Cloud Computing's Notes

Daniele Bertagnoli

2022/2023

Contents

1 Introduction

Cloud computing is the product of a set of integrated technologies, such as:

- **Virtualization**, a collection of solutions which allow the abstraction of the fundamental elements for computing (hardware, runtime environments, storage and networking). The hardware virtualization allows to simulation the HW interface expected from an OS and also allows the coexistence of multiple SW stacks on top of the same HW. The application virtualization allows to isolate the application and manage in a finer way the resources that it uses. In that **process virtual machine** (like Docker) only one process is run.
- **Web 2.0**, the Web 1.0 is composed by static and dynamic pages without web applications. Web 2.0 is a richer platform for the application development, each application can be built by composing the already existing services. The developers can deliver these application through the internet using a set of technologies and services that allow the information sharing and application composition.
- **Service Oriented Computing (SOC)**, service orientation is the core reference of the cloud computing. A **service** is the main block used for the applications and system development. It's an abstraction that rapresents a platform-independent component and that can perform any action. A service is supposed to be:
  - **Language independent**
  - **Location transparent**
  - **Reusable**

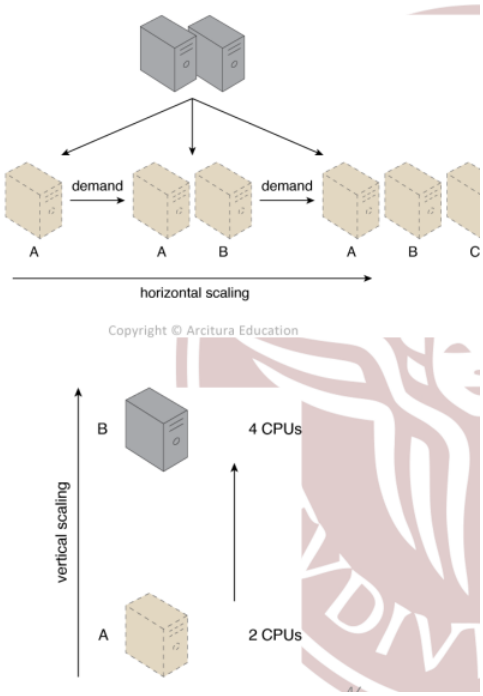
1.1 Definition of Cloud Computing

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.*
The main characteristics are:

- **On demand self-service**, allow to fulfill the service on-demand. Relies on interfaces (programming API, dedicated shell and web intefaces) and orchestration technologies.
- **Broad network access**, refers to the possibility of accessing the service via internet relying on the broadband network.

- **Resource pooling**, the computing resources can used from multiple clients simultaneously by using a multi-tenant model. Physical and vrtual resources are assigned dynamically according to the clint needs. The **Multi-tenancy** is a SW architecture where multiple clients (tenants) access the same application at the same time. Each tenant has its own vision of the application as a dedicated instance, this can be performed by using the virtualization. The difference between the multi-tenancy and the virtualization is that for the virtualization the physical resource is shared from the clients so both the OS and the applications are replicated. In the multi-tenancy architecture, the clients shares the application.

- **Rapid elasticity**, is the concept of scalability. We can scale in a horizontal way (resource replication) or in a vertical way (resource improvement).



- **Measured Services**, both of the consumer and the provider must care about the amount of resources that are currently used. From the provider point of view, it is important in order to guarantee the availability and choose a proper service cost. From the customer point of view it's important since he basically cares about the costs and the availability of the used service.

1.2 Business Drivers

1.2.1 Capacity Planning

We need to deliver right amount of capacities when needed. There are three different strategies:

- **Lag Strategy (reactive)**, we add capacity when the IT is fullfilled.
- **Lead Strategy (proactive)**, we add capacity to an IT resource in anticipation.
- **Match Strategy (proactive)**, we add capacity to an IT resource with small increments as demand increases.

1.2.2 Cost Reduction

A company typically wants to expands its IT capacity in order to cover the workload. So we need to care about several costs related to the cloud solution:

- **Migration (or cloudification)**, migrate all the data to the cloud. That cost can be significant but is required only the first time.
- **Annual cost of the cloud resources**
- **Annual operational overhead**

By using this solution, the costs in the long term period will be lower than manage a personal infrastracture.

1.2.3 Organizational Agility

It is the measure of an organization's resposiveness to change, the typical response time should be in hours or at most few days.

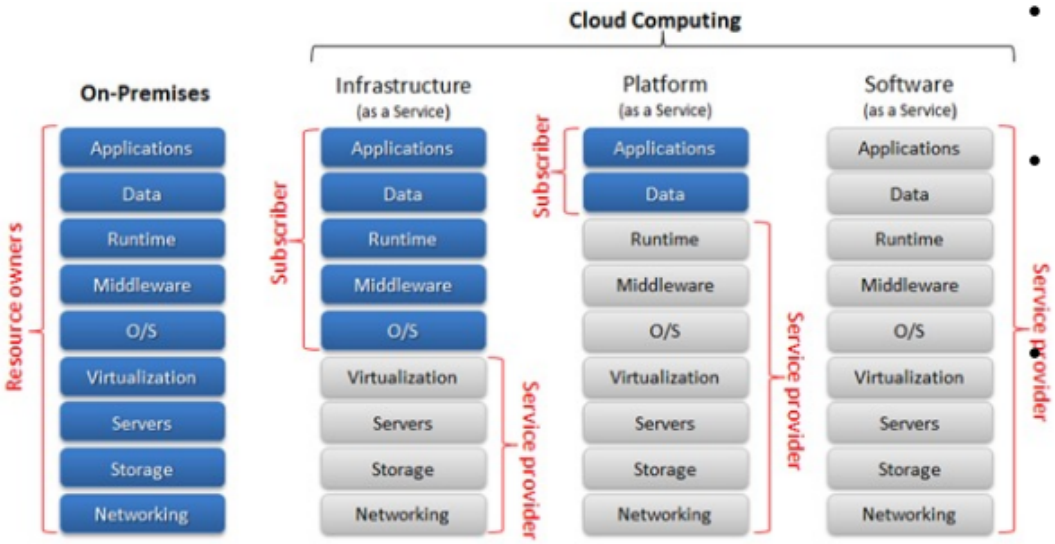
1.3 Service models

We can define the control of the resources based on the service:

- **Infrastracture as a Service (IaaS)**, the consumer controls from the application layer to the OS layer.
- **Platform as a Service (PaaS)**, the consumer controls the application and data layers, the other resources are demanded to the cloud provider.

- **Software as a Service (SaaS)**, the consumer has no control over the resources.

Each of these separation levels is used to limit the control of the cloud consumer and to separate the responsabilities between the consumer and the provider.



1.4 Deployment models

We can distinguish between:

- **Public Cloud**, all you can access.
- **Private Cloud**, any cloud is dedicated to a specific customer in logical/physical isolation with the other customers.
- **Hybrid**, we can run on our private cloud some processes and on the public cloud the others.
- **Community**, is used form a specific community of consumers from organization that have shared concerns. It can be own from third party providers, one or more organization inside the community or both of them.

