt-Manager“.

Comment: CLI = Command Line Interpreter; GUI = graphical user interface.
The Horizon Dashboard, which is the central operator console, is using the libvirt CLI to create a GUI that is even more abstract and comfortable than that of virt-Manager.

The central dashboard provides an identical command “look and feel” for all OpenStack services.

5.5 Virtualization of Memory, CPU and IO

Server Virtualization has the three tasks of memory virtualization, CPU virtualization and IO virtualization.

All three tasks are needed to mimic multiple PCs or servers on a single physical computer.

Def.: Server virtualization is the creation of multiple VMs on a single physical machine by software, such that the VMs share the machine's resources without noticing that.

The sharing of a physical resources between multiple VMs is a challenge because no hardware component was originally designed to be shared between multiple VMs.

In the worst case, the host OS, all guest OSes and the user applications want to access the same resource simultaneously.

A safe serialization of access requests for the resource must be established. Side-effects such as deadlocks are not tolerable.

As consequence, a purely software-based server virtualization is difficult and slow.

This fact is explained subsequently by showing the inherent complexity memory virtualization, CPU virtualization and IO virtualization have.
5.5.1 Memory Virtualization

**Def.:** Memory virtualization is the creation of a classical virtual main memory for every VM, although there is only one real main memory that is already virtual for the host OS.

- The virtual main memory of a VM is based on double virtualization, i.e. on a double illusion because:
  - Virtual guest OS memory addresses have to be translated into virtual host OS addresses which are translated a 2nd time into physical main memory addresses
  - “Physical guest OS addresses" of virtual device registers have to be translated into physical host OS addresses of real device registers
- Each main memory access of a guest OS and its user applications has to be intercepted by the hypervisor and executed by that QEMU that is responsible for the VM in its user address space
- Additionally, all accesses to physical device addresses have to intercepted by KVM and handled by QEMU
- Furthermore, all page fault exceptions, DMAs and interrupts in a guest OS have to be intercepted and handled by the hypervisor as well
  - this is needed because the guest-OS page-tables, its memory addresses and interrupt vectors are all wrong
  - they do not reflect the situation in the real hardware and in the host OS
  - they may even collide with the page-tables, memory addresses and interrupt vectors of other VMs

- Finally, if the user wants to have a virtual shared memory between virtual CPUs in the same physical server this has to be emulated by KVM/QEMU as well
- To handle all these tasks in software is difficult

5.5.1.1 Virtual Shared Memory

- The only possibility for a virtual shared memory between VMs in the same server is ivshmem which is a QEMU feature

5.5.2 CPU Virtualization

**Def.:** CPU virtualization is the emulation of multiple CPUs, cores and hyperthreads for a PC or server by a small set of real processors or cores.

- CPU virtualization is difficult because each QEMU and its VM have only user space privileges while cores have an instruction set that requires sometimes system privileges
- As a consequence, VMs cannot execute all CPU instructions
- This implies the described complex cooperation between KVM and each QEMU in case of access privilege violations
- CPU virtualization implies also to create a virtual Translation Look-aside Buffer (TLB) inside of a virtual MMU
- However, the virtual TLB requires frequent updates which is why there is a hardware assistants in every real core to accomplish this, the IOMMU
With QEMU, it is even possible that one CPU of type x emulates another CPU of type y from a different vendor

This requires complex software and is very time-consuming

Comment: In addition to that, there is some confusion in CPU virtualization w.r.t. terminology: OpenStack calls the emulated cores in a CPU “vCPUs”, although they are only cores. Furthermore, a VM is termed by OpenStack an “instance” but not a VM.

5.5.3 IO Virtualization

Def.: IO virtualization is the mapping of virtual IO resources, such as network interfaces or hard drives of the VMs onto really existing periphery.

The mapping is based on time sharing the real periphery between VMs without interferences

This is a challenge because the periphery was never built to be time-shared between host OS processes, i.e. VMs

Each host OS can only serialize IO accesses to real periphery but does not time-share them between its processes

Example: Time-sharing of real periphery between host OS processes requires multiplexing of data frames in the Ethernet interface during ongoing transmissions.

It requires also multiplexing of records on the hard disk during ongoing read/write operations

Multiplexing finally requires input and output queues and a queue scheduler for frames and records which the host OS does not have

The software that has to handle these issues alone is necessarily complex

5.5.3.1 Example of IO Virtualization by Sending an Email from a VM

If a QEMU VM wants to send an email, everything is as on real hardware except the last step where the guest-OS device-driver wants to access the vNIC of a VM by means of a privileged instruction

Then KVM intercepts this instruction and calls the host-OS Ethernet device-driver

Afterwards the host-OS Ethernet device-driver sends the email through the real Ethernet interface

Finally, KVM generates a “done” interrupt for the guest-OS device-driver of the vNIC, so that it gets the impression that it would control real hardware

5.6 Hardware Accelerators for Server Virtualization

The goal of hardware accelerators for server virtualization is to obtain a close entanglement between hardware and software so that hypervisors are unburdened

This speeds-up the execution of VMs significantly and simplifies the software as well

Because of such accelerators, there is nowadays nearly no difference between an application running in guest OS or in host OS
Example: KVM/QEMU, XEN, ESXi and Hyper-V support hardware accelerators by cooperating with them in the cores, the motherboards, the BIOS and the periphery.

- Because of these benefits, more and more accelerators for virtualization are coming into existence
- Meanwhile they have fully superseded purely software-based server virtualization
- Without these accelerators, cloud computing would not be feasible because of speed, i.e. they are important
- In order to use them, the BIOS must support them and the BIOS configuration flags must be set properly

5.6.1 Accelerators for Memory Virtualization

- In each physical core of a multi core CPU is a 2nd memory management unit (MMU) called IOMMU which is responsible for real-time mapping of “physical“ guest-OS IO-addresses into physical host-OS IO-addresses that really exist

Example: The Intel “Extended Page Tables“ EPT and the AMD “Rapid Virtualization Indexing“ RVI are other commercial hardware accelerators for memory virtualization.

5.6.2 Accelerators for CPU Virtualization

Example: The hardware accelerators for CPU Virtualization from Intel are the “Virtual Machine Extension“ (VMX) and the “Virtual Machine Control Structure“ (VMCS). They are subsumed under “Intel Virtualization Technology“ VT-x. AMD has similar solutions.

5.6.3 Accelerators for IO Virtualization

- There are three main methods for hardware accelerators for IO virtualization:
  1.) In PCIe devices is an “Address Translation Service“ (ATS) for the real-time mapping of guest-OS PCIe-device register-addresses that are virtual into host-OS PCIe-device register-addresses that exist
  2.) In PCIe devices are multiple input and output queues in hardware for multiplexing VM accesses
      • With multiple input and multiple output queues, a hardware-based time-sharing of a single PCIe device between multiple VMs is possible because every data item can be obtained and delivered from and to the proper VM
  3.) With the two main methods mentioned and the IOMMU, guest-OS device-drivers can even be entitled to directly control host OS devices
      • This is called Single Root I/O Virtualization (SR-IOV)
- All three main methods together unburden the hypervisor significantly

5.6.3.1 SR-IOV

- With SR-IOV, commands from a guest-OS device-driver are forward via QEMU and KVM to the real device without participating the responsible host-OS device-driver
Furthermore, SR-IOV multiplexes device access requests from VMs including their data such that each VM gets its data automatically by hardware.

SR-IOV thus enables guest-OS device-drivers from multiple VMs to control the same physical PCIe device in a time-shared round-robin manner.

⇒ SR-IOV eliminates the calling of the responsible host OS device drivers via QEMU/KVM which reduces the communication and overhead.

SR-IOV must be supported by the BIOS, the motherboard and the CPU in order to work.

SR-IOV is included in newer PCIe devices, but not in the low-price category.

Other important commercial hardware accelerators for IO virtualization from Intel are:

1.) I/O Acceleration Technology (IOAT)
2.) Virtual Machine Device Queues (VMDq)
3.) Virtualization Technology for Directed I/O VT-d

They are known as „Intel Virtualization Technology for Connectivity, VT-c“

AMD has similar technologies.

5.6.4 Status of Hardware Accelerators for Server Virtualization

The user of a VM and thus the user of a cloud typically does not know or does not understand the technical difficulties incurred by virtualization.

Instead, he expects from a VM in a cloud the same speed, functionality, main memory size, hard drive size, memory protection and cyber security as in a real PC.

The status of hardware accelerators for server virtualization is that a VM can execute a user application meanwhile nearly as fast as the physical computer could do, but for 10-20 VMs at the same time.

Hardware accelerators do not only speed-up server virtualization, they also simplify the software needed for emulation significantly.

5.6.5 Summary of Hardware Accelerators

IOMMU, ATS, hardware queues, SR-IOV and a better BIOS ensure together that each VM gets only its data, although real periphery is time-shared between VMs.

The I/O Acceleration Technology (IOAT), Virtual Machine Device Queues (VMDq), the Virtualization Technology for Directed I/O (VT-d), the Virtual Machine Extension (VMX), the Virtual Machine Control Structure (VMCS) and the Extended Page Tables (EPT) additionally simplify the job of the hypervisor and the emulator (if existing) drastically.

However, even with those accelerators cloud computing is still not high performance, w.r.t. to inter-VM communication in the same server or between servers.

Comment: It is possible that a hypervisor and LXCs can coexist in the same server and thus in the same cloud.
5.7 Inter-vCPU and Inter-VM Communication on the Same Server

- The basic concept of every cloud is to isolate VMs ("instances") from each other.
- This is good for stability and data security.
- VMs can typically communicate only as a distributed system, i.e. via TCP/IP, although being in the same server, the same rack or the same computing center.
- TCP/IP was not made and is not intended as effective mean for this use case.
- A problem arises if the VMs of different users that are working in the same project want to exchange data quickly.
- This is normally not possible in a cloud.
- Fortunately, communication between the vCPUs ("cores") of a virtual CPU is easy because each vCPU is implemented, at least in QEMU, as a child process of the CPU father process.
- All child processes and the father are sharing the same memory address space which allows for simple and fast data exchange by shared variables.
- Inter-VM data exchange instead is slow because of TCP/IP and other software overhead.
- This is bad for many cloud applications, as well as for High-performance Computing (HPC) which is thus normally not possible in a cloud.
- In Amazon's AWS and in VMWare's vCloud, there is no option to improve this situation.
- In OpenStack, however, there is a way-out by using virtual shared memory implemented by "Inter-VM Shared Memory (ivshmem)"

- Ivshmem is situated in a virtual PCI device and mapped onto physical shared memory in host OS.
- This is technically exceptionally complex, but about 10 times faster than TCP/IP which makes ivshmem important.
- It is part of QEMU and thus of Linux, however it is scarcely used ("unknown territory")

5.7.1 Inter-VM Communication Without Ivshmem

- In principle, there exist two software instances in a cloud that would be capable to perform the inter-VM communication inside the same server if ivshmem is not present.
- These instances are the QEMUs on the server, and OpenStack's "Neutron" network service in the cloud.
- In fact, the QEMUs are responsible for inter-VM communication inside the same server and not Neutron.
- Without ivshmem, there are two sources of overhead: TCP/IP and KVM/QEMU.
- The latter create overhead because of the following:
  - QEMU uses a software called "macvtap" in "Bridge Mode" for inter-VM communication in the same server.
  - Macvtap is a crossing of the Linux "macvlan driver" and a "tap device".
  - Macvtap replaces the vNIC-device driver-API of Macvlan by a classical Linux tap API.
5.7.1.1 Tap Device

- A tap is a user-space interface and delivers a virtual endpoint to which a NIC or vNIC can be connected.
- A tap device mimics the API of a physical or virtual Ethernet card (NIC or vNIC) but does nothing inside.
- Instead of passing frames to and from a NIC or vNIC, frames are only read from and written to a tap driver, which is an API only.

**Example:** The Linux bridge that is incorporated into Neutron is using a Linux tap.

- The Linux kernel makes the tap device available to user space via the `/dev/tap<N>` device file, where `<N>` is the index of the software-only tap interface.

5.7.1.2 Macvlan Driver

- The macvlan device driver connects virtual network interfaces (vNICs) to a single physical network interface (NIC) by multiplexing their data streams onto the NIC.
- Each vNIC has its own MAC address which is distinct from other vNIC’s MAC address.
- Frames sent to or from the vNICs are mapped to the same NIC, which is called the “lower interface“ or the “underlying interface“.

5.7.1.3 Macvtap

- Macvtap is a vNIC device driver with a tap interface to a user-space code.

- The output of macvtap can be connected to a second tap as it is the case with any vNIC.
- A user space code, such as QEMU for example, can open the `/dev/tap<N>` device file and send and receive Ethernet frames via it.
- Originally, it has been the system-space TCP/IP protocol stack that has sent Ethernet frames over a NIC, but with macvtap it is a user-space code.
- When the kernel executes macvtap, instead of sending frame to a NIC, it makes macvtap only available to the user space program via `/dev/tap<N>`.
- The user space program is in case of a cloud QEMU which virtualizes NICs for guest OSes.
- When QEMU reads an Ethernet frame using the file descriptor of `/dev/tap<N>`, it emulates what a real network card would do:
  - It triggers an interrupt in the virtual machine, and the guest operating system can then read data from the emulated network interface card.

5.7.1.4 Inefficient Inter-VM Communication as Result

- In the figure below, the resulting inefficient inter-VM communication in the same server via KVM/QEMU and Macvtap is shown.
- An overhead of 11 sequential steps is visible which are all needed to accomplish one inter-VM send operation.
- The same block diagram holds in principle also for the case that additionally OpenStack is present, only more software components are then engaged as overhead.
5.7.2 Inter-VM Communication With Ivshmem

- Ivshmem allows for a zero-copy VM-to-Host communication, which is highly efficient with respect to bandwidth and latency, because no internal data buffer, no TCP/IP and no KVM/QEMU overhead exist.
- By going the way from VM 1 to host to VM 2, a shared memory communication between the two VMs comes into existence, also without copying any data buffer.
- It is possible to create ivshmem by means of libvirt.
- Furthermore, ivshmem is compatible with OpenStack and its way to define and launch VMs.

Example: If a user creates a VM by means of the Horizon dashboard of OpenStack, a libvirt configuration file is output by the dashboard in xml format. This file has to be post-processed by the user before it is input to KVM and QEMU, and thus before the VM is launched. The post-processing has to add ivshmem by a few xml tags, and ivshmem comes into existence.

- In the next figure, the resulting fast inter-VM communication is shown.
- It works as follows:
- The address space of the guest-OS application is first "mmapped" to ivshmem virtual PCI-memory and in a second step to Linux/POSIX SHM that really exists.

Comment: Mmap is a Linux system call.
The shared-memory access-synchronization is accomplished via an ivshmem server which is also part of QEMU. The ivshmem server allows that a VM can send an eventfd (= “interrupt“ ) to another VM. By sending ready/done interrupts back and forth between VMs, a mutual exclusion of a shared variable can be achieved which resides in physical SHM. The result is that a send needs only 1 variable write at the source + 1 read at the target. ivshmem is even faster than SHM itself because normally a Linux system call is needed to access an SHM buffer in kernel space. This is avoided here, because the kernel space buffer is mmapped into user space which is highly intricate, but also highly effective. The only disadvantage of ivshmem is that the user applications must be disguised as the user part of a Linux uio device driver. Additionally, the kernel part of a uio device driver must be written and loaded into the kernel.

