1. **What is Virtualization?**

Virtualization is a concept of abstracting or creating a virtual device such as hardware, storage and memory and even OS. In the context of cloud computing, it means running a guest virtual machine on some underlying physical hardware: the host machine.

* For example, AWS offers VMs

1. **What is hypervisor?**

Hypervisor is a software that is required to create and run multiple VM’s in an Server. Also called as VMM (virtual machine monitor). A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine.

1. **3 goals of virtualization: (**Popek and Goldberg)

Fidelity – The device/program running on the VMM should have the identical functionality same as the machine running directly.

Performance – The VMM should be in complete control of the virtualized resources.

Safety – machine instruction should be executed without the VMM intervention.

1. **Instruction Architecture sets:**

**Privileged instructions**

Those that [trap](https://en.wikipedia.org/wiki/Trap_(computing)) if the processor is in [user mode](https://en.wikipedia.org/wiki/User_mode) and do not trap if it is in system mode ([supervisor mode](https://en.wikipedia.org/wiki/Supervisor_mode)).

**Control sensitive instructions**

Those that attempt to change the configuration of resources in the system.

**Behavior sensitive instructions**

Those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode).

1. **2 types of hypervisor:**
2. These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems. For this reason, they are sometimes called [bare metal](https://en.wikipedia.org/wiki/Bare_machine) hypervisors. Example:ZEN
3. These hypervisors run on a conventional operating system (OS) just as other computer programs do. A guest operating system runs as a [process](https://en.wikipedia.org/wiki/Computer_process) on the host. Example: Vmware, Virtual Box.
4. **Types of Virtualization:**
5. Full Virtualization: Here the guest OS is not aware that is has been virtualized and running on an VM. So, the host OS virtualizes the Hardware and so the command provided by the guest OS can be accessed here.
6. Para Virtualization: Here the Guest OS is aware that it is running on the virtualised environment and it is a guest. So, instead of issuing hardware command it directly commands the host OS.
7. **Xen hypervisor:**

Xen is a hypervisor that enables the simultaneous creation, execution and management of multiple virtual machines on one physical computer.

Xen was developed by XenSource, which was purchased by Citrix Systems in 2007. Xen was first released in 2003. It is an open source hypervisor. It also comes in an enterprise version.

Xen is an open Source created in 2003. Xen is primarily a bare-metal, type-1 hypervisor that can be directly installed on computer hardware without the need for a host operating system. Because it's a type-1 hypervisor, Xen controls, monitors and manages the hardware, peripheral and I/O resources directly. Guest virtual machines request Xen to provision any resource and must install Xen virtual device drivers to access hardware components. Xen supports multiple instances of the same or different operating systems with native support for most operating systems, including Windows and Linux. Moreover, Xen can be used on x86, IA-32 and ARM processor architecture.

**Domain 0 Guest  
o Domain Management and Control (Xen DM&C) • Domain U Guest (Dom U)  
o PV Guest  
o HVM Guest**

*Domain 0* Domain 0, a modified Linux kernel, is a unique virtual machine running on the Xen hypervisor that has special rights to access physical I/O resources as well as interact with the other virtual machines (Domain U: PV and HVM Guests) running on the system. All Xen virtualization environments require Domain 0 to be running before any other virtual machines can be started. Two drivers are included in Domain 0 to support network and local disk requests from Domain U PV and HVM Guests (see below); the Network Backend Driver and the Block Backend Driver. The Network Backend Driver communicates directly with the local networking hardware to process all virtual machines requests coming from the Domain U guests. The Block Backend Driver communicates with the local storage disk to read and write data from the drive based upon Domain U requests.

*Domain U* DomainU guests have no direct access to physical hardware on the machine as a Domain0 Guest does and is often referred to as unprivileged. All paravirtualized virtual machines running on a Xen hypervisor are referred to as Domain U PV Guests and are modified Linux operating systems, Solaris, FreeBSD, and other UNIX operating systems. All fully virtualized machines running on a Xen hypervisor are referred to as Domain U HVM Guests and run standard Windows or any other unchanged operating system. The Domain U PV Guest virtual machine is aware that it does not have direct access to the hardware and recognizes that other virtual machines are running on the same machine. The Domain U HVM Guest virtual machine is not aware that it is sharing processing time on the hardware and that other virtual machines are present. A Domain U PV Guest contains two drivers for network and disk access, PV Network Driver and PV Block Driver.