2 Enabling Technologies

2.1 Distributed Computing

Is a computational model which defines how the computation is carried on. Distributed computation means **several units executed in a concurrent way on different computing elements**, so processors on different nodes, different processors on the same machine or different cores on the same processor. In distributed systems we have to manage the hardware and software hetereoginity and also the different location of the computing elements. **A distributed system is a collection of independent computers that appear to its user as a single coherent system**. Cloud computing is a specialized form of distributed computing that introduces the use of models for remotly providing of scalable resources.

2.1.1 Components of a Distributed System

We can split in layers the components:

1. **Hardware**
2. **Operative System**
3. **Middleware**
4. **Applications**

2.2 Architectural Styles

It is the way to organize the software components of a distributed system. In particular, it defines **different roles of components and how they are distributed across the system**. We can define:

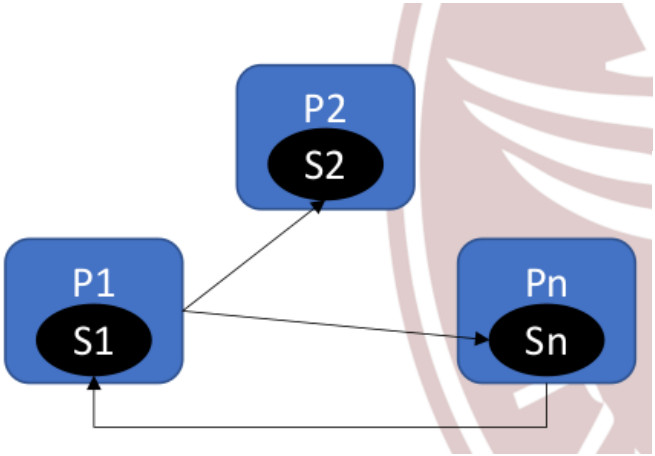
- **System Architectural Styles**
- **Software Architectural Styles**

2.2.1 Software Architectural Styles

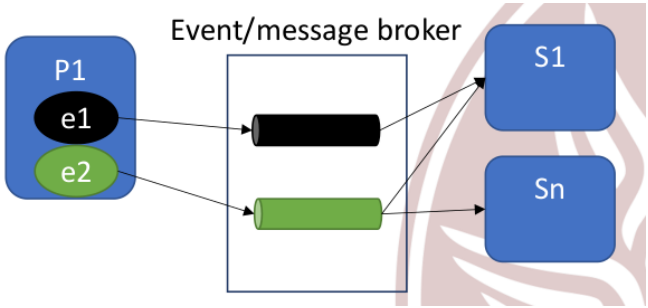
It defines the logical organization of the software. We have several styles, one of the most used is **Independent Components**.

Independent Components: each component has its own life cycle and interacts with the other components to perform activities. We can distinguish between two implementations:

- **Communicating processes**, each process provides services that can be used from the other processes.
- **Event based systems**, each componet exposes a set of events on which the other components can subscribe to. Each component that is subscribed to a certain event is called *Subscriber* and must provide a *callback function* that is executed when the event occurs.



(a) Communicating processes



(b) Event based systems

Microservices: is an architectural styles that structures an application as a collection of services (so independent components) that are **higly mantainable and testable**. Since each service is independent, we also have fault isolation between different services. This architecture enables the rapid and frequent deliver of large applications thanks to the high scalability obtained from the splitting of the microservices. A **monolithic architecture** (not splitted into microservices) is simpler to be developed, but when the technologies evolve and the applications are bigger than the scalability is much decreased. The main drawbacks of the microservice architecture are related to the **diffculty of find the right set of services that should be developed**. Moreover, the distributed systems are challenging to be developed and requires a lot of coordination among the development teams.

2.2.2 System Architectural Styles

It defines the physical organization of the components and processes over a distributed infrastracture. We have to main architectures **Client/Server** and **Peer to Peer**.

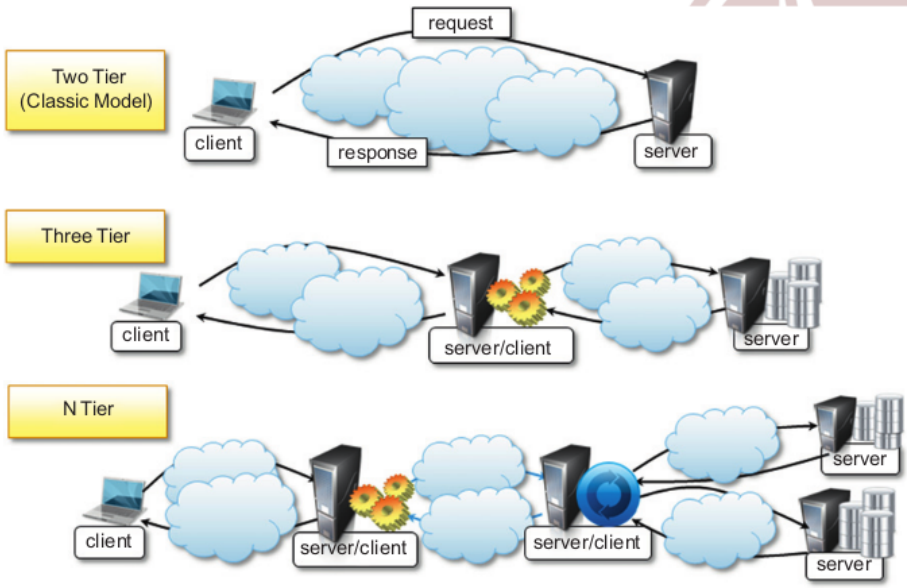
Client/Server Architecture: is suitable for *many to one* scenarios. The client will send *request* to the server that will reply with a *response*. We have 3 conceptual components called **tiers**:

- **Presentation**,
- **Application logic**
- **Data Storage**

These 3 components can be grouped by following two possibile models:

- **Thin clients model**, Client: Presentation | Server: Application logic and Data storage.
- **Fat-client model**, Client: Presentation and Application logic | Server: Data storage.

Based on the number of tiers, we can classify out system. The classic client/server architecture is a 2-tier model that suffers from scalability issues but is suitable for small systems. We can have up to N-tier architectures that guarantee a better scalability and a better performance since bottlenecks could be isolated.



Peer to Peer: is a symmetric architecture where all the components are called **peers**. Each peer act both as client and server, the consistency is mantained by using consensus algorithms. This architecture is suitable for **decentralized architecture** that has high scalability needs.

2.2.3 Service Oriented Architecture (SOA)

An application is an aggregation of services coordinated within a SOA. A service has these characteristics:

- **Boundaries are explicit**.
- **Services are autonomous**.
- **Services share schema and contracts**, a contract specifies the structure of messages that the service can send or receive.
- **Service compatibility is determinated based on policies**.

There are two main roles in SOA:

- **Service Provider**, it publishes the service in a registry together with a service schema and contract.
- **Service Consumer**, locate the service metadata in the registry and develop the required client components to use that service.

Services can be managed and coordinated in two ways:

- **Service Orchestration**, there is a *service orchestrator* which coordinates and manages the interactions between services.
- **Service Choreography**, the interactions are coordinated by exchange of messages and rules between the services without the needs of a central authority.

