Cloud Computing

Hui Zeng *

Summer Semester 2021

*All notes are summarized from the lecture and tutorial materials provided by Prof. Michael Gerndt and his team. Images are retrieved from the lecture as well as tutorial slides.

Contents

Part I.
Introduction & Technologies

1. Introduction

Different IT-trends boosts the need for cloud computing:

- Outsourcing, either infrastructure or management
- IT as a service: pay per use
- Re-centralization of data: similar to data centers, cloud be provided as a central place for data storage.
- Resource sharing instead of over-provisioning: same resource can be used for multiple purposes
- Server consolidation: instead of having multiple physical servers, with each dedicated to a certain service, servers are virtualized and put on one/reduced number of physical machines.
- Scalable computing
- Application dynamism: amount of request on web changes over time.
- Green computing, big data, stream processing, IoT, machine learning, etc.

Cloud Computing the definition is mainly divided by

- ubiquitous, convenient, on-demand network access to a **shared pool of configurable computing resources** (eg: networks, servers, storage, applications, services)
- resources can be **rapidly provisioned** and released with **minimal management effort** or service provider interaction
- cloud model is composed of
  - 3 service models
  - 4 deployment models
  - 5 essential characteristics

1.1. 3 Service Models: IaaS, PaaS and SaaS

Three service models, ranking from outsourcing the least to the most: IaaS → PaaS → SaaS.

1.1.1. IaaS: Infrastructure as a Service

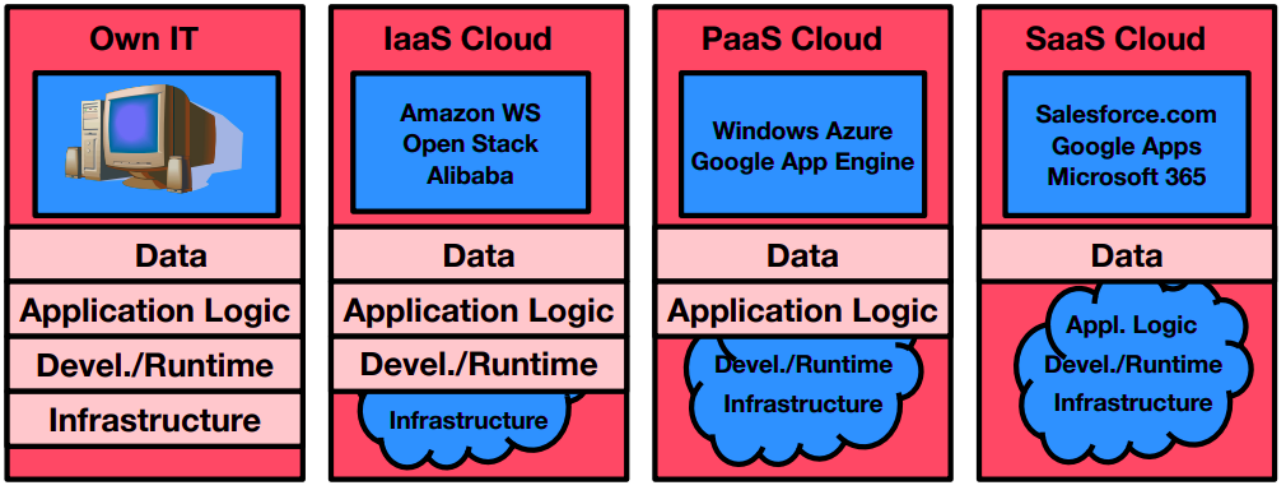
- Offering: provision processing, storage, networks, other fundamental computing resources
- Rights as consumer:
  - deploy and run **arbitrary** software, including **operating systems and applications**
  - control over OS, storage, deployed applications
  - limited control of select networking components
- No control as consumer:
  - underlying cloud infrastructure

1.1.2. PaaS: Platform as a Service

- Offering: application infrastructure services(eg: development platforms, libraries, tools, databases) through client interface
- Rights as consumer:
  - limited user-specific application configuration settings
- No control as consumer:
  - underlying cloud infrastructure
  - network, servers, storage, OS
  - individual application capabilities
- Example: MS Azure, Amazon FaaS, Google application engine

1.1.3. SaaS: Software as a Service

- Offering: provider's applications on cloud through client interface
- Rights as consumer:
  - limited user-specific application configuration settings
- No control as consumer:
  - underlying cloud infrastructure
  - network, servers, OS, storage
  - individual application capabilities



Service Model	IaaS	PaaS	SaaS
Service category	VM rental, online storage	online operating environment, online database, online message queues	application and software rental
Service customization	server template	Logic resource template	application template
Service provisioning	automation	automation	automation
Service accessing and using	remote console, web services	online development and debugging, integration of offline development tools and the cloud	Browser, web service interfaces, SDKs, apps

Service Model	IaaS	PaaS	SaaS
Service monitoring	physical resource monitoring	logic resource monitoring	application monitoring
Service level management	dynamic orchestration of physical resources	dynamic orchestration of logic resources	dynamic orchestration of applications
Service accounting	physical resource metering	logic resource usage metering	application usage metering
security	storage encryption and isolation, VM isolation, VLAN, SSL/SSH	data isolation, operating environment isolation, SSL	data isolation, application isolation, SSL, Web authentication and authorization

1.2. 4 Deployment Models: Private, Community, Public, Hybrid

- Private Cloud:
  - service offered **via private network** for **single client**.
- Community Cloud:
  - service offered to **a specific group of clients**.
- Public Cloud:
  - service offered **over Internet via Web-application** or third-party provider for **everyone**.
- Hybrid Cloud: combination of public and private cloud.

1.3. 5 Essential Characteristics

- **on-demand self-service**:
  - able to **provision computing capabilities** unilaterally(no interaction required with provider).
- **broad network access**:
  - capabilities can be available and accessed through by **diversely thin or thick client platforms** (mobile, tablets, cable, etc.)
- **resource pooling**:
  - **multi-tenant model** is used, multiple customers shares the computing capabilities at the same time, according to their self-customized demand. Specification of resource location can be possible at higher abstraction level.
- **rapid elasticity**:
  - computing capabilities can be **elastically provisioned and released** in any quantity at any time. The process can be automated or scaled according to dynamic demand.
- **measured service**:
  - automatically control and optimize resource use by **leveraging a metering capability**. Resource usage can be monitored, controlled and reported.

1.4. Pros & Cons of Clouds

- Advantages:
  - scalability, elasticity
  - rapid deployment
  - no capital investment for physical resources
  - outsourcing of infrastructure management

  - limited access to on-premise servers
  - fault tolerance: multiple servers have data replicas, if one node fails, other nodes will replace.
  - collaboration

- Disadvantages:
  - no control over security, based on "trust".
  - no control over hardware/infrastructure
  - vendor lockin: service is not standardized, not compatible to other vendors.
  - cost on monthly fees: if demand for same computational power is constant, fee may be higher than building own hardware. Only recommendable for dynamic demand.
  - breaking SLAs: your performance may be influenced by other tenants(multi-tenant model).

2. Base Technologies of Clouds

2.1. Process Technology

Production the processors are produced from semi-conductor materials. It's primarily produced on a **waver** consisting of a lot of chips. Later the waver is cut after the production process. Individual chips will be packaged into the system.

Transistor

- traditional 2D planar transistor: a 2D-planar structure, the gate controls how much current flows from source through the drain.
- 3D tri-gate transistor **FinFET**: conducting channels on 3 sides with a **vertical fin structure**. The width of a Fin is **10nm**, and it keeps shrinking.

The smaller the structure gets, the more transistors fit on the same space, the faster the transistor gets.

2.2. Processor Architecture

2.2.1. CPU

- current state of CPU development:
  - increase in transistors: up to 10nm or even 5nm.
  - stop in increase of clock speed: up to 4GHz. Limitation: cooling. (the faster the clock speed, the more energy consumed, the hotter)
  - continuous need of performance improvement: parallelism on the chip, since halt in clock speed.
- trends in CPU development:

  - multi-core processors: parallelism
  - SIMD support (Single Instruction, Multiple Data): parallelism inside of a single instruction, computation of vectors of values in parallel.
  - combination of core private and shared caches:
    * data saved in cache for repeated operations
    * with multiple caches, cores can communicate. However, this may disturb the usage of cache.
  - hardware support for energy control: **dynamic voltage and frequency scaling**
    * chips work in a dynamic frequency controlled by hardware according to the need of the running software.
    * It checks whether operations are memory-bound or compute-bound.
  - 64-bit architectures

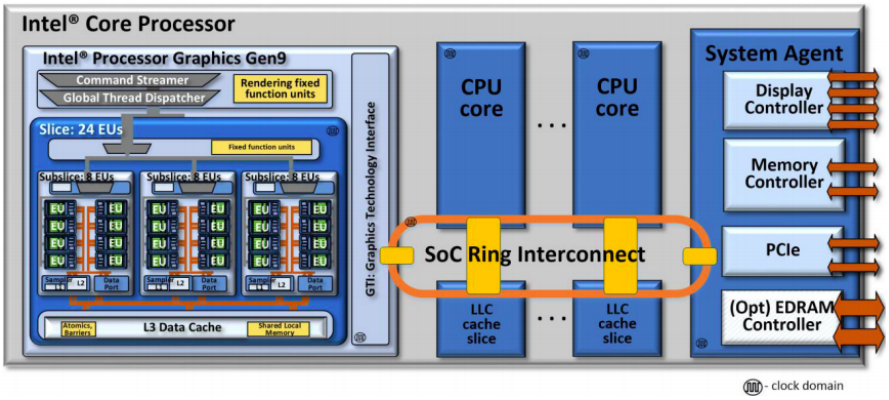
- challenge: the **memory hierarchy**
  - access to the main memory is slow, involving several hundreds of cycles for CPU, while each of these cycles can be responsible for multiple operations.