5.8 Inter-VM Communication between Servers without Cloud OS

A block diagram for the inter-VM communication between servers is given for the case that an Ethernet and an external switch are present. The essence is that QEMU uses again macvtap, but switches it from Bridge Mode to VEPA Mode which performs non-local communication. Afterwards, macvtap forwards all Ethernet frames to the physical Ethernet NIC. According to the block diagram, a total of 13 steps is needed in order to send one TCP packet from the sender on the left side to a receiver on the right side. This is not efficient.
However, so far exists no faster possibility

5.9 Inter-VM Communication between Servers with Cloud OS

- In Amazon’s AWS EC2 cloud, only a relatively slow inter-VM communication between servers is possible that prevents from HPC and from fast data exchange between servers as well
- The same holds for Microsoft Azure and VMWare’s vCloud
- However, with a private OpenStack cloud, it is possible to influence the way inter-VM communication is accomplished:

  Example: In a a private OpenStack cloud, it is possible to replace Ethernet by the faster Infiniband interconnect. It is also possible to install in each guest OS a communication library that support Infiniband from user applications, such as MVAPICH2-Virt from Ohio State University, for example.

- MVAPICH2-Virt is a high-performance message-passing interface according to the MPI standard
- It supports HPC in a cloud that has Infiniband as interconnect and can deal with VMs and containers

  Comment: HPC = High-Performance Computing

- Hardware installations are not possible in public clouds
- This has to be known if a decision for a public cloud is made
- Because of the restrictions in public clouds, only OpenStack will be discussed in the following w.r.t. inter-VM communication between servers with cloud OS
5.9.1 Inter-VM Communication between Servers with OpenStack

- If OpenStack is added, another level of complexity is reached
- Two block diagrams that belong together will follow in order to show the intricate interplay between QEMU, KVM, VMs and OpenStack’s Neutron
- These block diagrams represent the state-of-the-art for inter-VM communication between servers in the presence of Ethernet, KVM/QEMUs and OpenStack as cloud OS
- The essence is that again 13 steps are needed for a single send, but this time the huge software stack of Neutron has to be traversed additionally
- The basic rationale of the block diagrams below is:
  - QEMU switches macvTap again in VEPA Mode, but this time macvTap does not write to the physical Ethernet NIC
  - Instead, MacvTap writes to the Neutron network service of OpenStack which has inserted a thick layer of software between the Tap port 2 of macvTap and the physical NIC
  - In that layer, vLANs are implemented in order to isolate VMs from different tenants from each other w.r.t. to data communication
  - Additionally, a fire wall was added by Neutron to increase cyber security
  - From that block diagrams it is clear that inter-VM Communication between Servers with OpenStack as cloud OS is highly inefficient
  - For public clouds, it es even worse
  - The consequence is that clouds are not made for HPC, but are good for everything else

```
4. QEMU 1 reads
5. QEMU 1 writes
10. QEMU 2 reads
11. QEMU 2 writes
```
Part II - OpenStack as Cloud OS
6 Overview on OpenStack and its Services

- OpenStack (http://www.openstack.org) is a large open source project that started in 2010
- Every 6 months, a new release appears
- The change frequency of OpenStack is very high, and each new release can contain fundamental changes
- OpenStack comprises > 20 Mio. lines of code
- This is about the size of the Linux kernel
- OpenStack provides an Infrastructure-as-a-Service (IaaS) by means of various services
- It is mostly written in Python 2.6 and Python 3 and thus interpreted

Comment: Python is a popular object-oriented script language from the 1990s. In contrast to C, C++, C# and Java, it is not based on { and } for BEGIN and END but on code indentation.

- Data structures that have to be exchanged to and from OpenStack services are often in “JSON“ format

Comment: JSON = JavaScript Object Notation. It is a syntax convention for data interchange that is simpler than XML and easier to read. It is not a markup language as XML, but JavaScript.

- Calling an OpenStack service means often to set a JSON object and use it as parameter for the call
- OpenStack has more than 45 services with steadily increasing tendency

- The key feature of these services is that most of them are calls via a REST-based protocol

Comment: Those few OpenStack services without own REST API can be called cloud-internally by a messaging library called “oslo.messaging“.

- Because of that, OpenStack can be considered as a set of web services
- Most of the services have their own REST API

6.1 REST Protocol

- REST is a superset of http because also PUT, DELETE and PATCH are working
- In case of http, these calls are typically blocked by a web server
- Additionally, an URI (Uniform Resource Identifier) is used instead of an URL

Def.: An URI is a generalization of an URL which addresses an abstract or concrete web resource instead of a file by means of a predefined syntax.

Example: An online storage is an abstract web resource. It is abstract because the user only knows which files he has and how much space is left, but he does not know where his files are physically located. A concrete web resource is a file.

- Finally, REST is also a set of defined rules (“architectural style“) that create a stateless web server
- There are 9 REST calls: GET, HEAD, PUT, POST, DELETE, TRACE, OPTIONS, CONNECT and PATCH
Many service APIs have three sets of REST calls: for cloud-internal admins, for cloud-internal users, and for the public
All public REST calls are secured by https, the others not
This is important for data security

6.2 Calling and Controlling an OpenStack Service

Most OpenStack services can be called via a service-specific REST API
For those services without API, a Universal Unique Identifier is used as address instead of an REST URI

*Def.: A Universal Unique Identifier (UUID) is a 16-Byte number in hexadecimal notation which is split into 5 groups that are separated by „-“. Everybody can create an UUID because it contains data, time and MAC address of the computer where it was created. Additionally, pseudo random numbers or names of any items may be included.*

*Example: 550e8400-e29b-11d4-a716-446655440000 would be a valid UUID.*

*Example: Each VM has an UUID, as well as each Neutron network.*

In addition to its UUID a VM has also a name, a Neutron network has not
Furthermore, all OpenStack services can be controlled locally via an service-specific a CLI

*Def.: CLI = command line interface = Command line interpreter for string-based commands.*

Finally, many OpenStack services can be controlled remotely via a GUI in the “Horizon dashboard“

Comment: For an explanation of the Horizon dashboard, see the subsequent chapter about it.

6.2.1 Extended UUID Usage for Information Items

In OpenStack, the UUID addressing scheme is extended to address important information items which are not a service and have therefore no API and no CLI

*Example: Every component in the user-, project- and service level in a cloud, as well as every information item have an own UUID.*

The UUID of an information item allows to reference to it or to call it via OpenStack-internal Remote Procedure Calls or to notify it via OpenStack-internal messaging
UUIDs for information items and URIs for services complement each other
Finally, those items which have no URI and no UUID are referenced by the name they have

*Example: Each Heat stack has a name and a UUID.*

Comment: For an explanation about what a Heat stack is see the subsequent chapter about Heat.
6.2.2 Calling via REST API

- Each REST request that is issued by a caller to a callee has an UUID in its request body
- The UUID appears in the log files of caller and callee
- This allows for tracking the service request for debugging
- Furthermore, not all 9 REST calls are typically defined for a specific service but mostly a subset of it

Example: An OpenStack service called murano has 5 of 9 calls: GET, POST, PUT, DELETE and PATCH. Murano lists the user applications that are installed in the cloud. Each user application is called a “package“ in murano.

Example: GET http://myCloudController.de:8082/v1/catalog/packages would by a syntactically valid REST request with http://myCloudController.de as the URI to the computer where murano may reside. The port 8082 is the default port for murano where it listens for requests.

- The example above would list the installed packages
- The same effect would be achieved if the URI would be entered into the browser URL line
- An OpenStack REST request to a specific service is meant for being used from inside of a software via the Internet
- This REST request must have a port number in its URI where the service is listening

An incomplete list of OpenStack-services ports is under: https://github.com/openstack-infra/tripleo-ci/blob/master/test-environments/tls-endpoints-public-ip.yaml

⇒ Most OpenStack services have at least one host OS port where it listens for API requests

- In newer OpenStack releases, the node which has the Keystone service installed, contains a host OS file of name /etc/keystone/default_catalog.templates in which the ports for all services are listed

6.2.3 Calling from Host OS Shell

- If a user wants to call an OpenStack service by hand from the Host OS shell he should use the open source tool “cURL“

Def.: cURL allows to copy the content of an URL or URI to a file and vice versa. “file“ as output can also be the own display. “file“ as input can be an string or a JSON object that is entered at the shell prompt by hand. cURL generates automatically a GET request if the source is the URI and the destination is the file, and a POST request for the reverse direction. All other REST methods must be explicitly given as cURL parameter.

- cURL option flags are explained under: https://curl.haxx.se/docs/manpage.html

Example: curl -H "X-Auth-Token:0123456709012" "http://myCloudController.de:13000/v2.0/tenants" is a REST request to the Keystone service that would list all projects (i.e. tenants) of the users in the cloud. -H is the “header option“ of cURL, and it is needed to
authenticate the call as being issued a by an admin. 0123456709012 is his example certificate. Under port number 13000, Keystone is listening for REST requests at the API network.

- An incomplete list of OpenStack REST APIs is under: https://docs.openstack.org/queens/api/
- The positive implication of OpenStack being a set of web services is that its calls always follow the same simple scheme <method> <URI>
- This scheme holds also if the API of a specific OpenStack service changes or if the underlying host OS changes

⇒ There is a static OpenStack API calling scheme despite changing versions of services and host OS

6.2.4 Calling via GUI

- Beside its set of individual service APIs, OpenStack provides to user and admin also a GUI via its “Horizon“ service

Comment: GUI = Graphical User Interface

- Horizon allows to control many OpenStack services and all VMs either directly or with service-specific plugins
- Horizon is a “dashboard“ for OpenStack with different authorization rights for admins and users

6.2.5 Caveats

- Because of the fact that OpenStack is a set of web services that are called via REST, speed and latency of its API are low
- As a consequence, clouds in general and OpenStack in special are normally not suited for real-time or HPC or big data transfers

6.2.6 Calling via CLI

- Finally, in addition to API and Horizon GUI, practically all OpenStack services provide also a CLI for the direct control of them

6.2.7 Calling via UUID

- Calling a service component via UUID is accomplished via the OpenStack-internal communication library called oslo.messaging

Comment: Oslo.messaging provides for inter-process communication in the same node and between nodes. For a deeper explanation see a following chapter about it.

6.3 Overview on OpenStack Services

- By means of its services, OpenStack allows to provide, configure and manage indirectly huge amounts of computers, hard drives, network components and user applications
Example: In a cluster of 10 Tsd. physical servers, an OpenStack cloud can be installed that hosts hundreds of thousands of VMs.

- The attribute "indirectly" holds for the following reason: each OpenStack service is a user space process only without system privileges in the host OS

⇒ It can provide, configure and manage only virtual resources but not real ones

- Furthermore, a distinction must be made whether an OpenStack service pertains to an existing software (user application) or not:

1.) If an OpenStack service pertains to existing software or application then it has no API call for using it, but only for its provisioning, configuration and management

Example: A database service called Trove can provision and manage multiple databases, but they are still accessed by SQL and not by REST calls. This means Trove is no competitor to SQL but preserves existing standards.

⇒ an OpenStack service for an existing software is not just another API for it, but it complements the existing API w.r.t. to cluster-wide handling. The service API does not compete with the existing API.

2.) If OpenStack services pertain to new functionalities that are provided only by the service itself then the usage of these functionalities is included in the REST API

Example: An object store service called Swift can not only provision and manage an object store, but it can also read, write and delete the objects inside of the store. In fact, the Swift API is the only way to access these objects.

- The first main idea of OpenStack, and any other cloud as well, is to extend existing software and applications that are written for a single computer to a whole cluster of computers by giving powerful tools for their cluster-wide provisioning, configuration and management in VMs, but not for their usage

- The second main idea of OpenStack, and any other cloud as well, is to allow for new, cluster-wide services that did not exist before

- In the latter case, the REST API includes not only provisioning, configuration and management but also usage

- An incomplete list of OpenStack services is:

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Catalog</td>
<td>Murano</td>
<td>Databases</td>
<td>Trove</td>
<td>NFV Orchestration</td>
<td>Tacker</td>
</tr>
<tr>
<td>Bare Metal</td>
<td>Ironic</td>
<td>DNS</td>
<td>Designate</td>
<td>Object Storage</td>
<td>Swift</td>
</tr>
<tr>
<td>Block Storage</td>
<td>Cinder</td>
<td>Identity</td>
<td>Keystone</td>
<td>Orchestration</td>
<td>Heat</td>
</tr>
<tr>
<td>Clustering</td>
<td>Senlin</td>
<td>VM Images</td>
<td>Glance</td>
<td>Searching</td>
<td>Searchlight</td>
</tr>
<tr>
<td>Compute</td>
<td>Nova</td>
<td>Infrastructure</td>
<td>Watcher</td>
<td>Shared File Systems</td>
<td>Manila</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimization</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In addition to OpenStack services, there are also OpenStack projects which may cumulate later into a full service, or which will result into a newer release of an existing service

OpenStack projects are listed under: https://www.openstack.org/software/

Finally, there are also OpenStack libraries which are a software infrastructure for services and projects

The key feature of every service is that it is one central software for all VMs that is specialized on a specific task, and that can be controlled by one dashboard (Horizon)

This simplifies significantly the management of VMs

Example: In a cloud of 1000 physical servers, a new user has to be added to the keystone service only one time by means of Horizon but not 1000 times.

### 6.4 The 10 Most-Important OpenStack Services

The following is a subjective selection of the 10 most important OpenStack services which were taken from https://docs.openstack.org/queens/projects.html

1.) **Horizon** allows to provide, configure and manage VMs and cloud services via a web-based GUI
   - a Horizon dashboard allows the user to login into the cloud
   - it gives to users and admins the “look and feel“ of OpenStack
   - Horizon has therefore web pages for users, admins and settings
   - it has additionally an API to many but not to all OpenStack services
   - it has finally Python classes in order to integrate new services and projects into it
   - there is furthermore a tool called “OpenStack Client“ which provides a CLI for many but not all OpenStack services

2.) A **meta scheduler** named “**Nova**“ which statically allocates VMs to real servers

Comment: Each operating system has a scheduler that allocates tasks to cores in order to execute them in a round-robin manner. A meta scheduler does not works on the host OS or guest OS level but on the cluster level and allocates VMs to cores in servers.

   - VMs that are created by the user in Horizon are forward to Nova for processing the VM creation request and for scheduling the requested VM after it is implemented
     - Nova in turn uses KVM/QEMU to truly implement the VMs
     - with each VM, comes also a virtual hard drive as part of the VM
however, this virtual hard drive is ephemeral because data will be lost when the VM is terminated
by means of Nova, each VM gets a physical server in a cluster that will execute the VM inside of the assigned server, the local host OS scheduler decides which core in which CPU is executing the VM
Bare metal servers can also be created by Nova which uses the Ironic service for implementation
Finally, there is some support for containers in Nova

3.) A network Manager named “Neutron” which manages virtual network equipment, such as vNICs, switches, routers, gateways, including MAC and IP addresses of VMs
• Neutron provides for each VM the virtual network over which it can connect to other VMs in the same cloud or to the Internet
• it can operate either on ISO layer 2 or additionally also on layer 3
• it provides for a Software Defined Network (SDN)

Comment: SDNs is a concept that is used frequently in compute and data centers. SDN divides the coupling and management of network components inside of the center into two parts: the data plane and the control plane.