SOA vs Microservices: both are architectures composed by set of independent components. There are 3 main differences:

- **Communicating mechanisms**, microservices uses lightweight and open source technologies, instead SOA uses heavyweight technologies and *Enterprise Service Bus* (communication middleware that contains business processing logic).
- **Data**, SOA applications typically have a global data model and shares databases, in microservices architecture each service has its own data model.
- **Size of services**, SOA is typically uses few large services, microservices applications are composed by tiny services.

2.3 Interprocess Communication

Processes use messages, *serialized objectes*, to exchange data between each other. Processes can interact by using differnt techniques:

- **Remote Procedure Call (RPC)**, consist of calls of procedures beyond the boundaries of a single process. This technique uses an implicit client/server paradigm where the client is the process that calls the procedure and the server is the process that runs the code and computes the output. The **RPC Infrastracture** will manage the communication between the client and the server by using an *RPC API*. Each process that provides a procedure, must subscribe that function to a *register* to make it available to the other processes.
- **Distributed Objects**, is an implementation of RPC for object-oriented paradigm. Each process registers a set of interfaces, these can be accessed from the other processes by using a pointer. The infrastracture will transform the local method invocation into request for the remote procedure. The main drawbacks are that we need to manage the object states and also that each method remotly called is associated to an instance of an object, and that instance can be created:
  - **by value**, a copy of the object is created and the server will work on that copy.
  - **by reference**, the object is not duplicated and the server works on the object pointed. This method is more complex and the infrastracture must manage all the communications.

  The object needs to be activated, this can be done in two ways:
  - **Server-based**, the object has its own life and the remote method is called occasional. The object is created by the user.
  - **Client-based**, the object is created only with the purpose of executing the remote method. The creation is implicit and the infrastracture manages its life.
- **SOA**.

2.3.1 Web Services

It is an implementation of RPC paradigm over HTTP, it allows the interaction of components that are developed with differnt technologies. A *web service* is exposed as a remote object and the method invocations are transformed in HTTP requests by using one of these two protocols:

- **Simple Object Access Protocol (SOAP)**.

- **Representational State Transfer (REST)**, lightweight alternative to SOAP the HTTP methods (GET to retrive a resource, POST to create a resource, PUT to change an existing resource, DELETE to remove a resource) are used to implement the operation requested by the Web Service. REST Web Services must be **stateless and expose directory structure-like URIs to expose remote functions.**

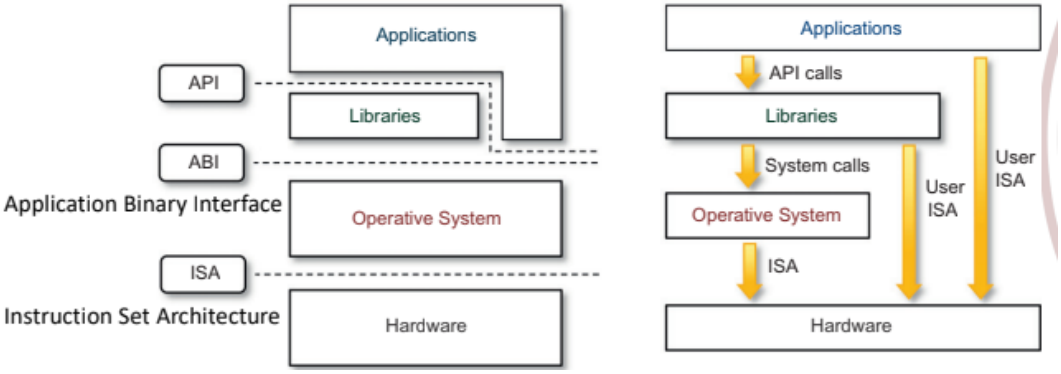
3 Virtualization

Virtualization is one of the main technologies for the Cloud Computing. Virtualization is usefull for several reasons, by using this technique we can use the resources when needed, otherwise we can assign that resources to other jobs. We also have **gains in term of portability** since these virtualized systems are run by using **virtual images** that can moved and used on several machines with a **Virtual Machines**. Moreover, by isolating the processes we can also **increase the system security**:

- By filtering the activity of the guest and preventing the execution of harmufull actions.
- Sensitive information can be hidden to the host without complex security policies.
- Some resources can be hidden by the host, so the guest cannot use them.

3.1 Machine Reference Model

To understand how the different virtualization techniques work, it is fundamental to recall the layered structure of the a computer system:



We can distinguish between two different type of instructions:

- **Non privileged**, can be used without interfering with the other instructions.
- **Privileged**, are executed under specific restrictions and are used for sensitive operations:
  - **Behevior-sensitive instructions**, operate on I/O.
  - **Control-sensitive instructions**, operate on CPU registers.

3.2 Hyper-visor

It runs in supervisor mode, its aim is to emulate and manage the status of the CPU for guest operating systems. All the sensitive instructions are executed in privileged mode. The Hyper-visor is also know as **Virtual Machine Manager** and is composed by:

- **Dispatcher**, the entry point.
- **Allocator**, it provides the resources to the VM.
- **Interpreter**, implements the interpreter routines and is executed every time that the VM executes a privileged instruction.

Each VMM should satisfies these properties:

- **Equivalnce**, when a system is virtulized, it must work in the the same way as it was executed as a physical system.
- **Resouece Control**, the VMM must be able to have the full control on the virtualized resources.
- **Efficency**, most of the instructions must be executed without intervention from the VMM.

3.3 Virtualization Techniques

We have 4 main virtualization techniques:

- **Full Virtualization**.
- **Hardware Assisted Virtualization**.
- **Paravirtualization**.
- **Partial Virtualization**.

3.3.1 Full Virtualization

The guest OS does not know that it is an emulated OS. The VMM will scan the instruction stream, the non-critical instructions run on the hardware directly, the privileged instructions are run by the VMM that emulates their executions. To emulate this instructions we use the **binary translation** technique. Sequences of instructions are translated from a source instruction set to the target instruction set. We have two differnt types of binary translation:

- **Static**, all the code of an executable is converted into the code that can run over the target architecture.
- **Dynamic**, small pieces of code are translated at run-time only when needed. We have an overhead due to the run-time translation and the translated instructions are cached, so when the instructions are run several time we obtain a efficency gain.

3.3.2 Hardware Assisted Virtualization

It's a full virtualization that uses dedicated hardware to improve performance. A set of additional instructions are added to control the start and stop of the VM and allocate memory pages.

3.3.3 Paravirtualization

A para-virtualized OS is assisted by an intelligent compiler to replace the non-virtualizable OS instructions by hypercalls. This technique prevents the performance losses but the guest OS needs to be modified and the compatibility is decreased. The virtualization is not transparent to the guest OS.

3.4 VM Migration

VM migration means transfer a VM from a machine to another machine, we can have two migration types:

- **Offline Migration**, the VM is stopped, moved and restarted.
- **Live Migration**, the VM is transferred without any service interruption.

There are 6 steps:

1. **Pre-migration**, the VM is active on host A, the new host is choosen.
2. **Reservation**, initilize a container on the new host.