    → save such cycles for accessing data but keep the **data near to execution**.
  - **Level-1 Instruction & Data Cache**: fastest, but too slow to feed all data in large size. (64 Bytes: 8 double precision values, 64 bit long values)
  - **Level-2 Unified Cache**: not separated, access better.
  - **Translation Lookaside Buffer (TLB)**: translates virtual addresses into physical addresses and loads into main memory, it only stores the **most recent translation**, no need to lookup constantly.

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way
All caches use 64-byte lines			

2.2.2. Skylake Architecture

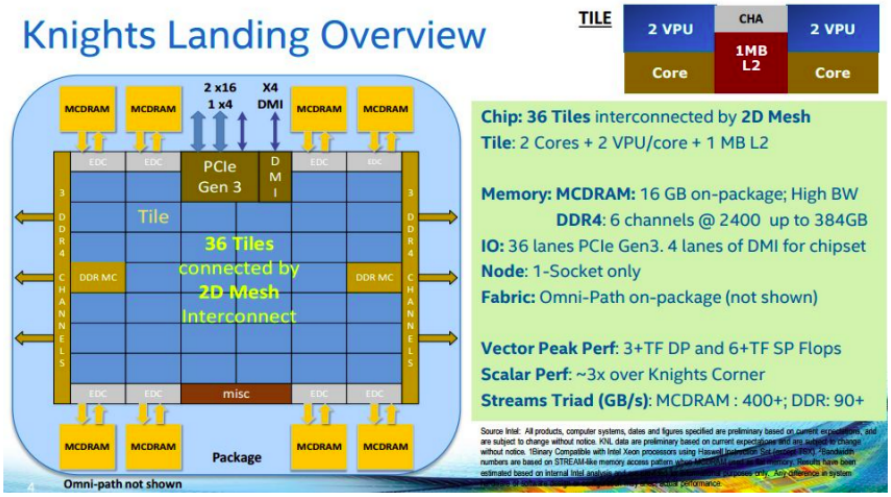


The archiecture of a processor (one chip)

- graphic processor: accelerator for specified computation
- system agent: support structures (eg: display controller, memory controller, PCIe for I/O, EDRAM controller)
- CPU cores: homogeneous cores with a **private cache each**.
- LLC cache slice: each slice cache is associated to each CPU core.
  - If core misses information in its private cache: through interconnect, it checks which slice cache contains the info, then it propagates to the cache associated the CPU core and returns the info back to the CPU.
- SoC Ring interconnect: all parts are connected by a ring bust.
  - if CPU writes to I/O device – PCIe, it first puts information onto the bust, then it propagates into the PCIe and is written to the disk.

→ the **interconnect ring** is the **bottleneck** for increasing the cores.
→ alternatives: Xeon Phi

2.2.3. Xeon Phi Architecture

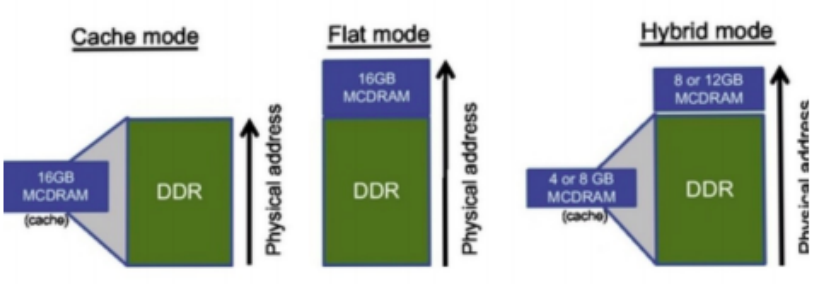


- Goal: allows significantly **more cores** in a single processor, in a single CPU die.
- Idea:
  - the die is organized into a **tile-architecture**.
  - 36 compute tiles are connected through a **2D mesh network** → connection between tiles in both x- and y-direction.
  - each tile has 2 cores → **72 cores** in total
- Tile structure:
  - 2 cores
  - 2 VPUs (Vector Processing Unit) for each core → 4 in total per tile.
  - L2 cache: shared between the cores, but as a private cache for each tile.

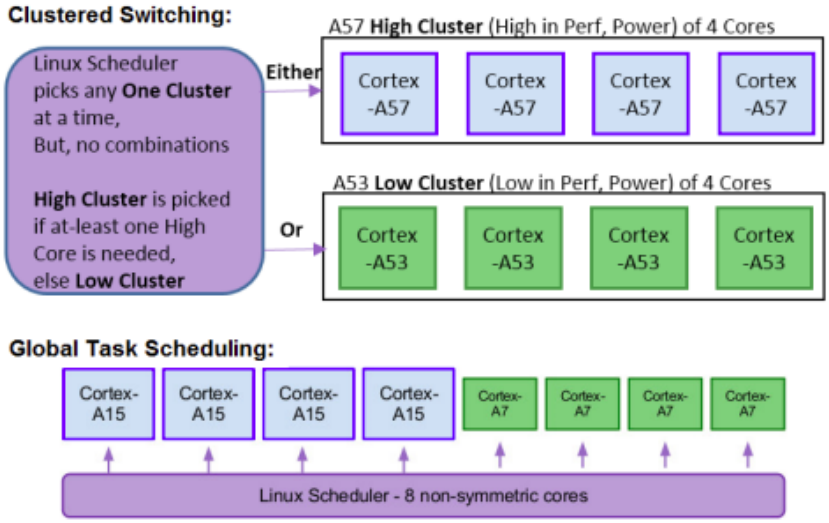
    → multiple copies of an address can be in different private L2 caches of different tiles, which **must be coherent**.
  - CHA (Caching Hold Agent): responsible for the **coherence**. It's connected to each tile, keeping track of the status of the copies by implementing a **coherence protocol**.
- Memory:
  - DDR4 memory
  - **MCDRAM**– Multi-channel DRAM, **high bandwidth**(450 GB/s)

    Memory modes:
    - flat mode: all MCDRAM is used as **physical address**. Data structure can explicitly choose between MCDRAM or DDR.
    - cache mode: all MCDARM is used as **L3 cache**. physical addresses only on DDR. If data is being processed from DDR, it's mapped to MCDRAM.

  * hybrid mode: combination of flat and cache mode. Part of MCDRAM is used as **L3 cache**, the other as **physical addresses**.



2.2.4. Processors for mobile devices: ARM



- **Big Little Principle**
  - Combination of **high clusters** and **low clusters**, controlled by clustered switching.
    * high cluster: high in performance
    * low cluster: low in performance, but energy efficient
    * only **one cluster** at a time, **no combinations**.
  - Global task scheduling: tasks are scheduled according to the requirement between 2 clusters.
- Use-cases: Apple Processor A14 (2 high performance Firestorm, 2 energy-efficient Icestorm)

2.3. Accelerator Programming

- Motivation:

  - increase computational speed and reduce energy consumption

    → achieved by **specialization** in operations/on-chip communication/memory accesses

    → **accelerator**

- Types:
  - GPGPU (General Purpose Graphic Processors)
  - FPGA
  - standard cores
- Designs:
  - CPU with accelerators attached: computation can be offloaded onto the accelerator.
  - accelerators-only design
  - accelerator booster: a collection of accelerators as a separate part from the whole system. Jobs can be computed by these accelerators when necessary. Accelerator booster can be shared among parallel jobs.

2.3.1. Graphic Processing Units (GPU)

- Usage:
  - visualization
  - **general processing** (NVIDIA)
- Parallelism: multi-threading, MIMD, SIMD
- Challenges:
  - a **specialized programming interface** for the GPGPU needed (eg: CUDA from NVIDIA)
  - **scheduling coordination** on system processor and GPU
  - **transfer of data** between system memory and GPU memory
- example: NVIDIA Tesla P100

NVIDIA Tesla P100

- GP100 (GPU):
  - L2 cache: shared among all compute units – streaming multi-processor
  - NVLink: able to connect multiple GPGPUs together
  - memory controller: access to high bandwidth memory
  - 6 Graphic Processing Clusters(GPC)
    * 10 Streaming Multi-Processor each GPC, 60 in total

    - 5 Textural Processing Clusters (TPC), 1 for 2 SM, 30 in total

- High Bandwidth Memory (HBM)
  - **vertical stacks** of memory dies connected by microscopic wires

    → near and tight connection between memories
  - 180 GB/s per stack bandwidth

→ good for data parallel processing like vector processing.

2.3.2. Field Programmable Gate Arrays (FPGA)

- hardware which is programmable
- Consist of:
  - **array of logic gates** to implement **hardware-programmed special functions**
  - **specialized functional units** (eg: signal processors, multipliers)
  - static **memory**
- programmed in VHDL: the program describe functions to be executed, it will then be translated into look-up tables, which are put into the logic gates
- Use-case:
  - as **accelerator** for specialized computations
  - **filtering** for databases
  - as **switches, routers** for communication
  - **preprocessing**: I/O hardware of FPGA accesses the DDR, local computation can be organized in the pipeline of FPGA. Local preprocessing will be done on FPGA and the output will be sent ot external processor via PCIe.
- examples: Altera, Xilinx

2.4. Architecture for Parallelism: Shared Memory Systems

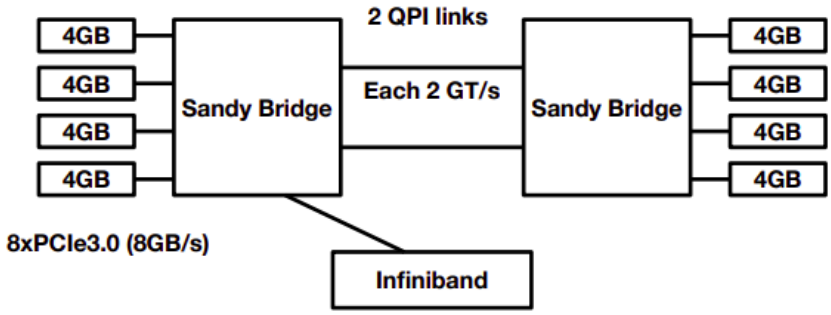
- Idea: 2 architectures to **achieve parallelism**, which is combining multiple processors together for computation.
  - shared memory system
  - distributed memory system
- **Non-Uniform Memory Access (NUMA)**:
  - **multiple CPU** with multiple cores are connected through **single physical address** space.