1. **Containerization:**

**Light weight Operating-system-level virtualization**, also known as **containerization**, refers to an [operating system](https://en.wikipedia.org/wiki/Operating_system) feature in which the [kernel](https://en.wikipedia.org/wiki/Kernel_(computer_science)) allows the existence of multiple isolated [user-space](https://en.wikipedia.org/wiki/User-space) instances. Such instances, called **containers** may look like real computers from the point of view of programs running in them. A computer program running on an ordinary operating system can see all resources (connected devices, files and folders, [network shares](https://en.wikipedia.org/wiki/Shared_resource), CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container. It uses the same OS for every an application to run on guest OS instead of installing each and every time.

1. **Docker:**

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don’t need the extra load of a hypervisor but run directly within the host machine’s kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines.

Docker provides tooling and a platform to manage the lifecycle of your containers:

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

## Docker architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

### **The Docker daemon**

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

### **The Docker client**

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

### **Docker registries**

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

### **Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

#### **IMAGES**

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

#### **CONTAINERS**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

1. **Similarities and differences:**

Similarities – Both emulate compute infrastructure – Both encapsulate the tenant • Differences – VMs provide “hardware level virtualisation” vs containers ‘operating system level virtualisation” – VMs need 10s of seconds for provisioning vs containers need milliseconds provisioning – Virtualisation performance is slower than containers in most dimensions except for networking – VM tenants are fully isolated vs containers provide process level isolation to tenants

1. **Example: run nginx on OSX : docker run -d -P --name web nginx**
2. pull image ‘nginx’ from docker hub • 2. start container from image, name ‘web’ • 3. forward all ports (-P) • 4. run as daemon (-d)
3. **Container Orchestration:**

Motivation:

Running Containers at scale requires management tools • Fault tolerance • Auto-scaling on demand • Manage accessibility from the web • Update/rollback without downtime

1. **Kubernetes:**

### is an open-source system for automating deployment, scaling, and management of containerized applications.

SERVERLESS:

1. **Serverless AWS Definition:**

“Serverless most often refers to serverless applications. Serverless applications are ones that don't require you to provision or manage any servers. You can focus on your core product and business logic instead of responsibilities like operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability. By building your application on a serverless platform, the platform manages these responsibilities for you.”

1. **Serverless deployment Abstraction:**
2. Bare metal servers: Lenovo, Dell, build your own
3. Virtual Private Server: Linode, OVH
4. Iaas: AWS, Google cloud platform, azure
5. Paas: Heroku, Phtyon everywhere, code star
6. Server less computing: Lambda, Azure
7. **Serverless architecture Layers:**

Compute – Function as a Service (FaaS): AWS Lambda, Google Cloud Functions, Azure Functions

• Data – AWS S3 / DynamoDB / Aurora, Google Firebase, Fauna

• Messaging – AWS SNS / SQS, PubNub

• User Management and Identity – Okta, Auth0, AWS Cognito

• Monitoring and Deployment – AWS CloudWatch, DataDog

• Edge – AWS API Gateway / CloudFront, Netlify

1. **4 pillars of server less:**
2. Scaling
3. No Server Management
4. High availability
5. Never pay for idle
6. **Lambda Services:**

* AWS Lambda is a compute service that lets you run code without provisioning or managing servers.
* AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second.
* You pay only for the compute time you consume - there is no charge when your code is not running.
* With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration.
* All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js, Java, C#, Go and Python).
* You can use AWS Lambda to run your code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway; or invoke your code using API calls made using AWS SDKs. With these capabilities, you can use Lambda to easily build data processing triggers for AWS services like Amazon S3 and Amazon DynamoDB, process streaming data stored in Kinesis, or create your own back end that operates at AWS scale, performance, and security.
* Architecture diagram

1. **Minimal Configuration: –**

Code (zip – max 50MB) – Memory (128MB – 3GB) – Timeout (max 300 secs) – Invocation Event Source or schedule – 2 sets of permissions: event source (invocation) & service (resource)

1. Stumbling performances:
2. **Performance limitations:**

* Cold starts, max execution time memory allocation, and strict resource limits)
* Cold Start Issue: Latency issue experience when an event is triggered. When there is no idle container to run the code, the user experience and also if an application has not been activated for longer period.
* Auto-provisioning and auto-scaling require dynamic provisioning – Issue: code that hasn’t been used for a while takes longer to start – Result: longer response latencies from serverless application
* Reasons for cold start, memory size, language choice, VPC, http calls.
* Statically typed languages Works bad in java and c# better in dynamically typed language Phyton and go lang, node js

1. **Vendor lock-in:** Trade off: deep integration vs agnostic architecture (open source)