4.) An identity management system named „Keystone“ which provides for every user of a VM authentication and authorization

Comment: Authentication = username and password or RSA access key for a VM. Authorization = Access rights inside of a guest OS with respect to read, execute and delete files.

• additionally, Keystone provides for every OpenStack service a short-term certificate without it cannot run
• Finally, it maintains a catalog where every service can register and enter its URI

5.) A 1st storage service named “Cinder“ which enables a reliable and distributed file system for disk records in virtual hard drives

Comment: A disk record is a fixed amount of storage of 2, 4 or 8 KB, depending on disk formatting.

Example: Relational data bases are optimized for record-based disks.

• Cinder provides, configures and manages virtual volumes which can be RAIDs for the whole cluster that are optionally also accessible from multiple VMs

Comment: RAID = Redundant Array of Inexpensive Disks

• it does not have calls to read, write or delete records in a volume. This is accomplished via standard readf, printf calls
• it provides for each volume a cluster-wide unique address (UUID) under which readf, printf can access the volume and its records
• it backups whole volumes by snapshots
Comment: UUID = Universal Unique Identifier

6.) A 2nd storage service named “Swift“ which allows for a reliable and distributed file system for objects in virtual hard drives

Def.: Swift objects are data together with metadata that explain the data.
Example: A photo with title, an email with header, or a file backup together with the names of the creator and the hard drive are objects.

- Swift distributes objects redundantly over virtual drives in the cluster
- It allows to read, write and delete Petabytes for big data applications

7.) A 3rd storage service named “Glance“ to store objects as Swift does
   - objects are stored in JSON format as keyword/value pairs
   - Glance is mostly used to store whole VMs in a single file which is called image
   - Glance can use Swift as storage backend
   - it provides a repository for images
   - All VMs are launched from glance images

8.) A 4th storage service named “Trove“ that provides, configures and manages both, SQL and “No-SQL“ databases

   Def.: A No-SQL database does not store properties that belong together by means of entries in the same row of a table. Instead, it follows other structuring principles, that depend on the concrete No-SQL database. There are various other principles for data structuring such as keyword/value pairs or object-oriented.

   - Trove uses Cinder as storage backend because SQL databases are optimized for record-oriented storages

9.) A measurement service named “Ceilometer“ which allows to collect and display any kind of statistical data from the cloud
   - Ceilometer allows to collect performance data for cloud tuning

   Example: CPU utilization, hard drive accesses per minute or vNIC bandwidth are typical statistical data.

   - it also allows to collect resource-usage data the CSP needs to bill customers
   - the definition of what has to be collected is accomplished by counter variables
   - as soon as admin or user has defined which data he wants to collect, Ceilometer does this autonomously

10.) A batch processing system called “Heat“ that can execute so-called templates, as well as scripts in various languages in order to make calls to OpenStack service APIs and to host-OS shells

   Def.: Heat scripts are named “stack“. The supported script languages are Chef, Puppet and Ansible.

   Comment: Chef, Puppet and Ansible are scripting languages that are optimized for an automated application deployment and configuration management.

   - by Heat, OpenStack services and VMs can be automatically deployed, started and re-configured
Example: Heat allows to provision automatically at a specific point-in-time a given amount of VMs and to decommission them later by auto-scaling in order to adapt to user habits.

- The listed 10 most important OpenStack services will be outlined in more detail in subsequent chapters

6.5 The Smallest Possible OpenStack System

- The smallest possible OpenStack System consists from a software point-of-view of 5 of the 10 most important services: Nova, Neutron, Glance, Keystone and Horizon
- These services have to be installed at least on one physical server which acts as a so-called controller node for OpenStack
- By means of the controller node, multiple VMs can be launched in turn that act as so-called compute nodes
- Typically, VMs are launched on other physical servers or PCs than the controller node
- As a consequence, the smallest OpenStack System consists from a hardware point-of-view of two physical computers, one that is the controller node and the other that is the compute node with several VMs
- For testing purposes, it is even possible to install the controller node not in a physical computer but in a VM, and to install the compute nodes even in the same VM as nested VMs inside of a VM

- Such an all-in-one VM is the truly smallest possible OpenStack system, but this system is more academic than a high-performance solution
- Subsequently, to the smallest system with performance more physical nodes with VMs can be added as needed
- It is also possible that some VMs are specialized for a specific OpenStack service
  
  Example: With exception of Horizon, Keystone, Nova and Neutron, all other OpenStack services such as Cinder, Swift, Glance, Trove and so on can specialize a compute VM to a record-based storage-VM, to an object-based storage-VM, to an image VM or to a database VM, or to s.th. else.

- In fact, compute VMs on which one or more OpenStack services are installed (with exception of Horizon, Keystone, Nova and Neutron) are specialized compute VMs only
- The block diagram of the smallest OpenStack system of performance is as below:
  
  ![Block diagram of the smallest OpenStack system](image)

- Inside of each compute node, user applications are executed in VMs
- Instead of VMs, also containers or bare metal servers are possible
- Instead of compute VMs, specialized VMs are also possible, such as those given in the example above

6.6 Controller Node and Other Nodes

- In order to be able to control one or more compute nodes or specialized nodes from a controller, each other node must be connected via a physical network to the controller
This means the controller NIC is either connected directly to a free NIC in the controller, or it is indirectly connected via a switch or router.

On top of the physical network sits a virtual network that connects controller node services to compute node services and VMs.

The virtual and physical networks transmit TCP or UDP-based application protocols.

In case of OpenStack, the main application protocol is REST, but two other application protocols exist as well: “AMQP“ and “ZMTP“.

Comment: AMQP and ZMTP are so-called messaging protocols and will be explained later.

In order to integrate a VM of a compute node into the virtual network, a part of the Neutron service must be present in the compute node that hosts the VM.

This part of Neutron is the “Neutron-Linux Bridge“ which is inserted by OpenStack between KVM and each VM vNIC.

It connects the vNIC of a compute VM to a vNIC Neutron manages at the controller.

### 6.7 Minimum Software Stack in a Compute Node

Beside the Linux Bridge for networking, a second software layer is needed.

This layer is part of Nova and called “Nova-Compute“.

It is also inserted by OpenStack between hypervisor and the VMs.

The minimum software for a compute node is therefore: host OS with KVM, Neutron-Linux Bridge and Nova-Compute.

The latter two are user space processes in the host OS.

On top of that infrastructure, we have several QEMUs with guest OSes which are also host-OS user-space processes.

In each guest OS, we have multiple user applications, which are host-OS user-space processes too.
6.8 Intra-Service and Inter-Service Communication

- Beside the REST API each service has, there is one more OpenStack mean for communication which is called oslo.messaging

6.8.1 oslo.messaging

- oslo.messaging provides for both, intra-service and inter-service communication
- Intra-service communication is needed for large services that consist of several processes so that the processes can exchange data with each other on the application level

  Example: Nova is a large service that has 4 components and a database. All of them are daemon processes that can even be executed on different physical computers. This is possible by means of intra-service communication via oslo.messaging.

- oslo.messaging is an OpenStack library for Remote Procedure Calls (RPCs) and for notifications

  Def.: A RPC is a function call a client makes at computer A that is transmitted over the Internet to a RPC server B which executes it. The result is returned to the client at A. Typically, the RPCs is blocking the client until the result has returned, as a local function call would do.

  Def.: A notification is a one-way message from a sender to one or more clients. Typically, notifications do not block the sender after transmission.

- This means that intra-service communication is possible either by RPCs or by notifications

  Example: Nova uses RPCs and notifications, ceilometer uses notifications.

- RPCs and notification are directed to other process components of the same service or to other services
- In oslo.messaging, RPC and notifications are implemented by sending one or more messages either by AMQP or alternatively by ZMTP

6.8.1.1 Advanced Message Queuing Protocol (AMQP)

- AMQP is a protocol for connection-oriented message-exchange on ISO layer 7
- It supports that a broker can publish messages to those receivers that have subscribed to it
- Messages have to be queued at a message broker in a mail-box-like manner to allow for asynchronous communication
- AMQP allows for routing between message brokers that are located between sender and receiver
- It provides for flow control on a message-basis between sender and receiver to avoid receiver overrun
- It supports authentication of the sender and encryption of the message
In case of transmission problems, a message is delivered either at-most-once, exactly-once or at-least-once, depending on its setting.

For better interoperability between heterogenous computer systems, AMQP converts variables of standard data types into an own intermediate exchange format and transmits them serially as bytes (= marshalling).

Additional metadata allows to annotate the own intermediate exchange format in order to create user-specific data-types as extensions.

In OpenStack, AMQP is implemented by RabbitMQ which is a AMQP message broker.

### 6.8.1.2 ZeroMQ Message Transfer Protocol (ZMTP)

ZMTP is as AMQP an ISO layer 7 protocol for message exchange, but without intermediate message brokers and with other features.

As AMQP, ZMTP supports also security and metadata for type extensions.

It generalizes on layer 7 the Berkeley sockets from one-to-one communication to many-to-many communication by defining sets of senders and sets of receivers.

It supports between a sender set and a receiver set three transaction schemes which are publish-subscribe, request-response and push-pull.

Publish-subscribe allows to connect a set of message publishers to a set of subscribers:
- each publisher can send a message to all subscribers (=multicast)
- each subscriber can receive messages from all publishers

Request-response allows to forward a task processing request from a client in the client set to a server in the server set:
- this is well suited for implementing RPCs

push-pull allows one client to scatter at the same time a set of task processing requests to multiple servers and to gather the results back:
- this is well suited for implementing parallel processing according to the master-slave principle

In OpenStack, ZMTP is implemented by ZeroMQ which is part of the oslo.messaging library.

### 6.8.1.3 Notifications

A notification is put by its originator onto a so-called “message bus“ by means of its Notifier class which is part of oslo.messaging.

There is one single message bus per message broker and a routing of notifications between brokers, i.e. between message buses.

A notification consists of two parts: an envelope with a fixed structure defined by oslo.messaging and a payload defined by the originator emitting the notification.

### 6.9 The Horizon Service

Horizon is web-based and uses the Django framework.

Comment: Django is a popular Python framework for the creation of web pages with a database behind.
6.9.1 Horizon GUI

- The first web page that may be displayed after login is the “settings tab“:

- The user web page of the project tab looks as follows:
The user web page of the identity tab is:

- Admins have access to similar but not identical tabs and sub-tabs
- Other tabs and options are also available

**Example:** The options “delete networks”, “delete routers” or “delete instances” are only available to an admin, as well as the admin tab.

**Example:** The compute tab for admins has different sub-tabs compared to the user. Also the overview tab is different.

---

**Example:** Admins have the following overview GUI:

---

**6.9.2 List of Tabs and Sub-tabs for the User GUI**

- The following list is from https://docs.openstack.org/horizon/latest/user/log-in.html
6.9.2.1 Compute Tab

1.) Overview: View reports for the project

2.) Instances: View, launch, create a snapshot from, stop, pause, or reboot instances, or connect to them through VNC

3.) Images: View images and instance snapshots created by project users, plus any images that are publicly available
   · Create, edit, and delete images, and launch instances from images and snapshots

4.) Key Pairs: View, create, edit, import, and delete key pairs

   Def.: “Key pairs" is only the public key part of an OpenSSH key pair for accessing a VM during login. The private key part is secret, also to the admin.

6.9.2.2 Volume Tab


1.) Volumes: View, create, edit and delete volumes

2.) Backups: View, create, edit and delete backups

3.) Snapshots: View, create, edit and delete volume snapshots

4.) Volume Groups: View, create, edit, and delete a group of volumes

   Def.: A volume group is used in Cinder for managing a set of volumes consistently.

   Example: Snapshots and backups for disaster recovery can be accomplished for a whole volume group. Without defining a volume group, admins cannot easily provide crash-consistent data-protection across multiple volumes.

6.9.2.3 Network Tab

1.) Network Topology: View the network topology

2.) Networks: Create and manage public and private vLANs

3.) Routers: Create and manage routers

4.) Security Groups: View, create, edit and delete security groups and security group rules

   Def.: An OpenStack security group is a set of vNICs of VMs with the same firewall rules. A vNIC of a VM can belong to one or more security groups. All traffic that is inbound to and outbound from a VM is checked by Neutron whether it matches the firewall rules. Only packets that match the rules can pass. In case of mismatch, Neutron blocks ports, port ranges, or traffic types for that VM.

5.) Floating IPs: Allocate an IP address to or release it from a project
6.9.2.4 Object Store Tab

- Containers: Create and manage containers and objects
  Comment: So far, containers are only weakly supported by Horizon. Physical machines are not supported at all.

6.9.3 List of Tabs and Sub-tabs for the Admin GUI

- The following list is from https://docs.openstack.org/horizon/latest/user/log-in.html

6.9.3.1 Overview Tab

- Overview: View basic reports

6.9.3.2 Compute Tab

1.) Hypervisors: View the hypervisor summary

2.) Host Aggregates: View, create and edit host aggregates
   - also view the list of availability zones

**Def.: An admin can subdivide a cloud into availability zones which are disjunct subsets of VMs. Each VM can be only in one availability zone. The user can see in which availability zone(s) he is. Only the VMs of his availability zone(s) can be accessed by him.**

- The computer centers that implement a cloud can be distributed around the world

- Each center defines an own availability zone

- Technically, the availability zone is metadata information attached to a so-called “aggregate“

- Each availability zone should have independent access to network, power, and cooling infrastructure to ensure uninterrupted access to data

- The advantages of availability zones are:
  - There is a full isolation between them, such that natural disasters and power outages do not tear down the whole cloud
  - User data can be redundantly stored in different availability zones which increases the cloud's reliability

**Def.: An admin can organize an availability zone into non-disjunct subsets which are called host aggregates. Host aggregates are only visible to admins. A VM can belong to multiple host aggregates inside of the same availability zone.**

- The advantages of the host aggregates are:
  - The admin can assign the same public/private OpenSSH key-pair to all VMs of a host aggregate by one command,
  - The admin can migrate all VMs of a host aggregate by one command
  - Nova can allocate all VMs of a host aggregate to the same physical server

3.) Instances: View, pause, resume, suspend, migrate, soft or hard reboot and delete VMs
· view the log of a VM
· access a VM through VNC

**Def.: VNC = Virtual Network Computing.** This is a software that displays the windows of a remote computer on the own computer, and that redirects the own mouse and keyboard to the remote computer. This allows to work at a remote computer while sitting at the own computer.