    → access time depends on the **distance to the physical address** (memory)

    → CPU accesses either local(near) or remote memory → locate the memory near for fast access.

Example: SuperMUC

- 2 multi-core processors(Sandy Bridge) with 32GB memory in total
- each processor can access to all 32GB memory, **however access time differs**
- **Latency**
  - local: ∼50ns, ∼135 cycles
  - remote: ∼90ns, ∼240 cycles



- Programming interfaces for Shared Memory Systems
  - explicit threading
  - automatic parallelization: sequential code is given to compiler, which **automatically** parallelize the work among available CPUs.
  - OpenMP: directive-based parallel programming, parallel computations are **explicitly expressed**.
- Challenges in parallel computing:
  - explicit synchronization needed.
  - **cocurrency bugs**: the outcome of the computation depends on the speed of access to the memory → non-deterministic results possible.
  - control of **data locality**

2.5. Architecture for Parallelism: Distributed Memory Systems

- Characteristics
  - **Coupling** of individual nodes via network: processor only have access to the memory in node.
  - **no shared** physical address space
  - communication between nodes: transfer of **messages**
- Programming in Distributed Memory Systems: **more difficult** than shared memory systems.
  - + : **rare race conditions** → cocurrency bugs low.

  - – : Message Passing Interface(MPI), have to explicitly decompose or insert message passing → more difficult to program.
  - Process to Process communication and collective operations

- Challenges:
  - **more difficult** to programm than shared memory systems
  - **expensive communication**, much slower than access to memory.

$$t(message) = \text{startup time} + \frac{\text{message size}}{\text{bandwidth}}$$

    * communication with one large message is more efficient than multiple small messages (startup time)
    * mapping onto processors has performance impact. Communication may not need to go through the entire but locally. (eg: 2D mesh network)

2.6. Data Center Networks

Inside the data center, the **servers** are connected by **LAN**– Local Area Network. Each **server** is connected to a **switch**. The **switches** are connected, which allows **communication between the servers**.

Multiple **LANs** are possible in a data center. The **LANs** can be connected through a **router**. A router is based on a IP-address, it can make decisions depending on the IP-address (eg: receiver of message).

The **routers** in the data center can be connected to a **WAN**– Wide Area Network, the internet. The **WAN** can be connected to a **local router**, which a client has access to . This is the **last mile** to reach the client, frequently **radio network** is used – WLAN or GSM.

2.6.1. Different Networks

- Types:
  - WAN – Wide Area Networks
    * homogeneous base technology (opto-electronic)
  - LAN – Local Area Networks and Cluster Networks
    * non-shared Ethernet
    * Infiniband
  - Last Mile
    * heterogeneous base technology (Radio, TV cables, etc.)

- Performance Metrics:
  - **Latency**: transport time of a message

    * physical delay: time needed to go through the links, limited by speed of light, **not optimizable**.
    * protocol delay: time needed to execute protocol operations. **compensated by increase of CPU performance.**
    * line waiting time: time to wait until the link is available. negligible up to 10% utilization. **reduced by increasing bandwidth**.
    * transmission time: time needed to send certain amount of data over the link. **reduced by increasing bandwidth**.

$$\text{transmission time} = \frac{\text{message size}}{bandwidth}$$

  - **Bandwidth** (byte/sec): the speed transporting a message

2.6.2. Local Area Network

Ethernet

- first implementation based on a **shared cable**: all computer are connected through one cable. Only one computer can transmit message at a time.
- now **switched Ethernet**: each computer is connected to a switch. Switches are connected together to enable communication. A switch replicates all packets to all ports.
- Speed: 10 Mbit/s, 100 Mbit/s or 1000 Mbit/s

VLAN – Virtual Local Area Network

- Characteristics:
  - a single LAN is **partitioned** into **multiple virtual LANs**.
    $\rightarrow$ direct traffic or for security reasons.
  - each virtual LAN is a **single broadcast domain**
  - **communication** between VLANs only through **router**
- **Port-based VLAN**
  - The **ports** of a switch are **specifically assigned** to a **VLAN**.
  - servers of a VLAN from two switches are communicated through a **link**.
  - # links = # VLANs
- **Tagged VLAN**
  - one port of a switch is not connected to server. This port is connected to other switches through a **link**. This link manages all packets.
  - **Tag**: packets are **only forwarded to ports with the same tag**.
  - # links = 1

Infiniband

- Characteristics:
  - low latency, high bandwidth
  - speed: 25 Gbit/s
  - RDMA access: **direct access of memory of other computer**. Instead of packing data into a message an sending to the operation system and then taking it out, RDMA can **directly fetch** the data from memory and forward it to the requester.

    No protocol overhead or handling of message → **faster**.
  - based on **Virtual Interface Adapter**: data transfer don't require operation system support.
- Use-case: clusters and servers

2.6.3. Software-defined Networks

- Motivation:
  - higher speed
  - automated network configurations
  - security
  - adaptation to performance variations
- Current network management:
  - Management plane –> Control plane –> Data/Forwarding plane
- Software Defined Networking:
  - allows network administrators to **programmatically initialize, control, change and manage the network behavior dynamically** via open interfaces
  - Protocol: OpenFlow
    - enables an open interface to **interact** with networking devices (machines, switches from different providers)
    - network layers on top of L3
    - SDN controllers communicate to L3 switches using openFlow protocols.

2.6.4. Last Mile Networks

Wireless Local Area Network – WLAN

- consists of **clients** and **access points** as routers.
- modes:
  - **infrastructure**: clients connect to the access point

  - **ad hoc**: clients communicate with each other

- Security:
  - Wireless Equivalent Privacy (WEP)
  - Wi-Fi Protected Access (WPA1, WPA2)

- Speed:
  - 802.11 n: 800 Mbit/s, 70m
  - 802.11 ac: 1733 Mbit/s, 35m

Digital Subscriber Line – DSL

- transmission of data **over telephone lines**, share with telephone service (different frequency)

- Speed:
  - asymmetric DSL: upstream bandwidth **much lower** than downstream
  - downstream: 256 Kbit/s to 100 Mbit/s

Very-high-bit-rate Digital Subscriber Line – VDSL

- higher speed than DSL:
  - different versions: VDSL, VDSL2, VDSL2-Vplus
  - VDSL: up to 55Mbit/s downstream, 16 Mbit/s upstream

- VDSL with **vectoring**: reduce crosstalk between different lines. (Crosstalk: communication on one line influences the communciation on other lines)

  → special encoding of neighbouring lines: a provider needs to have **access to all lines in a bundle**

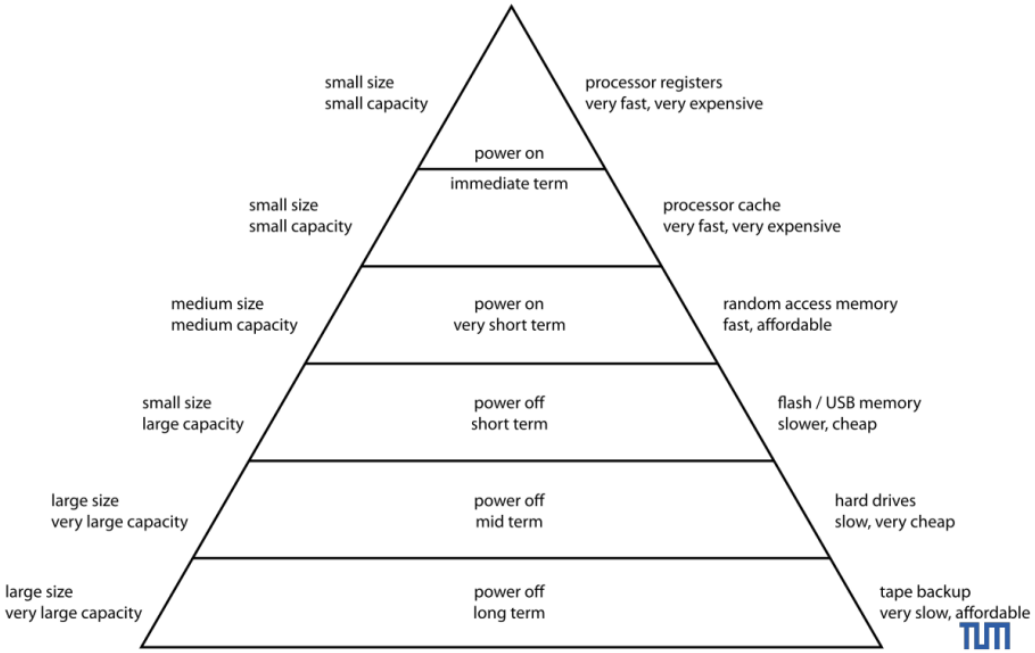
  → current implementation difficult

Global System for Mobile Communication –GSM

- network for **mobile phones**, 3G, 4G, 5G

- access to the network using SIM card

2.7. Storage Technologies



2.7.1. Local Storage

Redundant Array of Independent Disks – RAID

- Goal: increase reliability or bandwidth
- RAID 0: **distribute** blocks over **2 disks**, if **write both blocks on two disks** at the same time, it gets **double bandwidth** to the disk.