* AWS server less ecosystem • Benefits – Rapid development – Easy integration – Cost-effective (low TCO) – Large community & support • Drawbacks – Single vendor dependency – Proprietary tooling & APIs
* Open source server less ecosysytem: Benefits – Open code & architecture – Portability – Heavy customisation possible – No vendor dependency • Drawbacks – Higher TCO (‘UDH’) • Specialists & admin • Updates & patches – Variable documentation & support – Harder integration

1. **Monitoring and debugging:** Challenging to troubleshoot provider-controlled resources
2. **Security and Privacy:** Multi-tenancy, opacity and resource reuse add extra challenges

* **Strength and weakness of serverless security:**

**Strengths** – OS patched and managed by platform – Attack surface area can be reduced • No long-running processes • Granular permissions can be assigned •

**Weaknesses** – App code still vulnerable to library dependencies – OWASP top 10 vulnerabilities still relevant • Injection, Cross-site scripting, Sensitive data exposure – Permissions are often too lax – Wider and more diverse ecosystem (microservices) • Increases possible surface area for attack

1. User cases:
2. Use Case #1: Event-driven data processing (resize uploaded images)
3. Use Case #2: Serverless Web Applications (simple 3-tier app)
4. Use Case #3: Mobile & IoT Apps (Airbnb smart home)
5. Use Case #4: Application Ecosystems (Alexa Skill)
6. Use Case #5: Event Workflow (image recognition & processing)

**SCALABLE SYSTEMS:**

1. **Horizontal Scaling / Vertical Scaling:**

* **Horizontal scaling means that you scale by adding more machines** into your pool of resources whereas **Vertical scaling means that you scale by adding more power (CPU, RAM) to an existing machine**.
* An easy way to remember this is to think of a machine on a server rack, we add more machines across the **horizontal** direction and add more resources to a machine in the **vertical** direction.
* In a database world horizontal-scaling is often based on the partitioning of the data i.e. each node contains only part of the data, in vertical-scaling the data resides on a single node and scaling is done through multi-core i.e. spreading the load between the CPU and RAM resources of that machine.
* With horizontal-scaling it is often easier to scale dynamically by adding more machines into the existing pool - Vertical-scaling is often limited to the capacity of a single machine, scaling beyond that capacity often involves downtime and comes with an upper limit.
* Good examples of horizontal scaling are Cassandra, MongoDB and a good example of vertical scaling is MySQL - Amazon RDS (The cloud version of MySQL). It provides an easy way to scale vertically by switching from small to bigger machines. This process often involves downtime.

1. **Horizontal Scaling Components:**

**X AXIS:** Horizontal Scaling, Scaling done by cloning of services and data.Each clone can do the work of all other clones. work is distributed among clones without bias. Easy to do and inefficient than y and Z axis.

**Y AXIS:** Functional decomposition i.e. Scaling achieved by splitting different functionalities. Costlier than X axis. Split up is necessary. Example, login browse checkout functionalities of customers.

**Z AXIS:** Data partioning i.e. Scaling achieved by splitting up same things. Costlier than the other two. Improves fault tolerance and cache memory.

1. **Definition Software Architecture:** Software architecture is a set of structures needed to reason about a system. It includes software components, the relations between them and properties of both.
2. **What is Napster:** Napster is an online music store owned by Best Buy. It was originally founded by Sean Parker and Shawn Fanning in 1999 as a free online peer-to-peer (P2P) file sharing service, which mainly focused on sharing MP3 audio files.
3. **What is Gnutella:** Gnutella is a file sharing [protoco](https://searchnetworking.techtarget.com/definition/protocol)l that defines the way distributed [nodes](https://searchnetworking.techtarget.com/definition/node) communicate over a [peer-to-peer](https://searchnetworking.techtarget.com/definition/peer-to-peer) (P2P) network. Like [Napster](https://searchcio.techtarget.com/definition/Napster), Gnutella is often used to share music files and has been an object of great concern within the music publishing industry.Unlike Napster, Gnutella is not a Web site; instead, it is a decentralized network in which users can see the files of a small number of other Gnutella network members, and they in turn can see the files of other network members, in a kind of daisy-chain effect. After installing and launching Gnutella, the user's computer becomes both a [client](https://searchenterprisedesktop.techtarget.com/definition/client) and a [server](https://whatis.techtarget.com/definition/server) in the network, which is called GnutellaNet. Gnutella allows network members to share any file type, whereas Napster is limited to [MP3](https://whatis.techtarget.com/definition/MP3-MPEG-1-Audio-Layer-3) music files.
4. **Architecture: Centralised** Index Unstructured Using simple flooding
5. Advantages- Napster + Highly efficient data lookup + Rapidly adapts to changes in network + Gnutella Entirely decentralized, pure P2P network + Highly resistant to failure.
6. Disadvantages – Napster Questionable scalability - Vulnerable to censorship, failure, attack – Gnutella: Search is time-consuming - Network typically scales poorly.
7. Four Main Decoupling Components:
8. Load Balancers
9. Message Queues
10. Message Topics
11. Service Registries