3. **Iterative pre-copy**, the whole VM execution is stored in the memory and sent to the new host. After that, the old host sends the copy of all the data that are changed during the previous transfer. This is iterated until the *dirty portion of the memory* is small enough.

4. **Stop and copy**, the old VM is stopped and all the traffic that was directed to the old VM is now redirected to the new one. The old VM execution is interrupted when all the memory data are transferred to the new VM. The time between the stop of the old VM and the start of the new VM is called *downtime* and should be shortest as possible.

5. **Commitment**, the old VM state is released.

6. **Activation**, the new VM starts and all the services are resumed.

3.5 Containers

With containers we do not have any hyper-visor and no hardware simulation. A container can be used to distribute and run an application, microservice or an entire OS. It is an evolution of *chroot*, the process and its children cannot access other file system folders except those that they are allowed to access. Containers are based on:

- **Namespace**, wraps all the system resources into a single abstraction. They defines which resources are visible to other processes.

- **Cgroups**, is a Linux kernel feature which allows processes to be organized in a hierachical groups.

- **UnitFS**, unioning allows administrator to keep files separated physically but merge them logically into a single view. Each physical directory is called **branch**, in case of files named in the same way, we have to specify a priority between all the branches (unionFS will remove any duplicated files inside the lower-priority directories). UnionFS can also use the *copy-on-write* technique by mixing read-only branches and read-write branches, the resulting file system will be whole read-write. The read-only branch will be not modified.

3.5.1 Docker

Docker is an engine platform independent that provides a set of command to handle. We can interact with the **Docker Deamon** (the server) by using **Docker CLI** to or **REST API** (REST API is at a lower level and is used implicitly from CLI). Docker uses OS images, that are read-only, by instanciating them into **containers** that are read-write. The **Docket Registry** is a local (but can be also global) repository of docker images. When we use the command *docker run 'OS image' 'command'*, docker will:

1. Contact the registry to download the required image file.
2. Instanciate a new containter and allocate the read-write file system.
3. Run the selected command on that container.

3.5.2 Docker Storage

There are several solutions related to the **Docker Storage**:

- **Container writable layer**, does not persist when the container is stopped. This solution has a reduced portability and reduced performance.
- **Bind mount**, a directory of the host file system is mounted in the container (if the directory does not exist, docker will create it). That directory is shared between the container and the host system, this solution is not portable but the performance are better than volumes. It a suitable solution when we want to share files between containers, share source code between a development environment on the docker container and the host system.
- **Volumes**, is a directory created and managed from docker, so is completly isolated from the host. The pros of using volumes instead of other solutions:
  - Easy to backup.
  - Portable and easy to share between containers since are platform independent.
  - Volumes can be pre-populated.

  Volumes are a suitable solution when we want to share data among running containers or when we need to backup, restore or migrate from a docker host to another.
- **tmpfs**, it is an area outside the container writable layer. Is a temporary memory that is deleted when the container is shutted down. It's available only on Linux hosts, so is not portable and cannot be shared with other containers. The pro is that is a very fast memory. It is a good solution when we do not want that the data persist when the docker container is stopped.

3.5.3 Docker Service

We can create a **swarm**, so a set of docker daemons and containers, by using a **Docker Service**. It defines how many containers should be available, the load balancing, the constraint for each container and other options.

3.5.4 Docker Networking

We can connect the docker container together by using an external technique (like a server) or by using a **network driver**. There are several network drivers:

- **Bridge**, is the default network driver and is totally managed by Docker. This solution is not optimal since Docker introduces some delay.
- **Host**, used for standalone containers. It removes the isolation between the Docker host and the containers by using the host's networkig directly.
- **Overlay**, it connects two containers that run over two different Docker daemons. This technique removes the OS-level routing between these containers.
- **Macvlan**, allows to assign a MAC address to the containers making it appears as a physical device. The Docker daemons will route the traffic by using the MAC address. This solution is suitable for applications that have the need to be connected physically to the network.
- **None**, the networking is disabled.

4 Autonomic Computing

4.1 Data Center

A data center is a large system which manages several services such as:

- **Consolidate servers**, split servers into more virtual layers and manage the VM migration.
- **Guarantee Service Level Agreements**, so guarantee a certain level of quality of the service.
- **Optimize energy consumption**.
- **Manage the hardware and software failures**.
- **Manage software updates and fix bugs or vulnerabilities**.

These systems must be able to manage themselves, the human intervention is required only for physical tasks or very important decisions.

4.2 Autonomic Computing

Autonomic computing systems are capable to manage themselves given some high-level goals from the administrators. The **autonomic manager** will monitor the system state, analyze the data, perform action based on that data to mantain the system in the desidered state. We can define some properties for these systems, called **self*- properties**:

- **Self-healing**, guarantee that a certain number of VMs are always running. If one of these VM/container is not responding the manager will allocate/start a new VM/container.
- **Self-configuration**.
- **Self-protection**, use a *Intrusion Detection System (IDS)* to detect attackers and prevent them to compromise the system. The system must be able to detach a compromised node and replce it with a sanitized new node.
- **Self-optimization**, provide the right amount of VMs to guarantee the *SLA* and minimize the costs. The manager decides if VMs must me removed or added if a *Service Level Objective* is violated.

4.3 Autoscaling (Self-optimization)

The performance measures how fast a system can complete a task, the scalability measures how the performance varies related to the increasing of the load. Autonomic computing system must be able to scale in an autonomous way in order to keep the performance higher as possible. There are two main autoscaling mechanisms:

- **Course grain**, the system will replicate a whole application typically it requires the deployment of a new VM.
- **Fine grain**, the system will replicate an application component (microservice).

The **automated scaling listener** is a component which allows the **dynamic scaling architecture** to be used. The listener will take decisions based on the performance measurements and the number of the worload counters (e.g. number of service requests). The system can either scale

- **In/Out**.
- **Up/Down**.
- **Migrate to system with more/less resoruces**.

4.3.1 Autoscaling Algorithms

We can distinguish between several types of scaling algorithms:

- **Threshold Based (reactive)**, when a threshold on the workload or the resource usage is exceeded, the scaling mechanism is triggered. Not optimal.
- **Model Based (reactive)**, uses a mathematical model based on workload or performance metrics to choose how to scale. Could be optimal.
- **Proactive (both threshold and model based)**, the values of the workload is predicted (5-10 minutes ahead) and the previous algorithms are executed based on that assumption.

4.3.2 Simple Scaling vs Step Scaling

By using the simple scaling policy, our system will:

1. t_1, determine when to scale.
2. t_2, time in which the scaling is performed.
3. t_3, **cool down period** in which we cannot take any action.
4. t_4, time in which a new action can be taken.

In the step scaling policy, out system will:

1. t_1, determine when to scale.
2. t_2, time in which the scaling is performed.
3. t_3, time in which a new action can be taken.

Since in the step scaling we do not have any cool down period, we are more reactive to changes.

4.3.3 Scaling policies

We can define several way to specify how the system must scale:

- **Change in capacity**, we specify the adjustment in term of capacity (so VMs).
- **Exact capacity**, we specify the new capacity of the system.
- **Percentage capacity**, we specify the adjustment in term of percentage respect to the current capacities.