  → **higher bandwidth**
- RAID 1: **replicates** blocks on **2 disks**. If one fails, we still have the information on the other disk.

  → **higher reliability**
- RAID 5: **distribute** blocks over **4 disks**, while one disk saves the **parity information**. If one block fails, the parity information enables **reconstruction**. Parity information of blocks is not written on the same disk, but **distributed** over all disks.

  → **higher bandwidth and reliability**

Flash

- non-volatile memory, retains stored information even after power is removed.
- Write operation: tunnel injection, a high positive voltage between control gate and source **pushes electrons into the floating gate**. It stays/saves in the floating gate as stored information.

- Read operation: a higher voltage is required at the drain to make the channel conduct, the electrons move from source to drain.
- Increasing storage density: **increase floating gates** in a flash cell
  - Single Level Cells (SLC): stores **1 bit** of information
  - Multiple Level Cells (MLC): stores **2 bits** of information
  - # floating gates ↑, cost per bit ↓, storage density ↑, program-erase cycles ↓, write/reading speed ↓
- Use-case: USB-disks, SD-cards, mobile phone storage, built in SSD

SSD

- Comparison with hard disks:
  - lower latency, random access
  - smaller storage capacity
  - less power hungry
  - faster read/write speed

2.7.2. Data Center Storage

- storage comparison: €/IOPS
- only based on flash storage or SSDs, combined with RAID, with special controllers optimized for SSDs.

  → IOPS ↑, latency ↓, bandwidth ↑, cost ↑

2.7.3. Provisioning of Storage

- 3 ways to provide storage:
  - Direct Attached Storage: storage devices are attached to the individual computer
  - Storage Area Network
  - Network Attached Storage

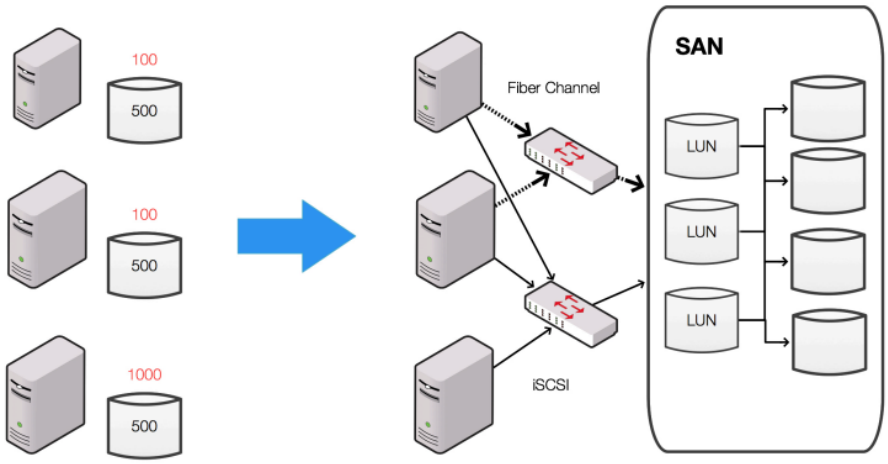
Storage Area Network – SAN

- access to **block level** data storage
- a **specialized network** connecting the servers which separates from LAN
- **no pre-existing file system**, the server can define its own file system according to needs
- a shared pool of spare resources, which allows **flexible allocation of spare storage**
- Advantages:

- **flexible distribution** of devices between clients, reconfiguration of distribution in software instead of adaptation of cabling.
- easy replacement of faulty servers
- back-up can be done centrally, easier disaster protection
- no pre-existing file system, allows **customization** according to needs.

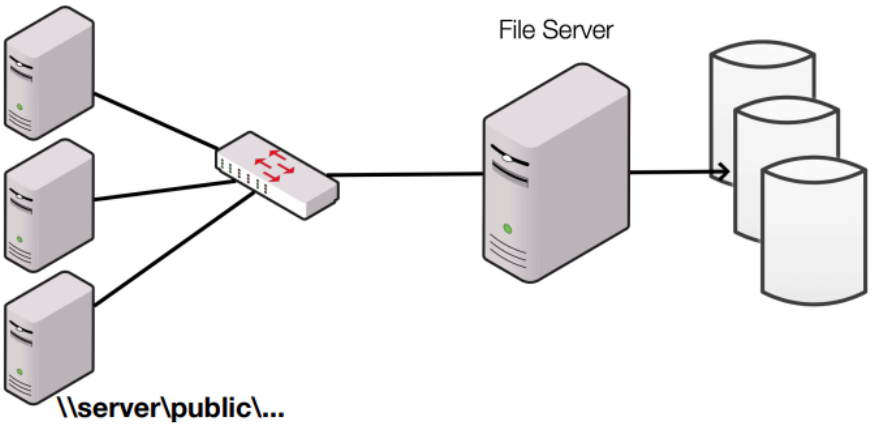
Disadvantages:

- shared network bandwidth
- shared performance of storage devices



Network Attached Storage – NAS

- storage devices(disks) are connected to a **file server**
- computers **go over the network** and **access the file system** on file server
- an **existing file system**.
- network file sharing protocols: NFS
- storage devices: RAID to increase bandwidth and reliability
- Use-case: streaming contents(movies, images) to home network. If connected to home WiFi, then access to local storage. Access out of home using VPN and public IP



2.7.4. Storage Virtualization

- Goal: location transparency
- Process: allocate a **virtual disk**, which will be **mapped to a real physical hard-disk** in the Storage Area Network. The **client won't know about which hard-disk is allocated** for the virtual disk.

Block Virtualization

- the **mapping** of a **virtual disk and block number** to a **physical disk and block number**.
- Use-case: SAN, flexible mapping, disk expansion and shrinking
- implementation:
  - host based: host runs virtualization software
  - storage device based: disk array provides a virtualization level
  - **network based**: virtualization device is in LAN and connected to a SAN, **most frequent implementation**
    - **in-band**: client sends request for certain blocks to the controller in the network, the controller fetches the data and returns the data to the client.
    - **out-of-band**: host contacts the controller, gets the mapping information and accesses the data in SAN directly without the help of controller.

File Virtualization

- Virtualization on **file level**
- Use-case: **Distributed File System**
  - allows transparent access to multiple NAS server. Files located on multiple NAS servers **appears as if on a single NAS**, the client doesn't know on which server the file exists.

3. Virtualization

- Idea: resource usage by a single user is **under-utilized**, the **efficiency** of usage of resource is **low**.

  → own a machine in a shared manner → **virtualization**

- Definition: Virtualization is a computer architecture technology where **multiple virtual machines** are **multiplexed** in the **same hardware**. (all VMs are connected and are owners of the hardware)

- Goal:
  - enhance resource sharing, **improve** machine **efficiency**
  - able to replace and upgrade hardware **on the fly**, **without interrupting the running program or rebooting**.
  - reduce down time
  - **faster provisioning** of multiple machines

- Modes of operation:
  - **kernel mode**: higher privilege
    - OS allows execution of **all CPU instructions**
    - kernel codes **don't execute in user mode**.
    - execute in **superuser/supervisor** privilege.
  - **user mode**:
    - OS allows execution of **few CPU instructions**
    - if user applications have to execute **privileged instructions**, they **ask kernel** to execute.
    - execute in user privilege.

3.1. Technology for Virtualization: Hypervisors

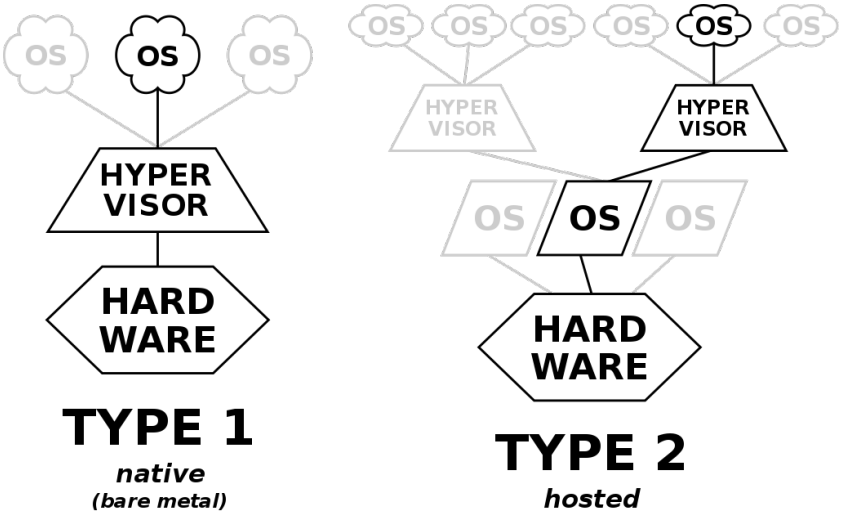
A **hypervisor** (or virtual machine monitor, VMM) is a kind of emulator; it is computer software, firmware or hardware that **creates and runs virtual machines**. A **computer on which a hypervisor runs** one or more virtual machines is called a **host machine**, and **each virtual machine** is called a **guest machine**.
The hypervisor **presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems**. Multiple instances of a variety of operating systems may **share the virtualized hardware resources**: for example, Linux, Windows, and macOS instances can all run on a single physical x86 machine.
This contrasts with operating-system-level virtualization, where all instances (usually called containers) must share a single kernel, though the guest operating systems can differ in user space, such as different Linux distributions with the same kernel.