**MapReduce: Google’s underlying architecture**

MapReduce (Hadoop) •

Google File System (GFS) (HDFS)

• BigTable (Hbase)

• Spanner

• F1

MapReduce is a programming tool, utilized for generating and processing very larger data sets. In simple, a flexible data processing tool.

|  |  |
| --- | --- |
| map(String key, String value): // key: document name // value: document contents for each word w in value: EmitIntermediate(w,”1”); | Reduce(String key, Iterator values): // key: a word // values: a list of counts int result = 0; for each v in values: results += ParseInt(v); Emit(AsString(result)); |

**How it works:**

**MapReduce Fault Tolerance:**

1. In reality, instances of MapReduce might involve hundreds or thousands of worker  
   machines, and so normal failure is a significant issue.
2. Master pings every worker occasionally – if a worker does not respond before a  
   timeout, then the master marks the worker as being failed.
3. Because the results of any map tasks completed by the (now failed) worker may  
   still be on the local disk(s) of the failed worker, the master resets the failed server’s  
   list of map tasks as unallocated, and reschedules them onto other map worker  
   machines, replacing the failed machine.
4. All reduce workers that still need data from the failed map worker are notified of  
   the replacement map worker(s), and so the reducers switch to reading from the  
   local disks of those replacement map workers.
5. When a map task finishes, it sends a notification to the Master and includes names  
   of the local temporary files where the map outputs are: if the Master receives such  
   a notification for a task that has already been completed by a different worker, it  
   ignores it; otherwise it records the completion data.

**GFS: (Google File System)**

Google File System (GFS) is **a scalable distributed file system (DFS**) created by Google Inc. When Google started, it committed to building a huge scalable distributed storage capability using cheap commodity components (PCs/servers). GFS provides fault tolerance, reliability, scalability, **availability, • Constant monitoring • Error detection • Fault-tolerance • Automatic recovery** and performance to large networks and connected nodes. GFS is made up of several storage systems built from low-cost commodity hardware components**. It is optimized to accomodate Google's different data use and storage needs, such as its search engine, which generates huge amounts of data that must be stored.**

1. Files in GFS are divided into fixed-sized 64Mb blocks **called chunks.**
2. Each chunk is assigned a unique i.d. called a **chunk handle**.
3. Chunks are stored on **chunkservers.**
4. **Fault-tolerance** provided by replicating each chunk across **multiple servers** typically three replicas, but user-configurable **Meta-data needs** to be recorded so GFS knows what chunk is where.
5. GFS consists of three main component types:

• GFS Master Server (single) – **maintains all file system metadata**

• GFS Chunkserver (multiple)

• GFS Client (multiple)

Chunkserver & Client can run on same machine, but risky if client crashes

**GFS Architecture**

**Mutation: Write control refer the slides pls.**

**V1.0 HDFS: GFS**

**Name node – Master server**

**Data Node- Chunk server**

**Block-Chunk**

**New Features of V2.0**

NameNode High Availability (HA): the name-node master service is made more resilient by addition of automatic failover to a hot standby.

Snapshots: timestamped backups for disaster recovery and error-rollback.

Federation: horizontal scaling of namenodes: multiple independent namenodes instead of one+backup; the namenodes talk to a generic block-pool layer that further separates namespace from storage, allows cluster resources to be used more effectively – datanodes send heartbeats up to namenodes

**HDFS (3.0)**

Hadoop 3.0 announced in early 2018 but is still in alpha phase – Final list of new features may change before the beta release • HDFS3 has major investment in Erasure Coding – Provides the same level of fault-tolerance as traditional storage replication but with improvements in storage efficiency: the storage overhead for EC can be 50% in comparison to 200% for straightforward replication – A failed storage drive (whole or part) is an erasure • NB you know the erasure has happened because you have blank – not corrupted data – EC involves use of Forward Error Correction in the storage array • When written, data bits are encoded along with additional bits • When read back, data missing from erasures can be reconstructed from the extra bits • Can reduce/eliminate need to maintain n identical replicas – Simplest example: parity checksum – Real-world example: Reed-Solomon (the “classic”) – Development of commercial cloud-scale data storage has driven innovation.

**New SQL**