4.) **Flavors:** View, create, edit and view extra specifications for flavors and delete flavors

**Def.: A flavor is the size of a VM w.r.t to number of CPUs, cores per CPU, main memory disk space and number of vNICs.**

5.) **Images:** View, create, edit properties for images and delete custom images

### 6.9.3.3 Volume Tab

1.) **Volumes:** View, create, manage and delete volumes
2.) **Snapshots:** View, manage, and delete volume snapshots
3.) **Volume Types:** View, create, manage and delete volume types

### 6.9.3.4 Network Tab

1.) **Networks:** View, create, edit properties for virtual networks and delete them
2.) **Routers:** View, create, edit properties for virtual routers and delete them
3.) **Floating IPs:** Allocate dynamically an IP address to a project or remove it

### 6.9.3.5 System Tab

1.) **Defaults:** View default quota values

**Def.: Quotas are hard-coded in Nova-Compute and define the maximum allowable size and number of resources**

Example: View max. main memory, max. disk space, max. number of vCPUs per CPU, max. number of CPUs per VM, ...

2.) **Metadata Definitions:** Import namespace and view the metadata information

**Def.: A namespace is generally a method to limit the validity scope of identifiers (“names”) to specific areas inside of a system. Namespaces allow to re-use the same name (identifier) in different contexts without creating confusion because the namespaces of the same identifier are different.**

Example: Linux has namespaces for processes and network interfaces. They are used for Linux control groups (cgroups). In OpenStack, Neutron uses namespaces to restrict identifiers to a certain router or to a certain protocol.

**Def.: Metadata are generally used to explain the semantics of data.**
Example: In OpenStack, VM images can have metadata that explain the image type and to schedule it by Nova after it is launched as a VM. For that purpose, Glance has an own metadata definition service.

3.) System Information: Use the following tabs to view the service information:
   - Services: View a list of the services
   - Compute Services: View a list of all Compute services
Comment: Compute service = Nova. Useful command to find out which of the 5 processes of Nova are running.
   - Block Storage Services: View a list of all Block Storage services
Comment: Block Storage service = Cinder. Useful command to find out which process components of Cinder are running.

4.) Network Agents: View the network agents
Example: Neutron-dhcp-agent, neutron-L3-agent, neutron-metering-agent, neutron-LBaasS-agent, ...
Comment: LBaaS = Load Balancing as a Service.

6.9.4 Horizon Projects (Tenants) And User Authorization
- A Horizon project is identical to the tenant notion we used previously
- Each user has a role in the project he is member of
- For each user role, parts or all access rights are given by Keystone
- Full access rights means to create, read, update and delete (CRUD) a file in Cinder or an entry in Trove or an object in Swift or in Glance
- Full access rights means also to call all API requests of all services
- Full access rights means finally to use virtual network equipment such as vNICs and virtual hardware, such as vCPUs without limitations
- Typically, only admins have full access rights in a project, and in all projects as well
- The user role is basis for the Role Based Access Control of OpenStack (RBAC)
- RBAC means that each user role has an own set of rights for access which is called user authorization
- If a user is member of multiple projects, then it has a role set with individual access rights per project

6.9.5 Extensions to Horizon
- Horizon allows applications and new services to “register“ an own dashboard which will become part of Horizon afterwards
- This means that a new GUI is hooked into Horizon's Python code in order to extend it
- Horizon extensions are also alleviated by Horizon code templates
- Additionally, new panels and panel groups can be added to Horizon without modifying the default settings
Furthermore, Horizon allows for “Pluggable Settings“
Pluggable settings mean that a Horizon configuration can be stored in separate files
Those files are read at startup and modify the default settings
Finally, Horizon code is well structured because its web pages consist of many panels and not one big panel
This allows to augment and customize Horizon easily

6.9.6 Abstract Service API of Horizon

- Horizon has a set of methods to communicate with the most important OpenStack services without knowing their concrete APIs
- These methods form sets of abstract APIs which are independant of new API releases of the services

⇒ If some service has a new API release the consequence is not the case that a new Horizon release must accompany it

6.9.7 Horizon GUI Terminology

- Horizon can present multiple dashboards as GUIs to the user
- Each dashboard has an own URL and must be registered at Horizon

Example: Dashboard.py is the base class of a GUI.

A dashboard comprises multiple panel groups
A panel group contains multiple panels which establish a drop down menu
A panel group has click-able tabs as elements
It is configured in dashboard.py
Panels are the main components of each web page
Each panel has an own Python class

⇒ No class is overwhelmingly big

- Each panel code is in an own directory and has a standardized structure
- Panels are configured in dashboard/panel/panel.py
- A tab group is a set of tabs within a panel
- A tab group is configured in tabs.py
- Each tab has an own internal data set
- A workflow is a set of internal steps that enable Horizon to accept new input from the user
- A workflow step results typically in one internal action

Example: LinkAction, FilterAction or DeleteAction. A LinkAction enters a new object into a table, a FilterAction selects what will be displayed to the user and a DeleteAction removes an object from a table.

- A SQL database contains which action has to be taken for which step of the workflow
This can be a DB private to Horizon or the general DB of the whole cloud

- Tables are used by Horizon as internal data structures for presenting information to the user
- Some of the tables, such as those for step actions e.g., are stored in a database
- Views connect a table with a panel
- Views are responsible for what content of the table is displayed to the user
- Views are configured in view.py which in turn is based on utils.py which is a Django class

### 6.10 General OpenStack Terminology

- Outside of OpenStack, the term driver pertains to “device driver“ and is part of an operating system
- OpenStack drivers are s.th. completely different

**Def.:** Driver: in a service or in a component of it, a driver written in Python is used to run a host-OS shell-command or any other code that is not in Python.

**Example:** There is an OpenStack AMQP driver for the RabbitMQ message broker because the latter is written in Erlang.

**Example:** There are about 55 OpenStack drivers for 55 volume types from 55 vendors which are simply a Python proxy for the real device drivers written in C for Linux.

**Def.:** Manager: in the database of a service, a manager is responsible for updating entries

- Components or services which want to change entries in the database of an other service or components of it should engage the manager of that database to do so

**Example:** Services that deal with Cinder volumes, should call methods of the Cinder VolumeManager instead of changing fields in the Cinder database directly. This allows to keep all code related to Cinder volumes inside of Cinder.

- The concept of managers is generalized in OpenStack as follows:

**Def.:** Manager: rather than attaching Python methods to Python objects, services should call manager methods that act on these objects.

**Comment:** This schema extends the concept of object-oriented programming.

- If a remote method for a Python object must be called by a service, this should be accomplished via an RPC to that service that hosts the manager with the needed method

**Example:** The Nova-API component has a manager that listens for RPCs and for REST calls.

- Many services have so-called API components which are more than only APIs
- API stands normally for Application Programming Interface
Def.: API: an “API“ does not only provide for an API, but it is an own component running in a host OS process with various tasks beside implementing an API for a service. Typically, an API component contains a manager.

Def.: Each physical server is called a node. Each node can host multiple VMs (=instances)

6.11 The Nova Service

- Beside scheduling VMs and containers to servers and providing ephemeral disk storage inside of the VM, Nova implements the concepts of availability zones and “compute cells“
- By these concepts, OpenStack becomes scalable from small to large number of VMs and resilient against failures of individual computer-center locations a CSP has for its OpenStack cloud
- To summarize, Nova is one of the first and most important services in OpenStack

6.11.1 Nova Compute Cells

- For large OpenStack clouds, the single database Nova has becomes a bottleneck, as well as the single message bus of that AMQP message broker that is responsible for Nova
- Both issues can be solved by Nova compute cells

- In each compute cell, there is an own Nova database and an own RabbitMQ message broker that provides for several VMs a new message bus with queue
  ⇒ With Nova compute cells, the name of a VM is not longer enough information to find it
- Instead, the respective Nova database that stores it and the message bus that is responsible for it must be given as additional information
- On the other hand, small OpenStack installations take place in one cell only
- If multiple Nova cells are present then a “global“ Nova SQL DB is needed to store “global“ Nova data in parallel to local data

Example: Global Nova data are: VM_types i.e. flavors, VM_projects, VM_type_extra_specs, quotas, project_user_quotas, quota_classes, quota_usages, security_groups, security_group_rules, security_group_default_rules, provider_fw_rules, RSA key_pairs and migration network tags.

Example: Local Nova data are: VMs, VM_info_caches, VM_extra, VM_metadata, VM_system_metadata, VM_faults, VM_actions, VM_events, VM_id_mappings, pci_devices, block_device_mapping and virtual_interfaces.

- The global Nova SQL database is called “API” database, while each local database is called cell1
- VMs that failed to start are stored locally in a cell0 DB for every cell
6.11.2 Nova Internal Setup

- Nova consists of the following host OS processes: API component, Scheduler, Conductor, Compute and database:

  - **Nova-API component**: receives REST requests from other services and RPCs from other components of Nova and forwards them via oslo.messaging to third Nova components

  - **Nova-Scheduler**: decides which physical server gets which instance
    - The scheduler is a complex component that even has an own API beside the main Nova API
    - the scheduler API is called "Placement API"
    - the scheduler takes for its placement decision into account which flavor each free VM has and how far it is away from the user with respect to network latency, for example

  - **Nova-Compute**: runs on each compute VM
    - triggers KVM/QEMU to create and launch an image or to decommission a running VM
    - triggers also KVM/QEMU to create a virtual disk in the VM

  - **Nova-Database**: stores the flavor and server allocation of each VM
    - is a standard open-source SQL database such as MySQL

  - **Nova-Conductor**: the Nova-conductor is for any Nova-compute-component the manager of the local and global Nova databases
    - it acts as a proxy for SQL database requests from a Nova-Compute component
    - it handles notification requests that need coordination among Nova components
    - it handles Python object conversions

6.11.3 Parallel Processing Inside of Nova

- With exception of the database, all components can exist as multiple parallel processes on one or more physical computers in order to increase Nova's throughput
- However, the Nova-compute component must exist only one time per node
6.11.4 Communication Between Nova Components

- Nova components communicate with each other via RPCs and notifications that are both part of oslo.messaging

*Example:* Nova has about 50 different notifications and about 200 REST API calls, including the Placement API.

- Inside of each Nova component exists an **RPC manager** that listens for remote procedure calls from other components
- Each RPC manager is externally connected to an RabbitMQ broker for AMQP
- RPCs are implemented for Nova in oslo.messaging via AMQP and thus with message queues
- More precise: the publish/subscribe feature of AMQP is used to implement Nova RPCs
- Because of that, Nova RPCs do not block, i.e. they are **asynchronous**
- For the same reasons, the RPC client also does not know where the server of the call is located, i.e. which oslo.messaging ID it has

**⇒ Nova RPC servers are anonymous for Nova RPC clients**

- Nova RPC client calls are load-balanced between Nova RPC servers, i.e. between parallel processes of the same Nova component

6.12 The Neutron Service

- Beside Nova, Neutron is the 2nd most important OpenStack service
- Its API has a similar complexity as the Neutron APIs together
- Neutron has many tasks which are managed together with plug-ins and drivers

6.12.1 Neutron Task List

- A list of Neutron tasks follows that is needed for the management of the virtual network equipment in a cloud:
  - **tasks on ISO layer 2:**
    - bridging between two VMs which communicate with each other in the same node via TCP/IP and "Linux bridge"
    
      Comment: Linux bridge is a kernel module that connects two NICs or vNICs. A Linux bridge also allows to plug a 1st VM via its vNIC into the vNIC of a 2nd VM that has already an Internet connection by means of a 3rd vNIC. Then, the 1st VM can use the Internet through the 2nd VM.
    
    - switching between multiple VMs which communicate with each other in the same node via TCP/IP and "Open vSwitch"
    
      Comment: Open vSwitch is a Linux kernel module that emulates a physical switch.
    
    - for vLAN creation and vLAN ID tagging to isolate tenant vLANs from each other that re-
side on the same physical LAN
· for the provisioning of MAC addresses to vNICs of VMs

Lemma: Please note: Nova, not Neutron is required to plug a vNIC of a VM into a particular Neutron network, because the vNIC is emulated by KVM/QEMU which in turn is controlled by Nova.

· tasks on ISO layer 3:
  · for the provisioning of IP addresses and other network parameters such as the Gateway IP address to VMs by means of DHCP

Comment: DHCP = Dynamic Host Configuration Protocol.
  · for routing IP packets from vNICs to physical NICs via virtual network equipment
  · for Virtual Extensible LANs (VXLANs) to increase the number of vLANs in very large OpenStack clouds
  · for Network Address Translation to replace the private IP address of a VM by a public IP address that is valid for the Internet (=source NAT)

Comment: For each cloud, there is only a limited set of public IP addresses which is typically much smaller than the whole amount of VMs the cloud has. This is why public IP addresses are “floating“ between VMs. Floating means that the private IP address of a VM is translated temporarily into a public IP address by means of source NAT, and after a while, an other private IP address of a VM is translated temporarily into the same public IP address.

  · source NAT is accomplished via the L3 agent of Neutron
  · for booting a VM via a specific vNIC and network
  · for monitoring IP traffic via Netflow

Comment: Netflow is a software that instruments a router to collect IP addresses and layer 4 port addresses and other L3 and L4 information about routed packets.
  · for tunneling to a destination network via VPN

Comment: VPN = Virtual Private Network is a software that allows to connect the own computer to a remote private network via the Internet as if the own computer were directly connecty to the private network.

· tasks on ISO layer 4:
  · plug and unplug virtual ports

Def.: Whenever a vNIC of a VM, a switch or a router is connected to a OpenStack subnet, that connection is called a port. OpenStack ports are not identical to Linux ports, because the latter are associated with a TCP/UDP port number, and because one can associate external Internet IP addresses with OpenStack ports.

· tasks on ISO layer 7:
  · for establishing security groups. They are the basis for firewalls.
  · for creating a Security Fire Wall as a service (FWaaS)
  · for establishing a Load-Balancer as a service (LBaaS)

6.12.2 Neutron Software Defined Networks

- These set of tasks allow Neutron to create a Software Defined Network (SDN) for either ISO layer 2 only or for layers 2 and 3
By Neutron, whole connected sets of virtual networks with arbitrary topology can be created that include also virtual routers which is needed for a large cloud.

To create and manage Neutron's virtual networks by means of a GUI, Horizon is needed.

### 6.12.3 Neutron Components for ISO Layer 2 Operation

If Neutron has only ISO layer 2 virtual network equipment then the following Neutron components are sufficient for the controller node:

- optionally the modular Layer 2 plugin (ML2 plugin)
- Linux bridge agent
- DHCP agent
- optional Metadata agent. Metadata are SSH keys, for example.

Additionally, Neutron needs the Linux network utilities which are always present in host OS.

### 6.12.4 Neutron Components for ISO Layer 3 Operation

If Neutron has additionally ISO layer 3 virtual network equipment then it needs its an so-called L3 agent as well.

### 6.12.5 Further Neutron Components

Further neutron components are:

- a SQL database to store permanently data for Neutron and some plug-ins
  - This can be a DB private to Neutron or the general DB of the whole cloud
- a Neutron-server:
  - it accepts and forwards REST API calls to the appropriate Neutron component
  - it implements additionally core L2 and L3 features such as allocation a private IP address to an OpenStack port
  - it is also the manager for the database by means of a driver
- numerous other agents, plug-ins and drivers as options
  - agents are Neutron-internal and manage drivers and plug-ins which are external non-OpenStack codes
  - a plug-in implements by means of a driver the virtual-to-physical transition and is therefore important

**Lemma:** Please note: only one plug-in can be used at a time in Neutron to avoid conflicts. Either the ML2 plug-in or the plug-ins mentioned below can be used alternatively.