3.1.1. Bare-Metal Hypervisor

- runs **directly** on **host's hardware** – Ring 0
- Guest OS operates at Ring 1
- provides **almost full isolation** to the users. All Guest OSs are **independent** and are owner of the hardware. VMM **controls the hardware and to manages resource allocation to guest OSs**.
- Use-case: Hyper-V, ESX, Xen

3.1.2. Hosted Hypervisor

- **runs on a conventional OS** just as other computer programs
- abstracts guest OSs from the host OS.
- Use-case: VMware player, VMware workstation, VirtualBox



3.2. Full Virtualization

- for **any guest OS without any modifications**
- VMM works in Ring 0, guest OSs work in Ring 1 – Bare-metal hypervisor
- Problem: **execution error** of privilege instructions from guest OS being in **lower ring** (Ring 1)
  - Solution: VMM intercepts such error and **emulates** the instructions **on the fly**
    → not all instructions are trapped.
  - Solution: a **binary translator**, which overrides these privileged instructions and places them in translation cache.
- Consequences:

  - system calls: takes **10 times more cycles** compared to no hypervisor, because **fault message** are issued, translated and executed **for every system call**.
  - I/O virtualization: major issue. **more I/Os** due to more CPUs (guest computers) but I/O chipset **can't be easily extended**
  - memory virtualization: 2-stage mapping process with VMM.
    * program's memory addresses → virtual physical memory (on VMM) → real physical memory
    * VMM maintains a **shadow page table**, this process takes **3 to 400 more cycles** than without VMM.

3.3. Paravirtualization

- guest OS needs to be **modified at source code level**. Information is given in prior in code, that if such instruction is being executed, it needs to go directly to the VMM instead of the hardware.

  → no execution error because of ring privilege. → no trapping or binary translation needed

- Hypervisor provides **interface** to accommodate critical kernel operations (memory management, interrupt handling)

- Advantages:
  - runtime changes are avoided. (no on-the-fly modification)
  - unnecessary trapping of critical instruction avoided (modified in code)
  - lower virtualization overhead

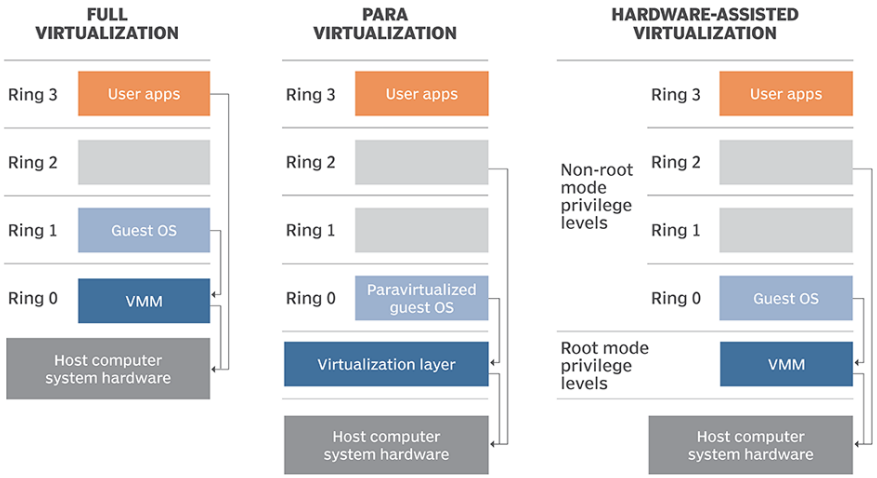
  Disadvantages:
  - a modified guest OS needed when changing machines.

3.4. Hardware-assisted Virtualization

- Idea: enables efficient full virtualization using **help from hardware capabilities**, primarily from the host processors.

  → a processor extension is introduced in a **higher priority layer** – Root mode privilege level with VMM.

- Guest OS now operates at **Ring 0** and can execute all critical instructions

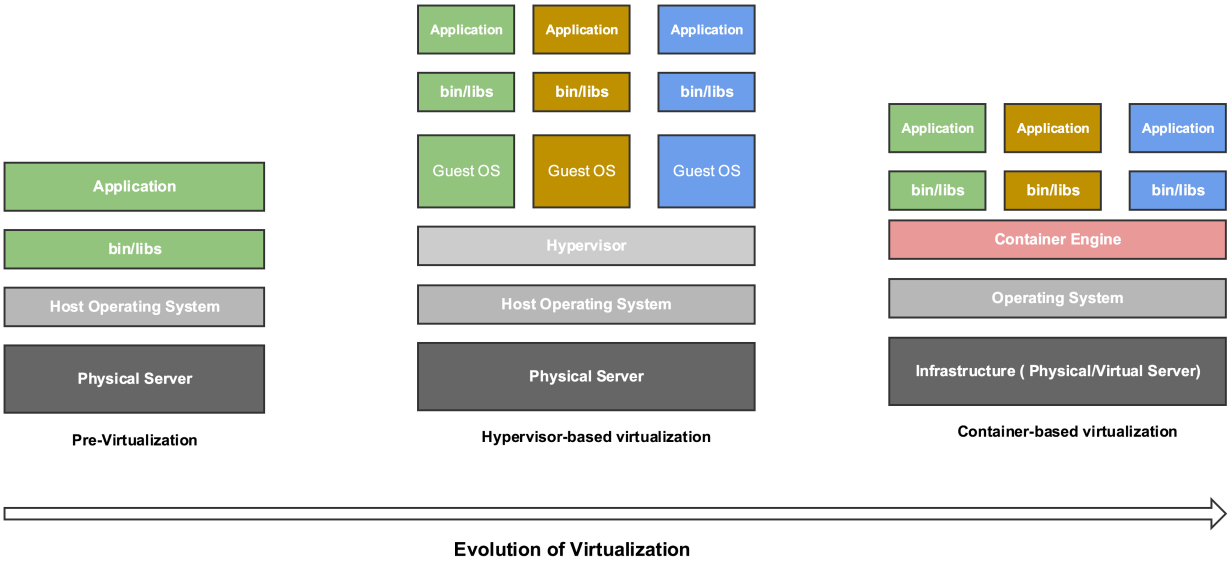


3.5. OS-Level Virtualization

- Idea:
  - disadvantages of hypervisor-based virtualization: OS dependent, not scalable, not portable, slow deployment.
  - higher scalability required in application.

  → **OS-Level virtualization**
- VMM is **on top of host OS**
- building blocks: linux kernel features
  - **namespaces**: **limit the views**, wrap a group of resources for limited access, managed using APIs
    - PID: create another set of PID
    - cgroup: new views to root directories
    - network: new views to network resources
  - **cgroups**: control groups, **limits the applications to a specific set of resources**
    - memory cgroup: memory resource controller, isolates the memory behavior of a group of task from others. when memory exceeds, running processes will be killed.
    - others: cpu, blkio, cpuset, devices, freezer cgroup
- **Container**:
  - allows multiple **isolated** linux systems of same kind on a single host OS.
  - resources are **isolated in a container** for each user.
  - it utilizes **cgroups and namespaces** to limit the views and resources of each container.
  - Use-case: **Docker**

- Advantages:
  - runtime isolation: isolates different runtime environments based on application requirement
  - cost-efficiency: no creation of entire virtual OS for each user.
  - easy portability of containers
  - high scalability, easy replication of containers across environments
  - faster deployment: layer-concept, only new layers/updated layers are built.



Part II.
Infrastructure as a Service – IaaS

4. Infrastructure as a Service – Amazon Web Service

AWS is distributed among **regions**. Each region is **isolated** for **fault tolerance and stability**. Each region consists of **availability zones** (eg: data centers), where data can be **replicated**.

- region availability: 99.99% from Amazon's SLA(service level agreement). To profit from SLA, data must be replicated over at least 2 zones in case of failure.
- IaaS service model:
  - compute: EC2,
  - network: VPC, CloudFront
  - storage: S3, EFS, EBS, instance storage

4.1. Compute Service: Elastic Compute Cloud EC2

- **virtual machine instance** running based on an AMI
- **Amazon Machine Image**: a copy of server with OS and preinstalled software (eg: Ubuntu server 18.04 LTS, SSD volume type)
- instance attached **ephemeral storage, block storage and virtual firewall**
- resource allocation through Xen Hypervisor (**bare metal hypervisor**)
- **elastic IP address** possible: private & public IP address **remains same** after start/stop
- lifecycle: pending – running – stopping – stopped –shutting down – terminated

Comparison Amazon EBS-Backed & Instance Store-Backed EC2 instance:

Characteristics	EBS-Backed	Instance Store-Backed
root device volume	EBS volume	Instance store volume
stopped state	stop-state exists, restart at a different server possible	no stop-state, only termination
data persistence	lost when termination, kept in EBS when stopped	lost when termination
modification	modifiable at stop-state	fixed

4.2. Storage

- **Elastic Block Storage (EBS)**
  - similar to Storage Area Network (SAN)
  - can be **mounted into VM**
  - multiple block storage can be **combined into RAID**
  - **snapshots** of the block storage are **stored in S3 for backup or replication**
- **EC2 instance storage**:
  - disk or SSD connected to the physical machine
  - **data erased** when instance is **stopped or terminated**
- **Elastic File System (EFS)**:
  - similar to Network Attached Storage (NAS), scalable
  - can be created and mounted into VM
  - files can be **shared among instances**
- **Simple Storage Service (S3)**
  - large size storage (up to 5TB)

- **slower access** compared to EBS or local disks, **high durability, low availability**
- Use-case: backup

	Amazon S3	Instance	Block storage
Speed	Low	high but unpredictable	High
Availability	Medium	High	High
Durability	Super high	Super low	High
Flexibility	low	Medium	High
Cost	Medium	Low	High
Strength	Backup data	Transient data	Operational data
Weakness	Operational data	Nontransient data	Lots of small I/O

Storage attached to EC2:

- root device volume: EBS, instance storage.
  - instance stops: data **preserved**
  - instance terminates: data **deleted**
- instance store volume: local disks of the server. Data **erased** if instance stops
  - instance stops: data **erased**
  - instance terminates: data **erased**

4.3. Network

- **elastic IP address**: public IP address **remains same** even after start/stop.
- **Virtual Private Cloud (VPC)**:
  - resembles network in own data center, resources are launched into own VPC
  - configuration of IP address range, subnets, route tables, network gateways, firewall settings
- **CloudFront**: content distributed network
  - automatic distribution: content changes in S3 and will be automatically updated in every data center from edge locations
  - response delivered from data center closed to requester.