The **peformance metric value** is the metric used to take the decision to scale or not. For example, we can consider the avarage of the CPU utilization of all the VMs at time t: $\frac{\sum_{i=1}^{N} CPUutil(t, VM_i)}{N}$ with N number of VMs. The **warm-up period** is the time that each resource takes to work to its full capacity. Until that period is expired, the resource is not considered for the performance metric value. We can also use **alarms**, to avoid that we allocate or deallocate resources in a fast way. Basically we enable the scaling if the metric exceeds a certain threshold for a time period.

4.4 Cloud Orchestration

Cloud orchestration is a process of automating the needed tasks to manage connections and operation workloads.

4.4.1 Ansible

Ansible is an orchestration tool, in particular it's an **agentless tool** that runs in a **push model**. Agentless means that there is no need of installing software on the remote machines to make them manageble and so there is no overhead. Ideal for the systems where there are high-security and high-performance needs. Since Ansible uses the push model, there are **Ansible modules**, so the code and the instructions, that are stored on the **control server** and then pushed to the host by using existing frameworks present on the host (e.g. SSH). In Ansible we have **playbooks**, YAML descriptions of what each component of the IT infrastracture have to do in order to perform the required operation. Through playbooks we can perform automation on the IT environments and how the components have to cooperate. Playbooks are composed by **plays** which define the automation among a **host inventory** (set of hosts). Each play is a set of Ansible modules (small task). When a task is required to be executed, **core modules** check that the required task should be effectvely executed (e.g. if the task is start a server, the core modules will check that the server is not already started).

4.4.2 Kubernetes

Kubernetes is a platform for managing containerized workloads and services. This means:

- Discover a service and perform load balancing.
- Storage orchestration, so mount and unmount storage.
- Automated rollouts and rollbacks, bring back the container into a certain state.
- Automatic bin-packing, manage the resources for each container.

- Self-healing.
- Self-configuration.

The main components of Kubernetes are:

- **Control Pane**, which runs on a dedicated VM or server. Through the **kube-scheduler** we can schedule tasks based on some parameters (deadlines, resources, hardware and software constraints, etc...). The **kube-API** is entry point of the control pane. It can connect to cloud by using the **cloud-controller-manager**.
- **Nodes**, nodes can be either a VM or a physical machine. The **kubelet** is the main process that manage and execute the received the instructions on the pods. The node is accessed from the control panel through the **kube-proxy**, it also controls the network rules to access the pods. Each node has JSON file which describes its status, the address and other parameters. Each nodes runs a **container runtime** such as Docker that manages the containers.

Kubernetes uses **Kubernetes Objects**, persistent objects which describe the state of the cluster and the desidered cluster's state. More in detail, they describe which container is running, the assigned resources and the policies (update, restart, fault tolerance, etc...). In order to get the node into the desidered status, Kubernetes uses **Kubernetes Controllers**, so control loops that send information to the API that should take actions to correct the state. A **pod** is the smallest deployable units of computing that can be created by Kubernetes. Each pod can be considered as a *logical host* in Kubernetes is composed by one o more containers that shares storage and network resources. Each node runs multiple pods, when a pod is created the kube-scheduler checks for nodes that meet the pods requirements, in term of resources, and will mark it as **feasible node** (*filtering*). Then the best node will be chosen to run the pod (*scoring and binding*), if there are no feasible nodes the pod will wait for them.

5 Cloud Storage

There are differt days to store date on the cloud systems:

- **Distributed filesystems**, Google filesystem, Hadoop filesystem, etc...
- **NoSQL databases**, Cassandra, MongoDB, etc...
- **Key value storage systems**, Dynamo, BigTable, etc...

The main goals is to have **high availability, massive scaling on demand and simplified application development and deployment**.

5.1 Atomicity

The multi-step operations called **transations** should end without any interruptions, they must be **atomic**. This requires **HW support and mechanism to access shared resources** (such as semaphores, monitors) which allow to create a **critical section**. The HW operations are:

- **Test-and-set**, write a memory location and returns the old memory content.
- **Compare-and-swap**, compare the the content of a memory location to a given value, if the two values are equal it modifies the memory with new given value.

There are two main types of atomicity:

- **All or nothing**, is composed by two phases:
  - Pre-commit phase, some preparatory actions that can be undone are performed. Such as memory allocation, resource allocation, etc...
  - Post-commit phase/commit step, in which all the irreversible actions are performed.

  In order to manage failures and ensure consistency we have to mantain the logs of the activities.
- **Before or after**, all the concurrent actions have all the *before-or-after property*, so their effect, from the point of view of the invokers, can be applied completly before or completly after another.

5.2 Storage Model

The storage model describe the layout of a data structure in a physical storage. A physical storage can be a local disk, an USB disk, disk that can be accessed from the network, solid or magnetic, etc... We want **read/write coherence and at leat one of the two atomicities**. We have two main storage models:

- **Cell Storage Model**, we have cells all of the same size and each object fits exactly in a cell. The read/write unit are either sectors or blocks. We have the **primary memory** composed from an array of cells and the **secondary storage device** (e.g. disk) organized in blocks or sectors. Guarantees only before-or-after atomicity.
- **Journal Storage Model**, model used to store complex objects like records with multiple fields. Is composed by a manager and a cell storage. The enire history of a variable is stored in the cell storage, where each update of a data is a record appended to the log that can

be used to reconstruct the cell. The log is kept on the non-volatile memory, the record can be either kept on the non-volatile or the volatile. The user cannot access directly the cells but has to send requests to the manager which translates requests (read a cell, write a cell, etc...) into commands sent to the cell storage. An all-or-nothing action first will record the action in the log and then perform the action. This model can guarantee both atomicities.

5.3 Distributed Storage

5.3.1 Google Filesystem (GFS)

Is a **distributed filesystem, the main idea is to use normal server to store data** (instead of dedicated storage solutions). Designed for the cloud, so with **high availability, robust against failures and errors** (software bugs, human errors, system software errors). The most important feature is the **cloud access model**, the file sizes ranges between few GBs to hunder of TBs that can be accessed with the same performance. We can do sequential read and append write (so we increase the size of a file rather than add a new file). The responce time is not one of the main requirements. To simplify the system implementation, we have **relaxed consistency** (since the model is distributed the consistency was a problem). Data are stored in chunks, fixed-size segment (typically 64MB) replicated in more sites. Large chunks allow to **increase the performance during the read operations, decrease the probability to do another read request, decrease the amount of metadata stored and decrease the fragmentation of the disk.** GFS follows a master/slave paradigm, the **chunk servers**, so the slaves, store the chunks and the **master** stores the metadatas and manage the state of the chunk servers. The master also stores in the main memory the location of all the chunks, they are update at the system startup or when a new chunk server joins the network. The **master periodically checkpoints its state** in order to minimize the recovery time in case of failure. All the operations are atomic and cannot be seen by the clients untile the changes are committed on more replicas of the system, the logs are mantained by the master. When a **client wants to retrive data from the GFS, it will contact the master** that will reply with the chunk server address. Now the application can contact the chunk server and **the operation are handled by the application and the chunk server**. The **master is involved only in chunk creations**.