- One plug-in can have multiple drivers
  - each driver talks to a specific physical hardware such as a switch, e.g.

**Example:** Neutron comes with agents and their plug-ins and drivers for Open vSwitch and Linux bridge, for Cisco switches, for NEC OpenFlow products, for the VMware NSX product, as well as for many other vendors.
All Neutron components are host OS processes that are communicating internally via RPCs over a message bus from oslo.messaging

With exception of some agents, Neutron processes are located on the controller node

Some but not all agents are distributed onto all specialized and non-specialized compute nodes and run as user-space processes in remote host OSes

Also the remote agents communicate with the Neutron server in the controller node via RPCs over oslo.messaging

6.12.6Types of Data to be Exchanged in OpenStack

In total, there are 5 types of data that have to be exchanged in OpenStack: public data, private data, management data, REST data and Intra-service data

1.) Public-data exchange
   • this is implemented by the Internet to which the cloud is connected

2.) Private-data exchange
   • this is implemented by the data plane of a Neutron SDN that spans all VMs in order connect them
   • In OpenStack terminology, the data plane is also called Guest Network
   • the data plane is a set of Neutron vLANs coupled by virtual switches that may contain also virtual routers

3.) Management-data exchange
   • this is implemented by the control plane of SDN that spans all virtual switches and routers in order to manage them
   • the control plane is another set of Neutron vLANs coupled by virtual switches and does not contain routers

4.) REST-data exchange: this is implemented by the so-called “API network”
   • REST data are created by projects, i.e. tenants that call OpenStack services via their REST APIs
   • the API network is typically a Neutron vLAN that is connected to the Internet via a physical gateway
   • if the project has project VMs then the vNIC of each project VM is “source NATted” to a public IP address of the cloud
   • access to the API network is only possible from outside via https

5.) Intra-service-data exchange
   • this is implemented by oslo.messaging via message bus(es) that connect components of the same OpenStack service

6.12.7Example of Data-Type Exchanges

The block diagram above illustrates the data types and their exchange for the following use case:
   • the intra-component data-exchange is not shown above for reasons of simplicity
   • the Glance and Cinder services are allocated to the Controller node
• there is an own node for Horizon
• one compute node is specialized to an SDN node
• some compute nodes are specialized to network nodes to improve the connectivity to the Internet
• each network node implements a cell and a security group
• the network nodes have an own Neutron vLAN that is connected to the Internet via a physical router and gateway
• the same holds also for the API network
• both, the vLAN of the the network nodes and the API network are NAT-translated to public IP addresses of the cloud

6.12.8 Example Neutron Network Topology

- In the example neutron network topology, project A has 2 VMs and project C has 4 VMs
- The 2 VMs of project A are communicating with one Neutron vLAN with each other, and no port of it has a public IP address
- The 4 VMs of project C are communicating pairwise with 2 Neutron vLANs with each other, and no port of it has a public IP address
- The project A network and one of the two project C networks have the same private IP dress which does not disturb because they are fully isolated from each other
- All 3 private networks are routed to the Internet via 2 Neutron routers
- The Neutron routers perform NAT and translate the private-network IP-addresses into a public IP subnetwork address which is connected to a gateway
- The gateway in turn is connected to the Internet

6.13 The Keystone Service

- Keystone is normally the first service that must be installed for OpenStack because all other services depend on it for security reasons
- Beside its main task to authenticate users and to know their access rights, Keystone also maintains a list of all services which are installed in the cloud
  - Each service is registered with its name and the endpoints for internal, admin and public accesses (=service catalog)
- Last but not least, Keystone is responsible to secure all REST requests against misuse from hackers by issuing and validating user certificates that have to accompany each REST request
- Securing REST requests, implies also that Keystone can revoke a user certificate
- The user certificates used by Keystone are so-called Fernet tokens

Comment: See the chapter below about Fernet tokens.

6.13.1 Keystone Terminology

- Keystone has a user model for data privacy, authentication and authorization with own terminology that is as follows:
  - Endpoint: each service can be accessed by a URI that must contain a host-OS port-number
    - the URI with the port number is called endpoint because both together is enough data to make an API request, provided that a valid user token is presented to the callee
• **Token**: Tokens are data structures which contain among other fields an encrypted message and which are digitally signed
  - this is accomplished using the Fernet mechanism
  - Fernet tokens are generated for user authentication and authorization and are typically <250 bytes
  - since a token passes from every caller of a REST request to the callee and is subsequently validated by Keystone, REST requests with Fernet tokens are not very quick, but much faster than UUID or X.509 PKI certificates

Comment: *PKI* = Public Key Infrastructure.

• **Credential**: a credential is either a Fernet token or a username and password
  - a credential is associated either with a user or with user and project (=user-project pair)
• **Role**: metadata associated with user-project pairs for RBAC
• **Extras**: key-value metadata associated with a user-project pair
• **User**: has an account credential and at least one project credential
  - each user is associated with one or more projects or domains
• **Project**: organizational unit in OpenStack which contains one or more users
• **Domains**: organizational unit which contains users, groups and projects
• **Group**: is a collection of users which is associated with one or more projects or domains
• **Rule**: describes a set of requirements for triggering actions in Keystone
  - rules are stored in a JSON “policy file”
  - the Horizon configuration file contains the name and location of this policy file

### 6.13.2 Securing API Requests

- All API calls of OpenStack services are open to the world, even in private clouds, since the API network is typically a subnet of the Internet
- In order to prevent from global misuse, several mechanism are integrated in Keystone:
  1.) Every user must get a Fernet token from Keystone for authentication
  2.) The Fernet token implies the distinct access rights the user has in the cloud for each project according to RBAC
  3.) Every Fernet token expires quickly in typically 1 hour
  4.) Every REST request to any service must contain a valid Fernet token which is checked by Keystone before the REST request is entitled to get executed
  5.) Login into the cloud may be accomplished by a RSA key instead of username/password which is much safer
  6.) Several other measures which are not listed here

- All items together make an OpenStack cloud highly secure

### 6.13.3 Fernet Tokens

- The goal of data encryption is to encode data during transit and storage in order to maintain privacy
Typically, this is accomplished by a public and a private encryption key
With Fernet encryption, however, there is only a private key which encrypts and decrypts as well
This is why Fernet belongs to the class of symmetric encryptions
A Fernet token contains both, the private encryption/decryption key and a digital signature for the whole Fernet data structure
The goal of the digital signature in the Fernet token is to ensure that an encrypted message is from a proved issuer and not from a hacker
The goal of the complete Fernet token is to authenticate an REST request at the called service
Therefore, the message in a Fernet token contains authentication and authorization metadata of a cloud user
Keystone provides a repository for the Fernet key of each user and for every user-project pair
A Fernet key is the base64url encoding of the following two fields: Signing-key of 128 bits + Encryption-key of 128 bits

**Def.: Base64url encoding is the same as base64 encoding, but the signs + and / are replaced by - und _ and the = padding at the end is replaced by %3d.**

In total, a Fernet token is the base64url encoding of the following concatenated fields:
- Version, 8 bits
- Timestamp, 64 bits
- Initialization Vector, 128 bits
- encrypted message, variable length, multiple of 128 bits
- HMAC, 256 bits

**Comment: For HMAC and Initialization Vector, see below.**

The specification of a Fernet token is under: https://github.com/fernet/spec/blob/master/Spec.md

6.13.3.1 **The Algorithms in a Fernet Token**

Fernet is built on top of standard cryptographic algorithms:

1.) It uses for message encryption the Advanced Encryption Standard (AES) with a 128-bit key
   - AES encodes in the message individual blocks of 128 data bits each with the 128-bit encryption key part of the Fernet key

**Comment: AES is the US encryption standard.**

2.) It uses a hash-based message-authentication code (HMAC) with a key
   - the key is the signing key which is the 2nd part of the Fernet key

**Example: HMAC is also used in IPsec and TLS protocols.**
   - the hash function for HMAC is established by means of SHA256
· SHA256 is standardized a hash function that operates on 256-bit blocks, consisting of 128 bits of a data block and 128 bits of the signing key
· HMAC does not encrypt the message
· instead, the message (encrypted or not) must be sent together with the HMAC hash
· a receiver, that has the signing key, has to compute the hash value out of the encrypted data block again
· if the received hash value and computed hashes match then the data block is authentic
· the advantage of HMAC is that it decides about both, the authenticity of a data block and its integrity i.e. being not corrupted

3.) It uses urandom() for the initialization vector needed
Comment: In Linux, /dev/urandom is a pseudo-random number-generator.

4.) It uses the MessagePack format for the message

Def.: MessagePack is a compact format for binary data exchange and stands in contrast to JSON which is in ASCII and thus big in size (but human readable). Small integers are encoded together into a single byte, and a short string needs only one extra byte in addition to the string.

6.13.3.2 Creating a Fernet Token

The steps for creating a Fernet token are as follows:

1.) Bundle authentication and authorization metadata of a user in a short string as message
2.) Convert the message into MessagePack format
3.) Encrypt the MessagePack block-wise by means of the encryption key and AES
4.) Sign the result block-wise by means of the signing key and HMAC
5.) Create the Fernet token out of the encrypted data blocks, their signing and the other fields in the Fernet token

6.13.3.3 Using a Fernet Token

The steps for using a Fernet token are as follows:

1.) The caller of an API requests, i.e. an OpenStack user or an OpenStack service adds the Fernet token into the header of the REST request and sends the request
2.) Keystone accesses the Fernet key in the keystone repository and decodes the message
   · in case, that there are multiple keystone nodes, the repository must be shared between them
3.) Keystone checks the matching of the received and the computed signature
4.) If positive, the callee is allowed to execute the request and to answer with a REST response.

6.13.3.4 Checking the Fernet Token

- The Fernet token is processed by Keystone as follows:
  
  1.) It decodes the base64url encoding of the token
  2.) It tests the token version
  3.) It checks that the timestamp is not too far in the past
  4.) It computes the HMAC from the other token fields and from the signing-key
  5.) It checks that the computed HMAC matches the received HMAC field in the token
  6.) It decrypts the MessagePack using AES 128 in CBC mode with the initialization vector field in the token and the encryption-key
  7.) It unpads the decrypted MessagePack, i.e. it removes padding bytes
  8.) It undoes the MessagePack format, yielding the message in plain text
  9.) It checks whether the authentication and authorization plain text message matches access the roles and rules it has stored about every user or user/project pair
  10.) If positive, it grants the called service to execute the request

6.13.3.5 Example Interplay Between Caller, Callee and Horizon

- The use case is as follows:
  
  1.) A user logs in with a credential directly at Keystone via the proper REST request
  2.) He gets a Fernet token as response
  3.) He wants to create and start a VM in his project directly from Nova via REST
  4.) He gets a “created” response (status code 201)
- The time/space diagram of this use case is as below:

6.13.3.6 Advantages of Fernet Tokens

- Fernet Tokens have the following advantages:
  
  1.) They are at least 14 times shorter than X.509 PKI-based tokens
  2.) There is no need to store Fernet tokens permanently in the Keystone database
     - as long as each Keystone node shares the same key repository, an equivalent fernet token can be re-created everywhere in the cloud and anytime again
     - this stands in strong contrast to UUID tokens, because integral part of the UUID is time
and date of the creation
  · a Fernet token also has a time and date, but it is not part of the message and thus not important

⇒ only the private key of a user and of user and project pairs have to be stored permanently as a Fernet key of two subkeys
  · after expiration, a token can be deleted
  · this prevents the keystone database from overflow

6.13.3.7 Disadvantages of Fernet Tokens

- X.509 certificates have 2048 bits as signing key and are thus safer

6.13.4 Token Revocation

- Horizon can also revoke an issued token
- Furthermore, its API allows to list revoked tokens and to list those services that cache tokens
- Finally, it can ask services to remove revoked tokens from their cache
- Tokens are not deleted if revoked, instead they are added to a Keystone-internal list of revoked tokens
6.13.5 Keystone REST Request Examples

- As an example, we have the following use case:
  - An admin wants to login into Keystone without Horizon and to get a token back,
  - For that purpose, the following shell script could be used (at least in principle):
    ```bash
    curl -i \
    -H "Content-Type: application/json" \
    -d "
    { "auth": {
      "identity": {
        "methods": ["password"],
        "password": {
          "user": {
            "name": "admin",
            "domain": { "id": "default" },
            "password": "MySecretAdminPwd"
          }
        }
      }
    }
    "http://myCloudController.de:13000/v3/auth/tokens" ; echo
    ```
  - The script above assumes that Keystone is installed at a controller node named “myCloudController.de“ which is O.K.
  - Furthermore, it assumes for better illustration that the REST request and the token is transmitted in JSON format which is not O.K.
  - In fact, it is in base64url-encoded MessagePack format
  - As an other example, we have the 2nd use case:
  - An admin who has 999888777666 as own certificate wants to ask Keystone to validate a user certificate which is 887665443383838
  - The curl command below could do that (at least in principle):
    ```bash
    curl -H "X-Auth-Token:999888777666" "http://localhost:35357/v2.0/tokens/887665443383838"
    ```
  - Comment: For admins, there is an own port number which is 35357.
  - Comment: The tokens used in the examples are purely symbolic. Fernet tokens have up to 250 bytes.

- For better illustration, the REST response from Keystone is again a JSON object that contains the following information about a user of name "Antony ExampleUser":
  ```json
  { "access": {
      "token": {
        "expires": "2018-02-05T00:00:00",
        "id": "887665443383838",
        "tenant": {
          "id": "1",
          "name": "My great Foobar Project"
        }
      }
    }
  ```
Comment: The term “Foobar” is widely used in informatics. It acts a placeholder for any possible string. It means “this and that which I do not know yet”. In German “Dingsbums”.

- The JSON response does not only validate the token but tells the admin also that Mr. Antony ExampleUser is member of "My great Foobar Project"

6.13.6 Internal Keystone Setup

- Keystone comprises one server, many modules and a few drivers
- The central server provides authentication and authorization (A&A)

- There are about 24 modules
  
  Example: One module is an API manager that extracts the Fernet token from the REST header and sends it to the central server for A&A. An other module provides the service catalog.

- Modules are connected to the Keystone server by the Python “Web Server Gateway Interface (WSGI)“

  Comment: WSGI is a standard interface between a web server and a web application.

- Drivers are used for accessing information in external repositories

  Example: There is a driver for the Keystone SQL database and for an LDAP database

  Comment: LDAP = Light-weight Directory Access Protocol. LDAP is standard software for identity systems that provides for authentication of users and their authorization.

6.14 The Cinder Service

- Cinder is the block storage service of Openstack and has five tasks:

  1.) It allows to create a virtual hard drive that is block-oriented and delivers permanent storage
      - permanent storage is possible because it is implemented by physical storage subsystems comprising storage servers and disks
  
    2.) It can attach this virtual hard drive to a VM
the Cinder virtual hard drive is an additional storage to the VM, beside the virtual hard drive KVM/QEMU has created
- the Cinder virtual hard drive remains after the VM is terminated

3.) It allows to create a snapshot of a virtual hard drive

4.) It can make a backup of a virtual hard drive or of a group of them

5.) It can encrypt a virtual hard drive

Comment: All tasks will be explained later.