5. Cloud Storage Systems

- Comparison VM storage VS. Cloud storage:
  - VM: instance volume – lost when VM stops, EBS volume – lost when VM terminates.
  - Cloud storage: for voluminous data potentially long-term storage.
- requirements:
  - voluminous data
  - commodity hardware
  - distributed data
  - failures is norm rather exception: replication & fault tolerance
  - processed by application
  - optimized for typical access patterns
- **CAP Theorem**: aim for **availability** or **consistency**. Partition-tolerance has to be guaranteed.
- Types:
  - object storage: S3
  - shared file systems: NAS, EFS
  - relational databases
  - noSQL databases
  - Data warehouse

5.1. Object Storage: AWS S3

- Goal: infinite data **durability** by 99.99% availability
- Use-case: short-term or long-term **backup**
- storage classes:
  - standard: for **frequently accessed** data
  - reduce redundancy: for **reproducible, infrequently accessed** data
  - intelligent tiering: objects are **automatically moved** between frequent and infrequent access tier.
  - glacier: long-term data archive. long retrieval time
  - deep-archive: even longer retrieval time, for **rarely accessed** data.
- Data access: objects can be uploaded, retrieved and deleted. **No modification**.
- Versioning: versioning for individual files or entire buckets.

- Lifecycle: consists of rule that triggers **transition** to other storage classes or **expiration** of objects.
- consistency: **eventual consistency**. simultaneous puts – last write wins.

5.2. File Storage: Google File System

- **distributed** file system
- architecture:
  - 1 **master server** + many **chunk servers**
  - **master server**: metadata in main memory stored as **lookup table**
  - **chunk server**: stores many 64MB-chunks as linux files.
  - shadow masters possible to reduce load on master server
  - **decoupled** control and data flow. **Control flow runs through master server** to chunk server, **data can directly flow from chunk server to client**.
- replication: data is replicated to balance workload and combat failures.
- data access: client **first contacts master server** and finds out the corresponding chunk server through lookup table. Later interactions(writes, read) **directly between chunk and client**.
- consistency: cocurrent writes possible, but only writes on **primary chunk** will be **replicated**. **consistent but undefined state**.
- Limitations:
  - scalability of single master → development of distributed master
  - fixed chunk size, while application can have smaller files.
  - no latency guarantees.
- other file systems: AWS Elastic File System

5.3. Relational Database

- data represented as **n-ary relations**. Each relation is a **table**, which is defined by **schema**.
- data access: SQL-queries
- **ACID-properties** of transactions: atomicity, consistency, isolation, durability
- designed for **vertical scaling**
- Installation of own database:
  - installation and management **on you own**

  - data management: certificate, patching, performance tuning, backup, scaling, security

- Cloud relational database service: Amazon **managed** RDS, Amazon Aurora (own relational database)
  - **managed** standard relational databases: mySQL, PostgreSQL
  - different **certifications provided**
  - **pre-configured** database instances
  - standby copies in case of failovers
  - multiple read replicas to increase read throughput, guarantee fault tolerance
  - **automatic** backup snapshots in S3, **automatic** scaling

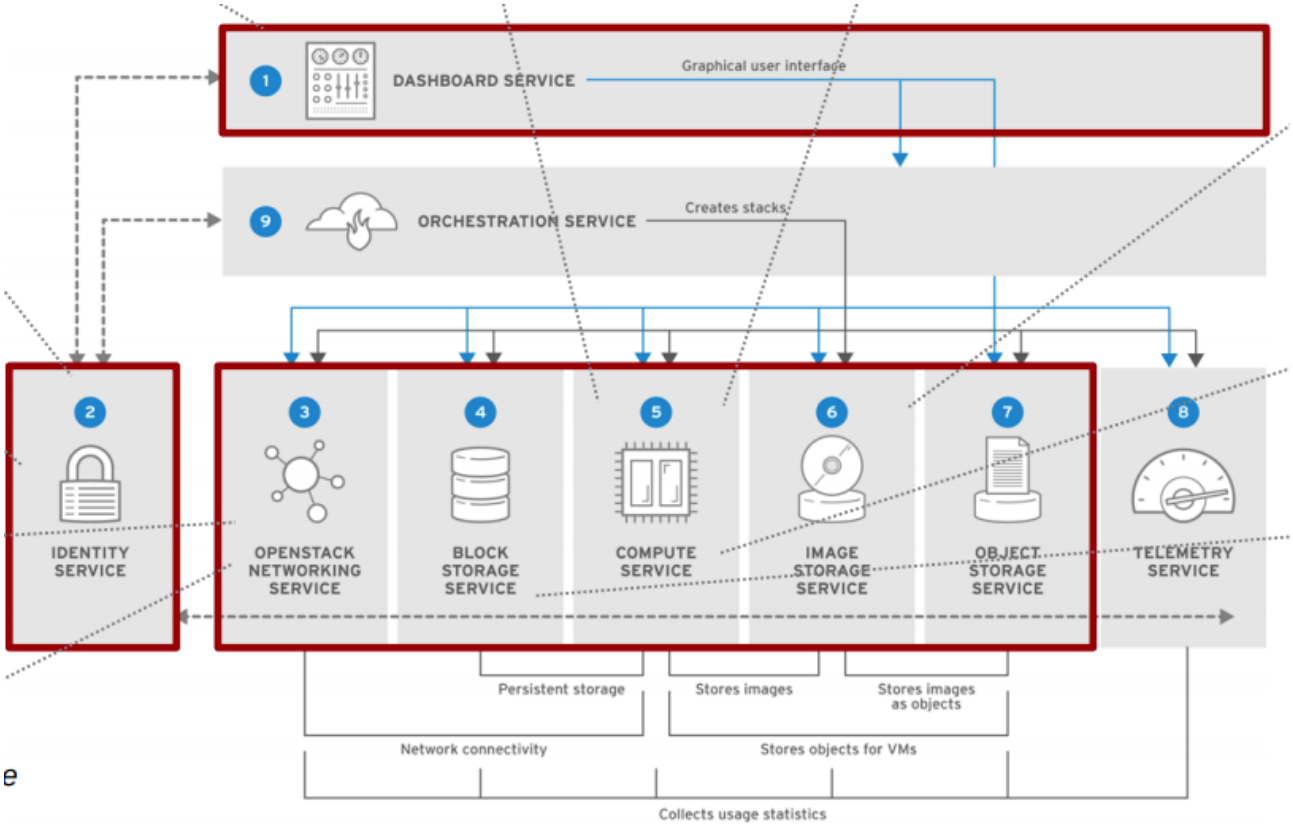
5.4. NoSQL Database

- database for **non-relational** data **without predefined schema**. Easy to define new attributes.
- designed for **horizontal scaling**
- Types of NoSQL databases:
  - key-value database: Amazon Dynamo
  - document oriented
  - graph database
- Cloud NoSQL database service: Amazon Dynamo DB
  - **distributed** data store → optimized for **small** request, **quick** access and **high availability**.
  - **key-value** database
    - **primary key**: unique for each row, **composite key** possible (partition key + sort key).
    - values: a set of attributes
  - Consistency: **eventual consistency**
    - **normal reads**: eventual consistency, might return old values
    - **strong consistent reads**: consistent, always the latest.
  - Architecture:
    - **key-value pairs** stored in **tables**.
    - **tables** stored in **partitions**
    - number of partitions depends on: size, required read/write capacity units
  - management of partitions: key → partition → virtual node → physical nodes
    - key → partition: key-value pair entries are **hashed by keys** onto a **ring space** of partitions.

    - partition → node: **ring** is split into **segments containing multiple partitions**. Each segment is managed by a **virtual node**.
    - virtual node → physical node
  - Replication: (N, R, W)
    - N: replication to **N consecutive nodes**. N ↑, durability ↑
    - R: success **read** on **R copies**. R ↓, latency ↑
    - W: success **write** on **W copies**. W ↓, latency ↑
    - **R + W > N**: ensures the **most recent write** is returned.
  - Failures is a **norm**: **gossip protocol**

6. Infrastructure as a Service – OpenStack

- main services:
  - Compute: Nova, central computation
  - Identity: Keystone, **authentication**, generate **token** for subsequent calls
  - Image: Glance, base image selection for VM (**template for VM**)
  - Networking: Neutron, determine networking **rules**, allocate **network resources**
  - Dashboard: Horizon
  - Block storage: Cinder, allocate **volume**
- Design concepts:
  - Goal: scalability (horizontal), elasticity
  - **asynchronous** communication: message **queues**
  - **decentralized** data management: each microservice has its **own database**, accepts **eventual consistency**
  - **distribute** everything: distributed into microservices, communication through APIs
- Architecture:



6.1. Identity Management

- provides authentication credential validation and data
- generates **authentication tokens** that enable clients to access OpenStack services REST APIs.
- clients obtain this **token** and **URL endpoints** for other service APIs.