5.3.2 Hadoop Distributed Filesystem (HDFS)

Is a distributed system written in Java which is used to process large volumes of data. Is portable but the performance are not very high (since we are focussed on the reliability of the filesystem). Replicates data on multiple nodes to be more robust against failures. Even in this case we have a master/slave paradigm where the master is called **name node** and the slaves that store data are

called **data node**. The size of the chunks can be configured from 64MB up to 128MB. The write operations are composed by 3 stages:

1. **Set up pipeline**
2. **Perform the operation through the pipeline**
3. **Shutdown the pipeline**

For the read operation, the client ask to the name node for the address of the data node to contact. After this it will ask for the chunks directly to the data node.

5.4 No SQL Storage

databases can be either relational or NoSQL. In **NoSQL DBMS the amount of data is very large and does not require a relational model**. We have several types of NoSQL DBMS:

- **Graph databases**
- **Key-values databases**
- **Documend based databases**

These are all designed to **have high availability without having a single point of failure**. Moreover, the can scale well horizontally. To manage the decision, they use consensous protocols.

5.4.1 BigTable

A Distributed Storage System for Structured Data designed by Google, it uses the Google Filesystem and the **SSTable** designed by google. SSTable is **file format is used internally to store Bigtable data**, it contains sequence of blocks and a block index to locate blocks. The access to a row is performed using a binary search process on the indexes that are prevently loaded in the memory from the disk. It guarantees **wide applicability, scalability, high performance, and high availability for large amounts of workload types**. It is based on simple and flexible data formats, supporting the dynamic control over them. The client can control the data schema used to rapresent them and if the data must be stored in the main memory or from the disk. The read and write are atomic. Data are **indexed using row and column names, that can be arbitrary non-interpreted strings, in a sparse map**. The rows are sorted by following the lexicographic order. **Tablets** are a small set of rows, they are created to read a small amount of rows in a more efficient way. **Columns are grouped in column families**, all the data stored in a column family must be of the same shape. We can also create a 3D table in which we save diffent

versions of data based on the timestamp. Only the newest version of the tables are kept, the other are discarded. The timestamp is application dependent and can be set, by default is milliseconds. BigTable follows master/slave paradigm, the master assignes tablets to tablet servers, balances tablet-server load, performs garbage collection of files in GFS and handles schema changes such as table and column family creations. The slaves are the **tablet servers**, they stores diffent amount of tablets by handling the read/write operation on then. **Chubby** is a component which ensures that there is at most one active master at any time.

5.5 Amazon Dynamo (Key-Value Storage)

Highly available and scalable distributed key-value storage used to manage the state of services that have very high reliability requirements. Many services on Amazon's platform only need primary-key access to a data store (such as best seller lists, shopping carts, customer preferences, etc...). Dynamo tradeoffs between availability, consistency, cost-effectiveness and performance. It is based on **optimistic replication techniques**, so the replicas can diverge. **Conflicting changes must be detected, resolved solved by the application at read time, through quorum in the replicas and then propagated to all the replicas in an asynchronous way**. **An operation is considered as successfull if the majority of nodes has completed it**, the time of completation depends on the slowest node. Due to the asynchronous propagation, could happen that a read is done before other write are completed, there applications allow this for some operations. In this case the data must be reconciliated, so the two versions are merged. To scale incrementally, the **data are partitioned using an hash function over the data keys** which allows to distribute the data among the nodes in the cluster. The main challenge is to distribute in a uniform way the data. If a node becomes unavailable, the load is distributed among all the other nodes. The other nodes in this way handle **virtual nodes** loads, the number of maximum virtual nodes depends on the capacities of the node. To achieve a high availability in this case data are replicated on multiple hosts, when a key is hashed into a node it will be replicated on the near nodes.

6 Amazon Web Services (AWS)

A web service is a piece of software that make itself available over the internet and used standardized formats (e.g. XML, JSON, ...) for the request and response of an API. AWS is a **secure cloud platform that offers a set of global cloud-based services**. So we can access on-demand each service and pay for the time we use each service. We should select a service based on the business goals and technology requirements.

We can interact with AWS by using:

- **AWS Management Console (GUI)**.
- **Command Line Interface (CLI)**.
- **Software Development Kits (SDKs)**, for developers only.

6.1 AWS Cloud Adoption Framework (AWS CAF)

It provides guidance and best practices to **help organizations to adopt the cloud solution**. AWS CAF is organized in six perspectives:

- **Business Perspective**, CAF can be used to create a strong business case for cloud adoption.
- **People Perspective**, CAF can be used to evaluate the organization's structure and roles.
- **Governance Perspective**, CAF can be used to minimize the risks and maximize the investments.
- **Platform Perspective**, CAF can be used to describe the system architecture through models and patterns.
- **Security Perspective**, CAF can be used to ensure that the organization meets the stakeholders' security needs.
- **Operation Perspective**, CAF can be used to define the daily, monthly and yearly operations.

The first three perspectives focus on the business capabilities, the last three are most focussed on technical capabilities.

6.2 AWS Pricing Model

There are three main drivers of cost in AWS:

- **Compute**, compute capabilities that we are using.
- **Storage**, amount in GB that we use.
- **Data Transfer**, amount of outcoming GB.

We can pay based on the used resources, but we can get some discounts:

- If we reserve resources instead of allocating on-demand them, based on the amount of reserved resources we can get up to 75% discount:

  - **AURI** (All Upfront Reserved Instance), lager discount.
  - **PURI** (Partial Upfront Reserved Instance), medium discount.
  - **NURI** (No Upfront Reserved Instance), small discount.
- If we use a lot of resources we will pay a lower cost for the individual resource.

In this way each custer can start to use AWS. Moreover, AWS offers some services without adding costs such as **Amazon VPC** (Amazon Virtual Private Cloud allows to create a logically isolated section of the cloud where we can start new services) and **AWS Identity and Access Management** (which is used to define the users' access rights and accounts).

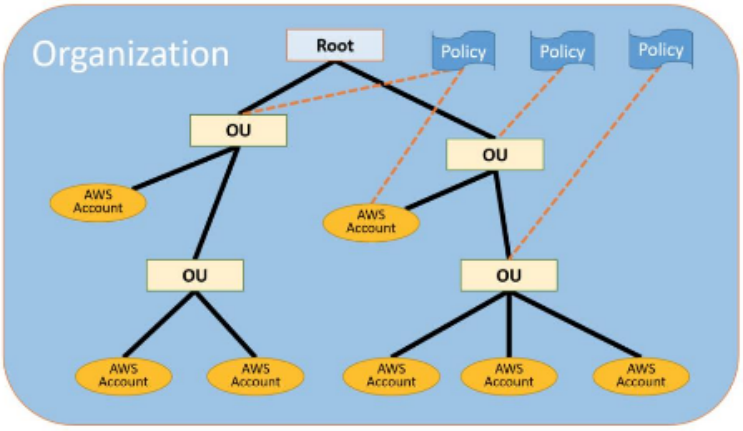
We can estimate the cost, both *direct* (like power, floor space, etc...) and *indirect* (like network and storage infrastracture), of our business that uses AWS through the **Total Cost of Ownership (TCO)**. With this model we can compare the cost between using AWS and using an on-premise solution. TCO considers:

- **Server Costs**, hardware and software costs.
- **Storage Costs**, hardware and management costs.
- **Network Costs**, network hardware and network administration costs.
- **IT Labor Costs**, costs to administrate the entire solution.