6.14.1 Cinder Terminology

Def.: Distributed storage: a virtual hard drive or file system that is distributed over some or many physical hard drives. A distributed storage is established via a storage protocol.

Def.: Storage server: a physical computer with at least one physical hard drive.

Def.: Storage protocol: a storage protocol controls one or more physical hard drives either directly or via a storage server. A storage protocol implements distributed storage.

Def.: Volume: a virtual hard drive Cinder has created.

Def.: Volume type: a symbolic name a user or admin can invent. Each volume type is implemented by a specific Cinder volume driver. More virtual hard drives with the same implementation can be easily created by specifying the same volume type.

Def.: Snapshot: a copy of a volume at a specific point in time.

Def.: Cinder driver = storage backend driver: a Python proxy for a storage back end. Several dozens of Cinder drivers exist.

Def.: Storage back end = storage protocol: the protocol that a Cinder driver uses for persistent storage, such as iSCSI, NFS or GlusterFS.

6.14.2 Cinder's External Setup

- Cinder acts upon a REST request from Nova
- Nova in turn acts upon a REST request from Horizon
- Horizon, finally, acts upon a user input into its GUI or CLI
- Cinder communicates in the cloud with Nova and Keystone
- It communicates externally via one or more storage protocols with physical hard drives or with computers containing physical hard drives
- The physical hard drives are controlled by local or remote host-OS device-drivers residing in specialized storage servers
- In a data and in a computing center, multiple physical disks are coupled either as a Storage Area Network (SAN) or as a Network Attached Storage (NAS)
- The default storage protocol for Cinder is LVM for a local volume group named “cinder-volumes”
Def.: LVM is part of the Linux kernel, and it allows to create a logical volume that spans across multiple physical disks. The size of a logical volume can be extended even after volume creation.

- Also the size of a Cinder volume can be extended after creation

- Since the KVM/QEMU virtual volumes inside of VMs are also block devices, Cinder can also manage them, although being not persistent
- Cinder supports RBAC for its volumes
- For projects, quota controls check default limits automatically:
  - The number of volumes that can be created
  - The number of snapshots that can be created
  - The total number of GBytes allowed
    - this quota is shared between snapshots and volumes
6.14.3 Cinder's Internal Setup

- Cinder can be ran on one or more nodes to increase throughput
- It consists of:
  - SQL database which is shared by all Cinder components of a node
    - Cinder nodes in different cells can have different SQL databases
  - API component: receives http requests, converts and forwards them to other Cinder components via oslo.messaging
    - it is implemented as a Web Service Gateway Interface (WSGI)
  - Scheduler component: decides which local or remote storage server gets which volume
    - scheduling is either via round-robin or via “filtering out“ a proper storage server according to capacity, availability zone, volume types and capabilities, for example
  - Volume component: manages Cinder volumes, including encryption
  - Backup component: manages backups of Cinder volumes
  - Storage Backend Drivers: python proxies to non-Python storage protocols
- Each Cinder component is an own process

6.14.4 Cinder Virtual Hard Drive

- Cinder establishes an abstract API for the creation and management of a virtual hard drive that is independent of the concrete storage protocol, of SAN/NAS, of device driver and physical disk, and of any other hardware and software details
- This is remarkable because there are many storage protocols, device drivers and physical devices with highly varying properties
- This is accomplished by means of the Cinder volume component and its storage backend drivers
- Cinder’s abstract API does not establish a distributed and reliable file system, neither allows it to read from the volume or to write to it
- If the chosen storage protocol is a distributed and reliable file system, such as Ceph for example, the created volume will be also distributed and reliable without notice to the user
- A typical Cinder volume has 1 TB, while a typical KVM/QEMU virtual hard drive has about 100 GB
Comment: KVM/QEMU virtual hard drives are small in order not to overcommit the local disk.

- The Cinder API supports also volume groups by means of Horizon's GUI
- This allows to handle sets of volumes collectively in the same way as a single volume, including snapshots and backups, which is very convenient for admins
- Furthermore, for each volume a Storage Quality-of-Service can be specified (=Storage QoS) with respect to data throughput
- Cinder allows to specify:
  - The maximum bytes per second (=upper limit)
  - The maximum bytes per second per GByte of Cinder volume capacity
  - The maximum Input/Output operations per second allowed
  - The maximum Input/Output operations per second per GByte of Cinder volume capacity
- Additionally, a Cinder volume can boot a VM because of its persistence
- Finally, a volume can be addressed uniquely even in a large cloud by a volume ID which is a UUID that is assigned by Cinder at volume creation time
- Alternatively, it can also be accessed by a human-readable name that may be not unique
- It is also possible to move a volume from one storage backend to another, in case of decommissioning a physical disk for example
- However, this implies a real transfer of data because the physical storage is changed

Cinder can also import non-Cinder storage and add metadata for it in its database so that it becomes a Cinder volume

- Furthermore, for a given volume type, a collection of criteria can be specified as a list of keyword/value pairs
- These keyword/value pairs are inspected by the Cinder scheduler when it decides which Cinder backend(s) can implement a specific volume type
- This implies that Cinder drivers inform the scheduler about the physical capabilities of their disk drives in form of keyword/value pairs
- Both sets of keyword/value pairs can then be easily matched by the scheduler

### 6.14.4.1 Attachment of a Virtual Hard Drive to a VM

- Each Cinder virtual hard drive can be attached to only one VM
  - It is not possible to share data in Cinder

Comment: Sharing files is accomplished by the Manila service.

- However, a Cinder virtual hard drive can be detached from one VM and re-attached to another VM without data loss
- This allows for high-speed data-transfer between VMs, because no data is transferred physically
- By the REST API, it is also possible to transfer a volume from one user to another, with no data physically transferred
6.14.4.2 Accessing Records in Cinder Volumes

- If a VM wants to access a record in a created Cinder volume then the first step is to attach the volume to the VM
  
  **Example:** A volume would be attached as `/dev/sdc1` in Linux if iSCSI is used as storage protocol. `sd` stands for SCSI device.

- The 2nd step is to format and partition the volume if not yet done
- The formatting and partitioning is accomplished in the guest OS of the VM
  
  **Example:** Use `cfdisk` for a Linux host OS.

- The 3rd step is to create a file system on the volume if not yet done
  
  **Example:** Use `mkfs` in Linux. Numerous file systems are possible such as `ext4` or `ntfs`.

- The fourth step is to mount the volume to a directory
  
  **Example:** Mount `/dev/disk/by-uuid/<volume ID>` `/myDirectory`

- Then you can access the volume in the language of your choice
  
  **Example:** In C use `open()` in `/myDirectory` to create a new file and `readf()` and `printf()` to read and write records for the new file as for any physical file.

6.14.5 Cinder Snapshot

- A snapshot is a read-only copy of a Cinder volume at a given point-in-time
- It is irrelevant whether the volume is attached to a VM or not
- A snapshot can serve as the data source for a new Cinder volume by setting the “source-volid” keyword in the extra settings of the new volume

6.14.6 Cinder Backup

- Cinder offers self-service backup and restore for projects
- Self-service means that no admin is needed
- In contrast to a snapshot, a backup is not only a copy of a volume, but it contains also metadata to restore the contents of the copied volume to any other volume of enough space, even if it is of another type
- Also in contrast to snapshots, backups are stored in a dedicated repository which is implemented not by Cinder but by Swift

6.14.7 Cinder Volume Encryption

- For a volume type, one can specify encryption by a set of keyword/value pairs
  
  **Example:** For keyword/value pairs: `cipher='aes-xts-plain64', key_size='256'`. 

Comment: Beside aes-xts-plain64 also aes-cbc-essiv is supported; Beside key_size='256' also 512 is supported. Please note, setting the keyword/value pair to 512 results in a 256 bits encryption key, while 256 will become a 128 bits encryption key.

- Non-admin users need the creator role to store the encryption key in the Barbican service and to create encrypted volumes

Comment: Barbican is the key-storing service of OpenStack.

- The data encryption can be performed either by Nova with control-location=front-end or by Cinder with control-location=back-end
- If Nova encrypts the volume then any data from or to the volume that is travelling over a Neutron network is already encrypted
- If Cinder encrypts then the volume only the data on the volume itself are encrypted

6.15 The Swift Service

- Swift is a complex OpenStack service with an easy API to the user
- Swift can store permanently and reliably big sets of data, together with their metadata
- Typically, Swift manages TBs to PBs of disk space

Comment: TB = Terabyte, PB = Petabyte.

- It can thus also be used for backup and archiving big data
- Swift's capacity scales arbitrarily by adding new Swift nodes

- In contrast to KVM/QEMU and Cinder blockstore, data is accessible from anywhere in the Internet, not only from VMs
- By its API, Swift can be called directly from applications in projects (tenants)
- Also in contrast to Cinder, the Swift API is tiny and allows also for reading and writing as well
- Only 16 REST requests are needed for creation, reading, updating and deletion of objects and their metadata and for object management
- Because there exists no standard for object reading and writing, the Swift API does not compete with anything
- Additionally, Swift ensures high reliability in data storage because of two methods:
  - a user-defined number of data replicas (copies) which are placed on disjoint storage nodes
    - The default value of the number of replicas is three which means one original and two copies
  - an “erasure code technique“ which is used to restore lost parts of objects
- Normally, only one of the two methods is chosen by the user in a “storage policy file“, but it is possible to combine both methods for extreme reliability
- Replica generation and erasure code technique are fully transparent to the user

Comment: Transparent means that after having edited the storage policy file, the user is not engaged any more.
- Erasure code is much more efficient with respect to required disk space than replica generation
Comment: See subsequent chapters about replicas and erasure code.

- Finally, a normal storage object cannot be larger than 5 Gbyte by default
- However, an object of arbitrary size can be composed out of any number of 5 Gbyte objects (or of any number of smaller objects)
- The result is called “manifest object“
- Swift can handle an “infinite“ amount of normal and manifest objects in the same cloud
- This number is only limited by free disk space but not by Swift

6.15.1 Swift Terminology

Def.: Swift account: is in principle a project (tenant) but with a restricted scope. It shows only Swift container and metadata of them. Do not confuse a Swift account with a user account in Keystone.

Def.: Swift container: organizes and stores objects in Swift. It is similar to a OS directory but cannot be nested. It has nothing to do with a LXC container.

Def.: Swift object: a large binary sequence of data (blob) and metadata for it.
Comment: Blob = binary large object.

Def.: Swift object expiration: a configurable option to automatically delete objects after a specified amount of time has passed or a certain date is reached.

Def.: Swift object hash: unique ID for a Swift object.

Def.: Swift object path hash: unique ID for the location of an object in the so-called Swift ring. The path hash maps objects to physical disks

Def.: Swift object versioning: a flag for an Object Storage container to version all objects within the container.

6.15.2 Object Replication

- If object replication is chosen by the user as redundancy method then replicas of the storage object are automatically created and distributed in the cloud
- This is accomplished decentrally by the local object replicator in that node where the original object is stored
- The object replicator pushes data from local records and files to remote disks
- If a replicator detects that a remote node has failed, then it chooses an alternative node
- In addition to object replicators, there are database replicators, which replicate account and container databases
- Replica placement on remote nodes is handled by a “hash ring“
Comment: See subsequent chapter about hash ring.

6.15.3 Asynchronous Eventual Consistency for Replicas

- In Swift, the method of asynchronous eventual consistency is used for replicas
In general, eventual consistency is used in distributed computing to achieve high availability by creating and comparing replicas out of original data.

In Eventual Consistency, sooner or later all replicas will have the same value as the original, provided that no update is made to the original in the meantime.

A system that has achieved consistency between original and copies is called replica converged.

Eventual consistency is a weak guarantee because it may be difficult to determine the state of the system.

However, retrieval of a replica is possible only after replica convergence is obtained in order to get response values that are consistent with the original data.

Asynchronous eventual consistency means that the writing of the original data can already terminate while not all replicas are fully written.

### 6.15.4 Erasure Codes

Erasure codes are an error correction method for the case that some parts of a message are lost during transmission, rather than parts are received wrongly.

A message of n symbols is expanded to k>n symbols by erasure code protection.

If n ≤ l < k symbols remain after data loss then the message can still be fully restored.

Optimal erasure codes have the property that any combination of l=n symbols out of k sent symbols are sufficient to fully restore the original message.

*Example:* Reed–Solomon codes were the first optimal erasure codes.

However, this comes along with a high computational complexity of O(n²), or at least O(nlogn).

Near-optimal erasure codes require any l>n symbols to recover the message but have linear time complexity O(n) only.

#### 6.15.4.1 Swift Use Case for Erasure Codes

If not data transmission but data storage is the use case, then a storage object is decomposed into n fragments which are erasure-code expanded to k>n fragments that are then distributed to multiple storage nodes.

Even if some storage nodes completely fail then the object can still be recovered.

Because of the lower disk space consumption, erasure code protection is preferred to replicas.

If erasure code protection is chosen as redundancy method then each storage object is automatically split into erasure-code protected-fragments and distributed by means of a hash ring.

For speed reasons, the core of erasure-code protection is performed in an external C library and not in Python.

### 6.15.5 Swift Data Hierarchy

The Swift data hierarchy consists of a Swift account that lists all containers of a project, of containers that store all objects, and of objects that store all data.
Every object reflects this three-level hierarchy in its REST URI:
https://<mySwiftNode>:13808/v1/<account ID>/<container name>/<object path and object name>

Comment: 13808 is the public swift port.

6.15.5.1 Swift Accounts

- A Swift account defines a namespace i.e. a scope for container names
- Any number of containers can be defined within an Swift account
- In an account, one can list, create and delete containers and create, update, show and delete account metadata
- Account metadata are e.g.:
  - number of containers
  - number of objects
  - number of bytes that are stored in the account
- Access to an account is normally regulated by RBAC
- However, a user with a proper role in RBAC can allocate a Access Control List (ACL) to the Swift account that bypasses the keystone RBAC mechanism
- Such an ACL may grant access rights for a user to a Swift account which are not checked by Keystone
- Account ACLs are stored in the account as metadata

6.15.5.2 Swift Containers

- A container defines a namespace for object names
- In a container, one can list objects, create, update, show and delete them and do the same for container metadata
- Similar to an account ACL, also container ACLs are possible
  - Their scope is the container where the ACL metadata is stored and pertains to all container objects
- Optional object versioning also takes place at the container level of the Swift hierarchy

6.15.5.3 Swift Objects

- All types of data can be stored in a Swift object, also those with block orientation
- Because of that, Swift would be in principle a competitor to Cinder and Glance
- However, it is not recommended to have Swift as store for a relational database, for example, because SQL databases are speed-optimized for block devices only
- It is also not recommended to store images directly in Swift without Glance because the latter provides, for example, also an image catalog
- Optional scheduled object-deletion takes place at this hierarchy layer
- In case of download with GET, an auto-extract of archive files is accomplished as well
- Object ACLs do not exist, but a time-limited GET access to an object is possible
- The number of objects is unlimited
6.15.6 Internal Swift Setup

- Swift consists of multiple components, and each of it runs as process in a host OS
- Because of the three-level hierarchy of Swift, also its internal set-up is structured this way

6.15.6.1 Account Components

- account server: a Swift component that allows to list, create, update and delete containers and their metadata
  - Stores container information in the account database
- account databases: multiple SQLite databases that contain Swift accounts and account metadata
- account auditor: checks for missing replicas and incorrect or corrupted objects in a specified Swift account by queries to the account databases
- account replicator: copies account databases
- account reaper: a Swift process that scans for and deletes account databases the account server has marked for deletion

6.15.6.2 Container Components

- container server: a Swift component that allows to list, create, update and delete objects and their metadata
- container databases: multiple SQLite databases that store containers and container metadata
  - The container server accesses these databases
- container auditor: Checks for missing replicas/erasure code fragments or incorrect objects in the container
  - It works via queries to the SQL databases of Swift
- container replicator: copies container databases

6.15.6.3 Object Components

- object server: a Swift component that is responsible for storing, retrieving and deletion of objects on local disks
- object auditor: reads objects for an object server and verifies that an MD5 hash, size, and metadata are consistent for each object

Comment: MD5 is a method to create a hash function. It can serve, for example, to check whether a download was presumably correct.