6.2. Compute Service

- manage and provision **virtualized server resources**: CPU/memory/disk/network
- VM management: start/stop/reboot/terminate
- communication with other services: image, identity, network, storage services

components:

- Nova API: interface to create instances
- RabbitMQ and compute database : provides communications hub and manage data persistence
  - Message **Queue**(MQ): pass messages between services **asynchronously**

  - compute database: **store** build-time and run-time **states**, available instance types/in use, available networks etc.

- Scheduler: determines **allocation of physical hardware** to virtual resource using **series of filters**
- **compute node**: **management of all interactions** with individual endpoints providing computing resource

6.3. Network, Block Storage and Image Storage

- network: manage **networks, ports, attachments on infrastructure** for virtual resources
- block storage:
  - create and manage **lifecycle of volumes**
  - respond to **read and write requests** sent to block storage to maintain states
  - **backup** volumes
- image storage:
  - storage in **independent** image **database**
  - accept API calls for **image discovery, retrieval and storage**

6.4. Server Creation Workflow

- Nova Client gets **token** from Identity Management service.
- After **authentication**, Nova API **sends request** of launching instance **into Message Queue**
- Nova scheduler **subscribes request** and **allocate resources using filtering**
- Nova compute communicates with Image service, Network service, block storage service to **get image, allocate network and storage**.
- Nova compute starts instance.

7. Infrastructure as a Service – Microsoft Azure

- **company-oriented** service
- services:
  - Azure Virtual Machines
  - Azure Virtual Network: connects **VMs and services**, connect **cloud resources with on-premise resources** through **Internet** via VPN.

  - Azure ExpressRoute: connects **cloud resources and on-premise resources** via **dedicated lines**.

- **Traffic Manager**: distribute resources to either Azure or on-premise data centers – different service endpoints.
- **Resource Manager**: management as a **group**. Resource allocation defined in **template** with version control.

Part III. Platform as a Service – PaaS

- Definition:
  - client: deploy **customer-created applications**
  - cloud provider: provides web interfaces, programming languages, libraries, services and tools
- Rights as customer:
  - application deployment
  - configuration settings application-hosting environment
- No control:
  - cloud infrastructure: storage, network, servers, operation systems

8. Platform as a Service – Microsoft Azure

- Focus: integration with **on-premise IT** – company oriented
- Services:
  - Azure Web Apps
  - Azure Service Fabrics: microservices
  - Azure Functions
- Azure access:
  - **Tenant**: a Microsoft Active **Directory** for managing accounts, groups and permissions. (eg: a company)
  - **Subscription**: a subscription is **associated with a unique tenant**. A **resource container** for the tenant.
    * can be divided into **multiple resource groups**. (eg: different teams) Each has its **own resource allocation**.
    * account (eg: an employee): has a **role** in subscription: eg: account administrator, service administrator

8.1. Azure Web Apps Service

- **development, deployment and management of web applications** without managing VMs
  - different programming languages provided: development
  - a **managed web environment**: deployment
  - automatic load balancing: **scale in/out**
  - VM sharing possible
- App development:
  - creation of a **deployment user**
  - creation of a **resource group**: name, location, template
  - creation of a **app service plan**: VM instance type, number of VM instances, scale count, subscription (resources), region, pricing tier.
  - creation of a web app → development ready
- App scaling: triggered by **automatic load balancing**, scales **according to app service plan**, apps will be **scaled together**.
  - manual scaling
  - **automatic** scaling: according to defined metric/target/time period.
- App monitoring:
  - per app:

metrics	definition
Average response time	time taken to serve requests (s)
Average memory working set	average amount of memory used (MiB)
CPU time	amount of CPU consumed (s)
Data In	amount of incoming bandwidth (MiB)
Data Out	amount of outgoing bandwidth (MiB)
Requests	total number of requests regardless of HTTP status code

  - per app service plan:

metrics	definition
CPU percentage	**average** CPU used **across all instances** (s)
memory percentage	**average** memory used **across all instances** (MiB)
Data In	**average** amount of incoming bandwidth **across all instances** (MiB)
Data Out	**average** amount of outgoing bandwidth **across all instances** (MiB)
Disk Queue Length	**average** amount of read & write requests queued on storage (disk I/O)
HTTP Queue Length	**average** amount of http-requests (plan load)

  - granularity (level of detail of data): minute/hour/day granularity metrics

8.2. Microservices Platform – Azure Service Fabric

8.2.1. Monolithic VS. Microservices

- **Monolithic** service:
  - tiered architecture (eg: presentation/user interface – logic – data layer)
  - **replication** of the **whole app** to other instances
  - all states stored in a single database.
  - testing and failure: one part fails, the whole app fails. Changes of one part requires testing/deployment of whole app.
- **Microservice**:
  - app is **decomposed** into microservices.
  - **individual replication** of each microservice over instances. **individual scaling**.
  - stateless services: **individual database** connected to microservice for state storage.
  - stateful services: state is stored in the service.
  - testing and failure: **individual** testing and deployment. If one microservice fails, the rest is unaffected.
  - increased network traffic and latency sensitivity: communication between microservices, especially chatty services.

8.2.2. Service Fabric

- platform for **microservice based applications**
- Transformation of a monolithic service to a microservice:
  - lift and shift: containerize existing codes in service fabric
  - modernization: add new microservices alongside the containerized codes

  - innovate: break the monolithic service into microservices based on needs.

- Fabric cluster: **a set of virtual/real machines (nodes)** where microservices are deployed.
  - service - node allocation: placement constraints of services, properties specified by node in key-value pairs.
  - load balancing: **distributes/migrates** services to other nodes, **distributes requests**
  - scaling of a whole cluster possible: app scaling

- service models:
  - reliable services model
  - reliable actors model: **stateful** objects, actor **communicates** by receiving and sending **messages**.actor reacts faster than stateless services.
  - guest executables: treated like stateless services

- **service partition**: a group of service instances
  - stateless service: **not partitioned**. all instances belong to a **single partition**. eg: collection, gallery, overview
  - stateful service: **frequently partitioned or replicated**. eg: by country, name, payment status

- testing:
  - restart nodes/partitions of service instances
  - simulate load balancing, failover, app-upgrade
  - invoke data loss
  - simulate scenarios: chaos(continuous overlapping faults), failover(continuous overlapping faults targeting single partition)

9. Microservices Application Architecture

9.1. Monolithic Application Architecture

- a software program composing of **all in one piece**.
- each component is tightly **coupled**
- data: a single database
- layered/tiered architecture:
  - client-side user interface/presentation
  - server-side application: performs detailed processing
  - database

- Advantages:
  - easy development
  - simple testing: end-to-end testing
  - simple deployment
  - easy horizontal scaling: run multiple copies of complete app behind a load balancer

  Disadvantages:
  - limitation in size and complexity: no continuous addition of codes
  - large and complex size
  - slow start time: dependent of size of application
  - redeployment of **complete** app on minor updates
  - low reliability: one module fails, the whole app fails.
  - difficult to scale

9.2. Service Oriented Architecture

- functions of application as **service interfaces**
- data: a single database, data is **internally componentized**
- communication: each function logic **only accesses its data associated**. When **another function** requires data from other component, **communicate through interfaces**.

9.3. Microservices Architecture

- size: **split** the whole application into **a set of smaller and interconnected services**
- data: each service has **individual database/storage system**
- deployment:
  - each service runs **individually** on single or multiple machines.
  - services communicates with common communication protocol (eg: REST).
- operation: **stateless**, information is supplied on request. Once request is complete it's forgotten.

  → allows rapid scaling
- Advantages:
  - easier understanding and maintenance
  - service independence: developed, tested and deployed independently
  - fault resilience: one service fails, the rest is unaffected
  - individual service scaling

Disadvantages:

- complexity of creating a distributed system: difficult testing, inter-process communication implementation necessary
- deployment complexity: each service instance needs to be configured, deployed, scaled and monitored. a **service discovery mechanism** implementation necessary.

9.4. Microservice Application Framework Components

9.4.1. API Gateway

- Idea:
  - monolithic: access to application through **single REST call**
  - microservices: access to each microservice using different endpoints is suboptimal (client needs mismatch, difficult to refactor, in reality many microservices).

    → **API-Gateway**: a **single-entry point** into the system.
- encapsulates the whole internal system architecture. Clients don't know which microservice APIs are led to.
- **separate** API-Gateways tailored for client(web, mobile, public): customize data sent back to client.
- requirements:
  - service registry: database of network locations of service instances
  - service discovery: query and decide for the available service instance

9.4.2. Service Registry

- **database** containing the **network locations** of all **available** service instances.
- registration:
  - self registration: instances **send requests** to registry when up/down
  - 3rd-party registration: managed by a **service manager**, which checks status and sends request to registry, then starts/stops instances.