AWS offer the **Amazon Cost Calculator** which can tell us the monthly cost of our service, the cost reduction possibilities that we can take and the contracts that meet our needs.

6.3 AWS Organization

Is a free account management service which is able to create the organization as a tree. Under the root we can find several **organizational unit** which defines a role. We can attach a **policy** to a OU, they defines which services can be accessed from the accounts under that OU. All the accounts under the OU inherits the policies attached above.



There can be at most 5 layers under the root with up to 1k policies and 1k OUs.

6.4 AWS Technical Support

AWS offers **Technical Account Managers (TAMs)** which provides a proactive guidance to keep us informed on our plans and solutions. AWS offers different support plans based on the customers needs:

- **Basic Plan**, we can access FAQs, dashboards, forums and ask for support checks.
- **Developer Plan**, can ask for help only during business hours with normal or low priority (12h or 24h)
- **Business Plan**, customers that are running more than one application and have multiple services active. Can ask for support at any hour (1h or less).
- **Enterprise Plan**, customers that are running business and mission-critical workloads on AWS. Can ask for support at any hour (15m or less).

6.5 AWS Infrastracture

The AWS infrastracture is designed to be flexible, reliable, scalable and secure with high worldwide performance. A **region** is a geographical area, they can communicate with eachother through the AWS backbone infrastracture. The **data replication** around the regions depends from the customer. When we choose the region in which we want to deploy our service we must care about government laws, costs and latency. Each region has multiple **availability zones**, a full isolated partition of the AWS infrastracture. We can replicate data around the different availability zones

in order to increase the fault tolerance and the availability of the service. The **data centers** have a redundant design (fault tolerance) and the data are backed up multiple times around multiple availability zones. **Point of presence** are located around the world, they monitor performance in order to find the best way to route the requests.

6.6 AWS Service

AWS mainly focuses in IaaS and PaaS rather than the SaaS. The main service's categories are:

- **Storage Services**
- **Compute Services**
- **Database Services**
- **Networking Services**
- **Storage Services**
- **Security Services**
- **Cost Management Services**
- **Management and Governance Services**

6.7 AWS Shared Reponsability Model

AWS is responsible for the **security of the cloud**, so hardware security and all the software security related to the compute/storage/database/networking services. The customer is responsible for the **security in the cloud**, so the customer's data security, the application deployed, the operating system deployed on the VMs, the network firewalls configured and the account management on the application.

6.7.1 Identity and Access Management

The user can use **Identity and Access Management (IAM)** to manage the access to AWS resources by defining who can access the resource, how the resource can be accessed and what the user can do with it. The main components are:

- **IAM user**, a person or application that can authenticate with an AWS account.

- **IAM group**, collection of IAM user that have same authorizations. There is no default group, so gropus must be explicitly created. A user can be part of more than one group.
- **IAM policy**, the documents that defines the level of access to each resource and what can be done with it. We can specify two types of policies:
  - **Identity Based**, a policy attached to a IAM entity (user, group o role).
  - **Resource Based**, a policy attached to a resource.

  Each user should have the *least privileges*, if a right is not explicitly defined then the user will be unable to use it.
- **IAM role**, is similar to a IAM user, but is not associated to a specific user. It grants temporary permissions.

We can define the **type of access** for each user:

- **Programmatic Access**, the user can authenticate himself through an *access key ID* and *secret access key*.
- **AWS Management Console Access**, the user can authenticate himself through *IAM user name and IAM password*.

6.8 Amazon VPC

Enables to create a logically isolated section inside the AWS Cloud that is dedicated to a specific account. A VPC belongs only to one AWS region, but can span over multiple availability zones. A VPC can be splitted in more **subnets**, range of IPs that can belong only to one availability zone. When a VPC is created, we can specify a **CIDR block**, so a range of private IPv4 (also IPv6 is supported) that cannot be changed after the creation.

We can also assign a public IPv4 address to a VPC, this is done through an **elastic IP address** or a **public IPv4 address**. An elastic address is a static address associated with an AWS account and can be allocated and remapped anytime but has a cost to be paid. The public address is free but cannot be reassigned.

To redirect the network traffic we can attach to a VPC an **elastic networking interface**. As for the elastic addresses, also these interfaces can be reassigned anytime.

6.8.1 Route Tables

A route table contains a set of rules that can be configured to direct the traffic inside the subnet. Each rule **the destination and a target**. Each VPC must be associated with a route table.

6.8.2 VPC Networking

A **gateway** is a scalable component which allows the communication between the VPCs and the Internet. In VPC we are using **NAT**, so an addressing system that is valid only inside the VPC. Inside the private network we can address each resources but they cannot be contacted directly from the internet. The **NAT gateway** is the device that connects the internet with the VPC and act as a bridge by mantaining both the public and private routing tables. We can **share a VPC** between different accounts of the same organization. Two private VPCs that are part of the same network can communicate directly with the **peering**. If the data center used is far away from the chosen data region, the performance can be negatively affected by it. We can use **AWS Direct Connect** to establish a private connection between the data center and the DX Location to increase the bandwidth. A **VCP endpoint** is a virtual device that allows to connect the VPC to the AWS services without the need of VPN, NAT or gateways. We can have two types of endpoints:

- **Interface VPC endpoint**
- **Gatway endpoint**

AWS transit gatway is a service used to simplify the networking model for VPCs. It act has an hub which manages the traffic between multiple VPCs. It can be difficult to be deployed if we have hundred VPCs to connect with eachother, but simplifies a lot the deployment of a new VPC since it has only to connect to that hub instead of configuring all the connections with the other networks.

6.8.3 VPC Security

A **security group** act as a firewall for the VPC by controlling the incoming and outcoming traffic and filtering the packets. They are composed by **rules** which establish the allowed traffic, by default the inbound traffic is denied and only the outbound one is allowed. We can create personal rules where we can specify the allowed traffic (but not the denied!). Security groups are **statefull** and all the rules are evaluated before the decision of allowing a conneciton. We can add an additional security layer with the **Network Access Control List (ACL)** which defines (as the security groups) the rules of incoming and outcoming traffic but it works only between the subnets. In this case by default all the traffic is allowed and we can specify both deny and allow rules. The rules are evaluated by starting from the lowest rule number. ACL is **stateless**.

6.9 Computing Services

6.9.1 Amazon EC2

Is the most simple compute service that provides VMs, so can basically run every application on it. We have full control on the machines (can be Windows or Linux of images) and we can launch them in every availability zone. We can also run **Amazon Machine Images (AMIs)** on a VM, it is an optimized version of the available images for the Amazon HW. For each instance we can choose different parameters:

- **RAM, CPU, Storage, Network performance**.
- **Type category**, for general purpose, compute optimized, memory optimized, etc...
- **Network configurations**, such as the VPC, subnet, public IP, port forwarding, etc...
- **User scripts**, run scripts to customize the VM.
- **Assign IAM role**, if the VM have to interact with other AWS services.