- object replicator: copies objects in case of replica policy
- object reconstructor: performs erasure code protection in case of erasure code policy

6.15.6.4 Proxy Server

- There is at least one proxy server in Swift that looks-up the location of account, container and objects and routes the request to the proper local servers
- The public Swift API is also managed by the proxy server
In case of erasure code policy, the proxy server is additionally responsible for encoding, decoding and streaming object data for upload and download.

Multiple Swift proxy servers should exist in each availability zone to increase throughput and reliability.

Swift proxies in each availability zone should first write to and read from local disks in the own availability zone before streaming to and from other zones.

6.15.7 Functional Concept of Swift

Swift is a distributed architecture with no central point of control and thus also with no central point of failure.

Should one or more storage nodes fail, Swift restores their content either by replicas or by the erasure code technique, depending which policy was specified by the user.

Retrieving of stored objects is based on hash rings which are available on each storage node.

6.15.7.1 Manifest Objects

A manifest object consists of logically concatenated objects as segments.
It contains by itself no object data but metainformation about their storage.
There are two types of manifest objects: static and dynamic.
A static manifest objects does not allow to add or remove segments after creation.
A dynamic manifest allows that functionality.

Static manifest objects are extra secured by a MD5 checksum.
As a consequence, data integrity can be assured.
This is not so easy with dynamic manifest objects because they can change.
Additionally, a static manifest object contains an ordered list of the names of the segment objects in JSON format.
A dynamic manifest object contains the name of the container where the segment objects are stored.
For a dynamic manifest object, all segments must be stored in the same container.
Swift concatenates automatically all segments for the GET request to a manifest object.
There is no limit in the number of segments.
The “Content-Length” field in the response header of the GET request shows the accumulated segment sizes.
Please note: the upload of manifest objects to Swift and the download from Swift can take a long time if segments are large.
Because of that and the eventual consistency of replicas, an uploaded segment might not appear in the container list until somewhen later.

6.15.7.2 Swift Proxies

Swift frequently communicates among servers that are hosting objects.
Even a small cloud generates megabytes per second of traffic caused by Swift.
- Internal communication is unencrypted, so private networks are required
- The swift-proxy is stateless, which means that one can easily add more services
- More proxies mean more throughput and reliability
- Between swift-proxies, HTTP load-balancing methods are possible to share bandwidth and availability between them

6.15.7.3 Replica Policy

- One should configure Swift with a sufficient number of availability zones to provide reliably the necessary quorum for the number of replicas/erasure code fragments
- Only if the necessary quorum is available a successful REST response is possible
- For example, with three replicas configured, the recommended number of availability zones is five
- Sooner or later, all replicas will have the same content as the original data

6.15.7.4 Erasure Code Policy

- If erasure code policy is enabled, data upload to Swift is made via PUT and has the following steps:
  1.) The proxy server streams in a part of an object, whose size is user-configurable, and buffers that part
  2.) The proxy server calls the external library to split and encode that part into smaller erasure code fragments
  3.) The proxy streams out the erasure code fragments to storage nodes with the help of the hash rings
  4.) The proxy repeats until the object is processed
  5.) The client is notified of completion when a quorum of $l$ nodes is met
- Data download to the client is performed by GET and looks like this:
  1.) The proxy server makes simultaneous requests to the $k$ nodes of the object by means of the hash ring
  2.) As soon as the proxy has the $l$ fragments it needs, it calls the object reconstructor which calls in turn the external library to decode and compose the object part
  3.) The proxy streams the object part back to the client
  4.) It repeats until the object is fully retrieved

6.15.7.5 Load Balancing

- Load-balancing for Swift projects and user applications is accomplished by means of proxy services that are distributed across multiple availability zones
- A HAProxy is one method of providing load balancing and high availability
Comment: A HAProxy is open source software that provides a high availability load balancer and proxy server for TCP and HTTP-based applications that spreads requests across multiple servers.

6.15.7.6 Hash Ring

- Object distribution and retrieving is accomplished by means of hash rings for both data protection methods
- A hash ring is a data structure for the mapping of a subset of stored Swift items onto their physical location
- Therefore, the hash ring is also called path hash map
- There are at least three hash rings, a first for mapping accounts, a second for mapping containers, and a third for mapping objects to physical disks
- Each hash ring is composed of 2 tables and one integer
- Typically, a hash ring is so big that it must be stored in a file
- If both data protection methods are engaged, or if multiple storage policies are defined, more hash rings are needed for mapping objects
- The 1st table in the hash ring is the "replica-to-partition-to-device table"
- The 2nd table in the hash ring stores mainly the socket addresses under which each local swift server that manages a disk is listening

Comment: A socket address consists of IP address and port number.

- The integer in the hash ring contains the number p of bits each “ring partition number“ has

6.15.7.7 Ring Partition Number

- The ring partition number is computed as follows by a Swift Proxy:
  1.) The URI of the object that has to be stored or retrieved is hashed by MD5
  2.) The p last significant bits (LSBs) as indicated by the hash ring integer are taken from the URI
      • selecting the p LSBs is a 2nd hashing procedure
  3.) This is the ring partition number

6.15.7.8 Replica-to-Partition-to-Device Table

- The replica-to-partition-to-device table is a 2D-table with row and column indices
- In case of a replica-object hash-ring, the row index is the replica number i of an object
  • Typical is: 0≤i≤2
- In case of an erasure-code object hash-ring, the row index is the fragment number of the erasure-coded object-part
- The column index is the ring partition number
- Each element in the table is the so-called “disk number“ where to store or retrieve the ring partition

Comment: For an explanation of disk number see next subchapter .

- The name “ring“ results from the way how disk numbers are entered into the table
The first table element of every row is the successor disk number of the previous first-element disk-number
All following elements in every row are the next disk numbers modulo the total number of disks
Mathematically spoken, the table defines a cyclic shift permutation between modulo 4 elements
The cyclic shift is the “ring“ in the natural sequence of disk numbers

Example: Given are one original object and two replicas \((0 \leq i \leq 2)\) and 4 disks \((0-3)\). Then the disk numbers in the 1st table are as follows:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 0 \\
2 & 3 & 0 & 1 \\
\end{array}
\]

The permutation described provides for an equal distribution of replicas/erasure code fragments across disks
It provides also that every 2nd Swift partition has a different subset of disks which is good for reliability in case of disk failure
Other permutation functions are possible as well

6.15.7.9 2nd Table In the Hash Ring
The 2nd table in the hash ring is a 1D table where the table index is the disk number, while the element in each row is the socket address, together with some metadata for load balancing
The 2nd table allows to access the proper Swift server via the Internet

6.15.7.10 The Swift-Ring-Builder
Technically, each hash-ring data-structure is composed by a tool called swift-ring-builder
The swift-ring-builder is also responsible for the allocation of disk numbers to physical disks which is not arbitrarily
For this numbering, the swift-ring-builder takes into account the regions the cloud has, the availability zones in each region, the partitions in each availability zone, the servers in each partition, and finally the specified storage policies
As with the rings, there is at least one builder for objects, containers and accounts, respectively

6.16 The Glance Service
Glance lists, uploads, downloads, updates, deletes, suspends and resumes images and manage image metadata
Example: For image metadata are the image ID and size, the image status and the members that share an image.

- Additionally, Glance can import and export an image from/to other OpenStack clouds via the Internet
  - Other images can be in another format, such as qcow2, for example, and are converted to the own storage back end format
- Finally, Glance can create a JSON schema for an image and for task metadata

Comment: See chapter below about Glance tasks.

- All operations are controlled by RBAC
- Glance images are used by Nova to start a VM and by Cinder to create a bootable disk
- Multiple Glance services can be run for scalability and reliability

### 6.16.1 Glance Internal Setup

- Glance consists of the following processes:
  - **API component**: accepts REST requests and forwards them to the proper Glance component
  - **Database component**: stores and retrieves metadata
  - **Image data backend**: stores the image
  - **Metadata definition component**: allows the user to add metadata as keyword/value strings
  - **Replication component**: copies images for other Glance instances and ensure their consistency
  - **auditor, updater, and reaper components**: check images for consistency, updates images and deletes images
  - **Task executor component**: processes Glance tasks

### 6.16.2 Glance Tasks

- Tasks are long running operations such as image export/import, image consistency check, image upload/download or image format conversion
- They can be run as a Glance batch job in parallel to REST requests from Nova and Cinder but decoupled from them
- A task is a JSON data structure defined by a JSON schema in the API documentation
- The JSON task structure contains as keywords some CLI commands of Glance such as `import_from` or `import_from_format`
- A task has also metadata such as `created_at`, `expires_at` and pending, processing", success, and failure

### 6.17 The Trove Service

- The Trove service manages various **Database Management Software** (DBMS) and the **datastores** created out of them
6.17.1 Trove Terminology

Def.: Database Management Software (DBMS): also known as Database Engine. This is a commercial or OpenSource software for creating and managing a database. There are DBMSes for relational and for non-relational databases.

Def.: Database: a set of files managed by a DBMS that contain user data.

Def.: Database VM: a compute VM that is specialized by a DBMS and that contains or will contain a database with data.

Def.: A Trove datastore is a database VM with Trove guest-agent, a guest-agent configuration-file, a DBMS and the database data.

Comment: For Trove guest-agent, a Trove guest-agent configuration-file, see later chapters.

6.17.2 Classification of Trove Datastores

- There are 3 classes of Trove Datastores:
  - Experimental: this class of DBMSes have only a basic functionality in Trove
  - Technical preview: this class of DBMSes have nearly the full functionality in Trove
  - Stable: this is the class of production-ready DBMS in Trove with all functions they are capable of

- Only mysql is classified as stable DBMS in Trove
- Cassandra and MongoDB are classified as technical preview

All other DBMS are classified as experimental

6.17.3 Storage Locations

- Database VM images and image templates are stored in Glance
- Backups of database VMs are stored in Swift
- Database data are stored in the file system of the datastore

6.17.4 Trove Configuration Groups

- One can configure a whole set of databases of the same type in the same way by using a configuration group
- A. configuration group allows to set configuration parameters group-wise

6.17.5 Trove Internal Setup

- Trove has the subsequent components:
  - API component: runs on a specialized Trove node or in the controller node
    - it is implemented as a WSGI Router and forwards the REST request to the proper Trove component

Def.: A WSGI Router is a WSGI-based application containing a list of URI routes. It takes the requested URI and decides which component will handle the request. It parses the URI and extracts out of it the input parameters for the target component.
Taskmanager: runs on a specialized Trove node or in the controller node
- creates and launches a compute VM by means of Nova for a later installation of a DBMS on it
- launches by means of Nova an already created database VM image
- performs operations on the database VM after it is specialized
- manages the lifecycle of the database VM

Guest-agent: runs on each datastore VM
- contains a datastore manager as main component
  - this is the OpenStack driver for the DBMS
  - for each supported DBMS, an own datastore manager exists and must be installed
- operations on the datastore’s data are performed by SQL or by the CLI of the DBMS not by the guest-agent

Conductor: runs on a specialized Trove node or in the controller node
- receives messages via oslo.messaging from the datastore VM to update status information about the VM in the infrastructure database
  Example: For datastore VM status: NEW, BUILDING or ACTIVE.
  - provides a “heartbeat” method that updates periodically the status of the datastore VM in the infrastructure database
  - provides an “update_backup” method that updates periodically the status of a backup of the datastore (not the VM), including its metadata in the infrastructure database
  Example: For metadata of a backup: size, type and checksum.

Infrastructure database: runs on a specialized Trove node or in the controller node
- contains a catalog of all DB VMs
- holds encryption keys for Inter-component communication
- holds metadata about status
6.17.6 Database-VM Minimum-Requirements

- The VM specialization requires that the VM's flavor matches the minimum requirements of the respective DBMS with respect to main memory, disk space and number of cores
- Each database VM must contain as minimum software:
  - a Trove guest-agent, a Trove guest-agent configuration-file, a DBMS, and sooner or later also the database itself with data
  - Since each database VM is a specialized compute VM, it contains also a host OS with KVM/QEMU, Neutron-Linux Bridge and Nova-Compute

6.17.7 Trove Guest-Agent Configuration

- It contains configuration data for
  - datastore manager
  - datastore-specific options for every supported DBMS such as mysql, percona etc.
  - options for backups in Swift
  - options for oslo.messaging
  - log files configuration
  - adding the "root" user with access rights to the DBMS
    - access rights are called "grants"
- The access rights of the root do not comply with the RBAC system

6.17.8 Save Inter-Component Communication

- As for any service-internal communication, oslo.messaging is used between the components
- However, Trove allows to encrypt its inter-component communication by means of oslo_messaging.rpc security
- There is an encryption key for the communication between the API component and the taskmanager
- There are individual keys for each DB VM for the communication between them and the taskmanager

6.17.9 Managing DBMSes and their Datastores

- By means of Nova, Trove allows to create a VM with a specific flavor and to launch it as database VM
  - Subsequently, if a specific database software is installed in a VM, it can provide for multiple datastores of the same type in the same VM
- User and admins can install the needed DBMS on the VM to create a database VM
- Users and admins can create as many database VMs as needed
- In a database VM, the following DBMS are more or less supported by Trove and can be installed:
  - cassandra, couchbase, couchdb, db2, mariadb, mongodb, mysql, percona, postgresql, pxc, redis, vertica
Several different DBMSes can be installed in the same database VM
Each database VM with a specific DBMS (or more of them) is stored as template in Glance for later launches of database VMs of the same type
Each template database VM image in Glance has a name and an ID Trove can refer to for launching
Trove datastores are not shared between VMs, but multiple users and projects can access the same datastore

6.17.10 The Trove API

The Trove API is always the same, regardless whether the DB is SQL-based or not and whether the datastore and its database software is localized on one node or distributed
The Trove API allows the following functions:
- Create and manage database VMs
- Create and manage databases
- Create and manage datastores
- Create and manage replicas of database VMs
- Create and manage backups of database VMs
- Create and manage so-called “configuration groups“

Def.: A Trove configuration group is a set of datastores that are managed all together in the same way.