Notice that we can attach several type of storage to each instance:

- **Amazon EBS**, persist file system.
- **EC2 Instance Store**, file system that will be deleted when the machine is stopped.
- **Amazon S3**, AWS service used to store each type of data.
- **Amazon Elastic File System (EFS)**, file system which scales with the VM.

When a **new instance is started, a key pair is generated to access it**. The public key is stored on Amazon and the private key is stored by us. We can hybernate an instance and keep the state persistent without stopping it, this can be done only on instances that uses EFS. We can monitor the EC2 instances status through the **Amazon CloudWatch**, is a free service that shows the metrics for all the instaces and the graphs related to them. These metrics are collected every 5 minutes, if we want a more fine-grained monitoring we have to pay.

6.9.2 Amazon ECS

Elastic Container Service is used to automatically manage and scale containerized applications. It orchestrate the Docker containers, removing the infrastracture management. We have **ECS clusters** composed by several EC2 instances, each of them runs multiple containers. We have to define **task definition** that defines how many containers form our application. **We can choose to manage the whole infrastracture** (Docker engine and VM guest operating system) **or not**.

6.9.3 Amazon EKS

Elastic Kubernetes Services, is a **managed Kubernetes service that makes it easy for you to run Kubernetes on AWS** without needing to install, operate, and maintain your own Kubernetes control plane. **It manages the availability and scalability of the cluster nodes that are responsible for starting and stopping containers**, scheduling containers on virtual machines, storing cluster data, and other tasks. **Amazon ECR is a fully managed Docker container registry** that makes it easy for developers to store, manage, and deploy Docker container images.

6.9.4 Amazon Lambda

AWS Lambda is an event-driven, serverless compute service. Lambda enables you to run code without provisioning or managing servers. We create a **Lambda function** that is a code written in any supported programming language. It can be triggered by events such as CloudWatch, events or external requests, that sends an asynchronous request to it.

6.9.5 AWS Elastic Beanstalk

AWS Elastic Beanstalk is another AWS compute service option. It **is a PaaS that facilitates the quick deployment, scaling, and management of your web applications and services**. We upload your code and Elastic Beanstalk automatically handles the deployment. It supports different programming languages. Elastic Beanstalk is fast and simple to start using and enables the developers to be more focused on the application rather than the infrastracture management.

6.10 Storage

6.10.1 Amazon EBS

Amazon EBS **provides persistent block storage volumes** for use with Amazon EC2 instances. Each Amazon EBS volume is automatically replicated within its Availability Zone to protect you from component failure. It is designed for **high availability and durability and low-latency performance**. We can either choose to store file in **blocks or object**, blocks are more efficient and have an higher throughput but is more expensive. There are 4 types of Amazon EBS storages:

- **SSD General purpose**, recommended for most workloads. Is low-latency and suitable for as system boot volume.
- **SSD Provisioned IOPS**, used for large databases that need high performance.
- **HDD Throughput optimized**, used for big data and log processing. Fast throughput at low price.

- **HDD Cold**, used for infrequent access, is very cheap.

We can create **snapshots** of the systems that can be used to recreate a volume in that state. Each snapshot has a cost since must be kept persistently. Typically the EBS storages are charged based on the amount of GBs or operations done. **Inbound data transfer is free, only the outbound transfers across regions have a cost**.

6.10.2 Amazon S3

Amazon S3 is object-level storage, which means that if you want to change a part of a file, you must make the change and then re-upload the entire modified file. Amazon S3 stores data as objects inside set of resources that are called **buckets**. Designed for high durability, bucket names must be universal across the all existing names. Amazon S3 also scales by handling the buckets based on the object fitted inside. Amazon S3 also provides **low-latency access to the data using HTTP, so you access them anywhere and anytime**. We have different types of S3:

- **Standard S3**, designed for high durability, availability, and performance object storage for frequently accessed data.
- **S3 Intelligent-Tiering**, designed to optimize costs by automatically moving data to the most cost-effective access tier. Only a small fee is paid for monitor each resources.
- **S3 Standard-Infrequent Access**, used for data that is accessed less frequently, but requires rapid access when needed. Low price.
- **S3 One Zone-Infrequent Access**, same as before but the data are replicated only in one availability zone.
- **S3 Glacier**, secure, durable, and low-cost storage with very low performances.
- **S3 Glacier Deep Archive**, lowest-cost storage for data that might be accessed once or twice in a year. We can move the unaccessed file from a standard S3 solution to the deep archive and then delete them if needed. We pay based on the upload and retrieval requests. The data are encrypted automatically and the keys are managed by AWS.

You based on the GBs per month, traffict that is out form the specified region and for requests.

6.10.3 Amazon EFS

Elastic File System, provides **simple, scalable, elastic file storage** for use with AWS services and on-premises resources. **Amazon EFS is built to dynamically scale on demand**, grows and shrinks automatically as you add and remove files. Works well with large amount of data and can be accessed by multiple EC2 instances in the same VPC simultaneously.

6.11 Load Balancing

Means distributing incoming application or traffic across multiple targets in a single or multiple availability zones. There are different type of load balancer:

- **Application Load Balancer**, load balancing of HTTP traffic at application layer. It routes the traffic based on content of request, provides advanced request routing
- **Network Load Balancer**
- **Classic Load Balancer (previous generation)**

With Application Load Balancers and Network Load Balancers, you register targets in target groups, and route traffic to the target groups. Instead with classic load balancers, you register intefaces with the load balancer. We should use elastic load balancing solution if we want to **achieve high availability and better fault tolerance for the applications, this helps also to automatically scale them**. We can also automatically load balance our containerized applications. Furthermore, through the load balancer we can access Lambda functions through HTTP requests. Can use Amazon Cloud Watch to very is the system is performing as expected and eventually perform actions to keep the metrics as acceptable. Through the load balancers we can obtain the **access logs**, detailed information about requests sent to the balancer. You can use AWS CloudTrail to capture detailed information about the calls that were made to the Elastic Load Balancing.

6.11.1 Amazon Cloud Watch

Is a service used to monitor the performances of our applications and services. We can also see how much is used the application and eventually resize it in order to save money. **We can use both standard and custom metrics** and set up alarms (based on thresholds or metric math expressions) over them to perform autoscaling or send notificaitons. For each service we have different **namespaces** with different metrics that can be used in the alarms.

6.11.2 EC2 Autoscaling

Scaling is the ability to increase or decrease the compute capacity of your application. We can resize the EC2 instances based on the needs, if we need less capacities we can reduce them saving money. We can increase the capacities also to ensure availability. An **autoscaling group** is a set of EC2 instances that can be allocated or deallocated specifying the minimum and maximum size. We have to define a **launch configuration** and the start the autoscaling groupd, as last phase we have to specify the **autoscaling policy** (such as manual scaling, scheduled scaling, dynamic scaling, predicting scaling, etc...) to scale in/out.