6.17.10.1 Create and Manage Database VMs

Trove allows for a database VM to create it, to list all database VMs, to attach the database VM to a configuration group, to update its name and version, to delete it and to list its configuration
Additionally, Trove can list the flavor of the database VM, resize the VM and restart it and manage log files

6.17.10.2 Create and Manage Databases

It allows for a database to create it, to list all of them with details, to enable, disable, publish them
Furthermore, Trove can add a root and normal users to the database

6.17.10.3 Manage Datastores

It is possible to list all datastores and to show their configuration

6.17.10.4 Create and Manage Replicas of Database VMs

It is avoided to make changes in a production datastore because this implies unwanted down time of the store
Instead, a replica (copy) of the datastore is made and changes are accomplished there
If everything works fine after the change, then the replica is declared to become the new datastore and replaces the old one
This reduces the downtime of a datastore because of changes to zero

6.17.10.5 Create and Manage Backups of Database VMs

The API allows also to create, list and delete backups of database VMs

6.17.10.6 Create and Manage Configuration Groups

Trove allows to create, list, delete and update configuration groups

6.18 The Ceilometer Service

Ceilometer provides for the collection of OpenStack-internal usage-data

Example: Usage-data are VM uptime, elapsed CPU time, consumed disk space, performed number of IO operations per VM and day etc.

Additionally, Ceilometer provides for the collection and storage of OpenStack-internal events

Finally, Ceilometer provides for the collection and storage of OpenStack-internal alarms

A Ceilometer component can be run in every compute node

Additionally, there is a component that is located in the controller node of the cloud

Ceilometer is not a typical OpenStack service because it does not have an API

Instead, Ceilometer communicates with fully-fletched OpenStack services via notifications that are sent via oslo.messaging

6.18.1 Ceilometer Terminology

Def.: Metering is collecting information about resource consumption with respect to resource type, consumption points-in-time and customer related to the consumption. The result of metering is a set of “tickets” a.k.a. samples.

Comment: A.k.a. = also known as.

Def.: A meter is a counter variable that is defined by a user or admin. It tracks a resource to collect consumption data by counting accesses to the resource. A resource can have multiple meters.

Example: For resources being metered: VMs, Swift or Cinder volumes, Glance images, Neutron networks and switches.

Def.: An Event is a state-change of a resource when something interesting has occurred. Each event is a data structure sent via oslo-messaging and comprising an event_type string, a message_id, a timestamp and a sequence of key-value pairs that describe the event’s traits.

Example: The flavor of a VM or its public IP address has changed.

Def.: Alarms allow to automatically monitor OpenStack resources and to react in case of malfunctioning. A user or admin can define a set of Ceilometer rules that create an alarm
action if they are broken. An alarm rule is a set of meters or events whose values are combined by AND and OR. Alarm rules are evaluated periodically every minute. Alarm actions are HTTP POST requests being sent to an endpoint, together with a description in JSON key/value pairs.

Example: The following Ceilometer CLI command sends a POST request to the URI http://example.org/notify if at least one of two other alarms were risen.

```bash
ceilometer alarm-combination-create --name meta \
  --alarm_ids ALARM_ID1 \
  --alarm_ids ALARM_ID2 \
  --operator or \
  --alarm-action 'http://example.org/notify'
```

### 6.18.2 Types of Meters

- Three types of meters exist:

  1.) Cumulative: the meter value increases over time
  
  Example: The number of performed disk I/O operations in a VM during a day.

  2.) Gauge: meter values that are either constant or fluctuating
  
  Example: For constant meter values: number of public IP addresses a VM has.

  Example: For fluctuating meter values: the number of Swift objects which are storing collected consumption data, or Glance image launches per day.

  3.) Delta: Incremental meter value change per time unit

  Example: Used bandwidth which is typically growing or decreasing over time.

### 6.18.3 Ceilometer Components

- Ceilometer has the following types of components:

  1.) polling agent:

  • polls an OpenStack API or an OpenStack tool at a regular time interval
  • it can be configured to run either on any compute node or on the central controller node(s)
  • any compute node can host a polling agent as decentral metering process
  • the controller node hosts a properly configured polling agent as metering process for resources that are not bound to a compute VM

  Example: Swift objects or Heat scripts can exist independantly of compute VMs.

  Comment: For Heat scripts see the next main chapter.

  2.) notification agent:

  • runs on the central controller node (or the central controller nodes)
  • listens to notifications sent via oslo.messaging from ceilometer-polling agents
  • sorts event by their type
  • composes events and metering values to data structures which are called Ceilometer samples
• forwards the data structures to Gnocchi or other OpenStack services via oslo.messaging
• this is the preferred method of data collection

3.) push agent:
• instruments an OpenStack service by adding push agent code to it
• is the only solution for services which do not expose the required data via an API or via oslo.messaging
• This is not the preferred data collection method

- Multiple Ceilometer agents can run in parallel on compute nodes and controller node(s) to scale Ceilometer from small to large clouds

6.18.4 Ceilometer Operations

- Ceilometer operations are collecting usage data, events and alarms and forwarding them to so-called endpoints

6.18.4.1 Collecting Usage Data

- Ceilometer is part of the so-called “billing“ work flow
- It implements “metering“ which is the first step for billing

  Comment: The 2nd step for billing is called “rating“ which means setting a price for a consumed resource.
  The 3rd step is billing which is writing an invoice to a customer for the resources he has consumed.

- The collected consumption data are sent to Gnocchi for post-processing

  Comment: Gnocchi is an OpenStack-service with API for efficient storage and statistical analysis of Ceilometer data.

6.18.4.2 Collecting Events

- Collected events are sent to the aodh alarm service for postprocessing or to mySQL or PostgresSQL for logging

  Comment: PostgresSQL is a DBMS for object-relational databases and supports SQL with many extensions.

6.18.4.3 Collecting Alarms

- Collected alarms are sent to the aodh service or to ElasticSearch for postprocessing, or to mySQL, PostgresSQL, MongoDB for logging, or to an URI via REST POST

  Comment: Aodh is the alarm service of OpenStack. It triggers actions based on defined rules for meter or event data collected by Ceilometer or Gnocchi.

  Comment: ElasticSearch can search all kinds of documents. Its features are available through a JSON and a Java API.

  Comment: MongoDB is a DBMS for NoSQL databases that are text-oriented. It uses JSON documents with schemas.

6.18.5 Supported Hypervisors

- If a polling agent wants to collect usage data, events or alarms from a compute node then it must cooperate with the hypervisor of the VM
Supported hypervisors are Hyper-V, XEN and ESXi, as well as all others which are supported by libvirt, such as KVM/QEMU, for example.

6.18.6 Other Supported Tools

The notification agent is also able to receive notifications from two external metering services for networks which are OpenDaylight and OpenContrail.

Comment: OpenDaylight is a platform that allows users to write apps that work with a wide variety of hardware and protocols in the context of SDN.

Comment: OpenContrail is a large collection of external software for OpenStack.

6.19 The Heat Service

Heat is the OpenStack service for orchestrating other OpenStack services.

6.19.1 Orchestrating

Def.: Orchestrating is the program-controlled combining of web services by a central controller. An orchestration controller initiates actions at other web services and can check for their fulfilment and correctness. Orchestrating is a centralized way for automating the deployment and management of software, including their configuration.

By orchestrating, multiple existing workflows for cloud administration can be flexibly coupled.

This is accomplished by program-controlled REST requests to OpenStack services and by executing Chef, Puppet or Ansible scripts.

The Heat service is the orchestration controller for other OpenStack services.

It allows to automate:

1.) Typical admin workflows such as:
   • conditional launching and decommissioning VMs
   • conditional deploying and removing application packages
      • deploying and removing of packages typically implies that a specific sequence is obeyed how the software components of the package are installed
      • this can be ensured by Heat
      • deploying and removing of packages implies also that proper hardware resources are available
      • this can be checked by Heat before the installation begins
   2.) The installation of new OpenStack services in the same cloud
      • this allows for an automatic adaption of the software infrastructure of the cloud to new requirements of users and projects
   3.) Cloud monitoring and conditional intervening in case of events and alarms
   4.) Restoring of cloud functionalities after errors
5.) The installation of OpenStack services on new hardware in the proper sequence so that they can work together
   • this allows to create a OpenStack cloud in a new computer cluster by means of an existing OpenStack cloud
   Furthermore, admins can customise Heat stacks by installing Python plugins in their code
   In essence, cloud admins are significantly unburdened by orchestration
   Additionally, also users can profit from orchestration by writing their own stacks

6.19.2 AWS CloudFormation Templates
   Beside its native API, Heat has also an API for stacks written in Amazon's AWS “Cloud-Formation“ batch processing language
   This language is called “AWS CloudFormation template language“
   Def.: An AWS CloudFormation template is a parametrized text file that contains everything needed to carry out an orchestration. As soon as the parameters for it are set, the text file is ready for execution.
   CloudFormation templates are useful for orchestration because they model and create the virtual infrastructure of a whole AWS cloud
   In OpenStack, CloudFormation templates are called “cfn“ templates

6.19.3 Heat Orchestration Templates
   Heat has also its own template format called “Heat Orchestration Template (HOT)“
   HOTs are expressed in JSON or yaml format
   Comment: Yaml is functional similar to JSON. In contrast to JSON objects, yaml are objects not directly executable by Python. Furthermore, JSON brackets are replaced by line indentation in yaml for better reading, yaml is preferred to JSON because of its simplicity, readability and clarity. It is closer to human speaking than JSON and much closer than XML.
   HOTs are not compatible to cfn templates
   Compared to cfn templates, they are more restricted in functionality, but is in the process of surpassing the functionality of cfn templates
   A HOT describes the software infrastructure with respect to “OpenStack resources“ that a user application needs for running in the cloud
   Example: Typical OpenStack resources are compute VMs, floating IPs, Neutron ports, Cinder volumes, security groups, projects, user accounts and OpenStack services.
   A list of OpenStack resources is under: https://docs.openstack.org/heat/latest/template_guide/openstack.html
   From a practical point-of-view, a HOT is a named file in JSON or yaml format that contains all information to make a user application work
Each HOT is written in the HOT language which is the counterpart to the AWS-Cloud-Formation template-language.

Def.: The HOT language is a collection of predefined keyword/value pairs in JSON or yaml format, together with control structures such as if, repeat, not, and, or, in which also shell commands can be included, as well as Chef, Puppet or Ansible scripts.

Shell commands, Chef, Puppet or Ansible scripts must be stored in separate files and are included into the template by means of the “get_file“ function of the HOT language.

Example: For the inclusion of a shell script in a HOT:

get_file: http://ExampleOpenStackComputeNode.de/myShellScript.sh

Def.: Each file that is included into a HOT via the “get_file“ function is called plugin.

Furthermore, a HOT describes the relations between OpenStack resources.

Example: A HOT can state that a given volume is attached to a given VM, or that a given database user is added to a user group, or that a given keystone user is added to a Neutron security group.

Additionally, a HOT ensures that the installation of a user package happens in that sequence that is needed for the packet components to interact properly.

Example: A database component must be installed first before other components can be installed because they need typically a database.

Finally, a HOT allows not only for resource instantiations in the proper sequence but also for “higher functions“.

Example: For higher HOT functions: the creation and launching of a non-stop VM that has high availability, the auto-scaling of VMs, or so-called nested stacks.

Def.: Nested stack means that inside of a parent stack are one or more daughter stacks incorporated. This provides for the analogon of subroutine or macro calls, but on the template level. Nested stacks are referenced UUIDs or by their names. The concept of nesting stacks is recursive, i.e. a daughter stack can be a parent stack as well.

The specification of the HOT language is under: https://docs.openstack.org/heat/latest/template_guide/hot_spec.html#hot-spec

6.19.4 Difference between Heat Stack and HOT

Def.: A stack is a single resource instantiation or a sequence of it inside of a HOT.

Since the template is written in the HOT language, each resource instantiation is also accomplished in the HOT language.

By a stack, an instantiation from every resource type can be created analogously to a concrete variable with a name that is derived from a variable type.
Example: A security group in Neutron is a valid resource type. The creation of a specific
security group that is referenced by a name and used in an existing project is an instan-
tiation of the security group resource.

- Conversely, a HOT is a named file in JSON or yaml format in which individual resources
  are instantiated
- Each HOT has an URI to reference it, while each stack has an UUID or a name as pointer
to it
- A resource instantiation is referenced either via an URI or via an UUID or via a name

Example: A service resource instantiation is a concrete REST request to a given service
with a UUID in its request body.

Example: A stack resource instantiation is a daughter stack inside of a nested stack
which is referenced via UUID or name.

Example: For a VM resource instantiation in a HOT: A concrete VM is derived from an
image in Glance. The instantiation is described in yaml. The stack comprises in the HOT
everything except the first two lines.

heat_template_version: 2018-03-02
description: Simple template to deploy a single compute VM
resources:  -- from here on begins the stack
  MyVM:  -- instantiates an OpenStack resource
    type: OS::Nova::Server               -- OpenStack resource type for a
          -- compute VM. OS = OpenStack name
          -- space
    properties:  -- specifies the properties of
                   -- the instantiated VM resource type
      key_name: MyKeyEnvironmentVariable -- specifies name of OpenSSH key-pair
      -- to be used in the compute VM
      -- via an environment variable
    image: MyImageTypeEnvironmentVar   -- Image to be used for compute VM
    flavor: MyFlavorEnvironmentVar     -- specifies flavor of compute VM

6.19.5 Heat Setup

- Heat has the following components:
  - **heat command-line client:**
    - a CLI that allows for manual control
    - it communicates with the heat-api-cfn component to run cfn templates
  - **heat-api-cfn component:**
    - is the cfn API that implements some compatibility with AWS Clouds
    - processes cfn API requests by sending them to the heat-engine via oslo.messaging
      over RPCs
    - OpenStack developers, however, should use the native Heat API
  - **heat-api component:**
    - is the native Heat API
- processes native API requests by sending them to the heat-engine via oslo.messaging over RPCs
- **heat-engine:**
  - is the core of Heat
  - orchestrates the execution of HOTs
  - returns REST responses back to the caller
  - returns also events via oslo.messaging
  - returns finally **stack output**

### 6.19.6 Heat Terminology

**Def.:** The output section of a HOT defines stack output values that are available to the user after the stack has been executed.

**Example:** For stack output: IP addresses of deployed VMs, or URIs of deployed applications inside of a VM.

**Def.:** Stack snapshot: takes a snapshot of all resources in a stack into a JSON or yaml file.

**Def.:** Stack actions: performs actions on a stack given by UUID or name.

**Example:** For stack actions: suspend and resume a stack.

**Def.:** Stack events: Heat can monitor if an event is risen during the execution of a stack. The respective Heat REST request gets a response with the UUID of the event (if any), among other information.

**Def.:** Project software-configuration: Heat can set and manage the software configuration for a project (tenant). This includes the deployment of user applications and tools.

### 6.19.7 The native Heat API

- The native Heat API deals with resources, resource types, stacks, stack output, stack snapshots, stack actions, stack events, HOTs and project software-configurations
Amazon Web Service AWS

to be disclosed later