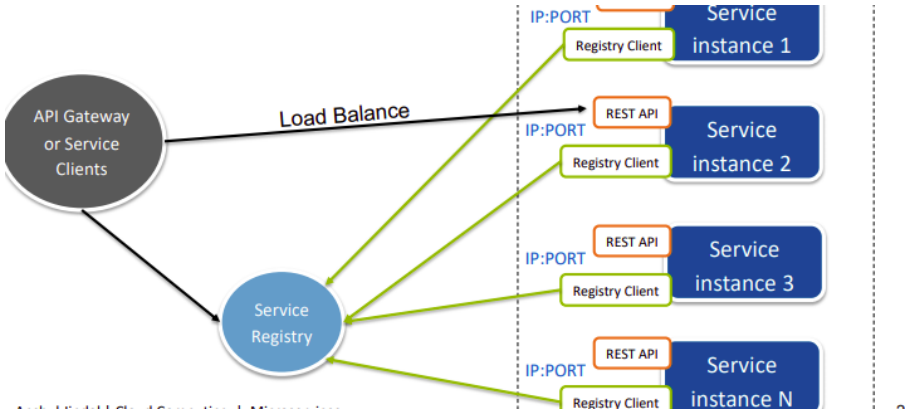
9.4.3. Service Discovery

- Idea: **dynamically assigned** IP-addresses of instances due to failure, upgrades, scaling

  → discover the **current available** instance and its network location for API Gateway from the service registry
- **client-side** service discovery:
  - **clients are responsible** for determining network locations
  - client **queries the service registry** for available instances

- client **executes load-balancing algorithm** and selects an instance
- Advantages:
  - straightfoward, no other moving parts except service registry
  - able to make application-specific load-balancing decisions, weighting possible

  Disadvantages:
  - coupling of service registry with client
  - more code on client side

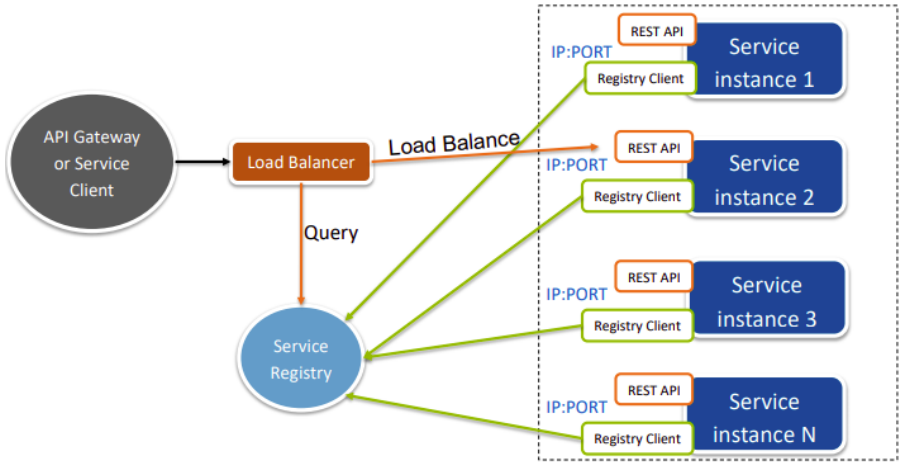


- **server-side** service discovery:
  - an extra **load balancer**: it queries the registry and implements load balancing.

    → client has **no contact to service registry**

    → client **doesn't need to load balancing**
  - Advantages:
    - discovery abstracted away from client
    - less code

    Disadvantages: extra load balancer required



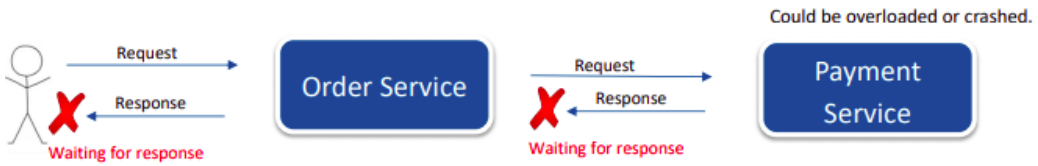
9.4.4. Resilience – Fault Tolerance Strategies

- basic request flow:
  - a user/load balancer sends a **request** to service and waits for a **response**.
  - a **thread is assigned** along the request, in order to get data and send it back.
  - when response is sent, the **thread is freed**.

- **immediate failure**: fails at **request** (eg: connection refused) to service.
  - possible reasons: overhead of requests
  - solution: **try...catch**-code catches the error and returns error to client → thread is freed

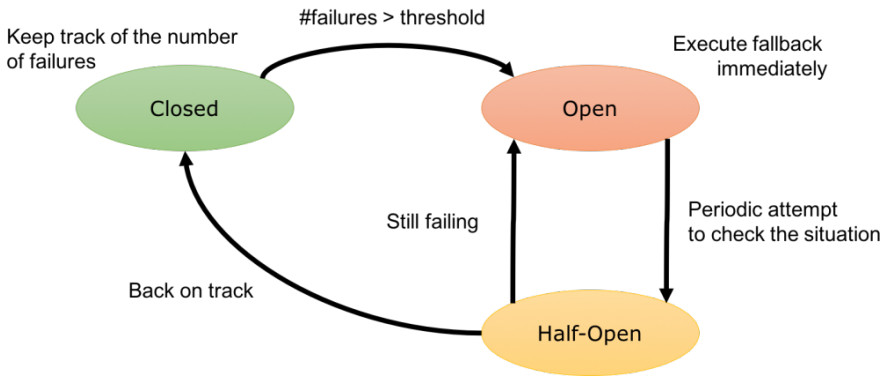
- **timeout failure**: fails at **response** from service to service. At high request rate, all threads are **waiting for response**, no free threads.
  - possible reasons: service is overloaded or crashed

- **cascading failure**: due to timeout failure of one service instance to another service instance, this **timeout failure** is **propagated** to neighboured connected instances.



- Solutions to timeout/cascading failures:
  - timeout-value: returns error after timeout, thread is freed
  - request interceptor/**circuit breaker**

* **intercepts all requests** from service to target service.
* **counter & threshold implementation** of success & failures.
  - closed: #failures < threshold
  - open: #failures > threshold, default error response, thread is freed.
  - half-open: remains open for a predefined sleep period and rechecks.
* checks regularly and refreshes.



9.4.5. Deployment Strategies

- **multiple service instances** per VM: service instances share resource of a VM together → for resource-efficient services
- **single service instance** per VM: for resource-hungry services
- **multiple service containers** per VM: each service instance is containerized with its needed libraries & dependencies. → high portability, high scalability
- **single service container** per VM

9.4.6. Rolling Updates Strategies

- Goal: implementation of updating of services.
- **Ramped-slow Rollout**: update in **rolling update** fashion. It **decreases old version** instances and **increases same number of new version** instances, until all old versions are removed.
  - Advantages:
    * new version is slowly released across instances
    * convenient for **stateful applications** that can handle rebalancing of data

  Disadvantages:
    * takes more time
    * supporting mulitple APIs harder, if both versions use different patterns.

    - no control over traffic, since load balancer directs request to either of the instances, it might follow wrong robin pattern.

- **Blue/Green**: new version of app is **deployed alongside** the old version. After testing new version, the load balancer will be **redirected** to send traffic to the new version.
  - Advantages:
    - instant rollout/rollback
    - no versioning issue (multiple versions coexist)

    Disadvantages:
    - requires double the resources. (same size of empty instances needed)
    - requires testing before releasing, but still can't simulate the workload by client requests

- **Canary**: new version is **deployed and receives a subset of client requests** alongside the old version. We distribute the traffic amongst the versions by **adjusting the number of replicas** managed by a ReplicaSet. After successful testing, all old versions are removed at one time.
  - Advantages:
    - version released for a subset of clients → testing of workload possible
    - convenient for performance and error rate monitoring
    - fast rollback

    Disadvantages:
    - slow rollout: testing of workload, monitoring and improvement takes time
    - fine tuned traffic distribution can be expensive.

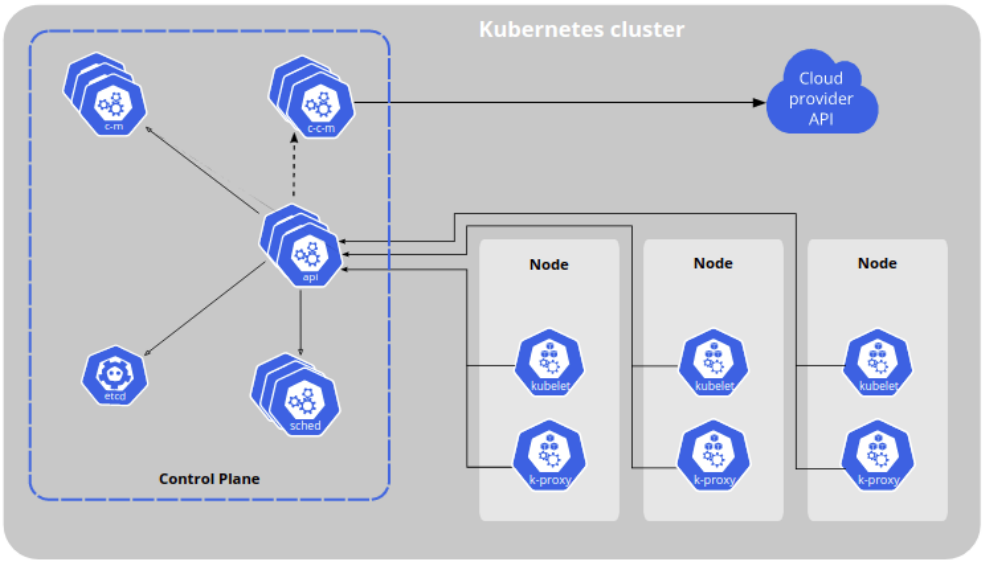
- **A/B Testing**: distribute traffic among versions **based on filter parameters** (weight, user, cookies). After successful testing, old versions are removed.
  - Advantages:
    - intelligent load balancing
    - full control over traffic distribution

    Disadvantages:
    - harder troubleshooting, distributed tracing of versions necessary
    - additional tools needed

10. Container Orchestration Platform – Kubernetes

- **Container Orchestration**: management and deployment of lots of containers
  - container placement: selection of host for containers
  - container resource usage monitoring
  - container health checks
  - container scaling
  - access to services: IP management & load balancing
  - container networking: microservice communication
  - persistent storage management
- Techniques for container orchestration: **Kubernetes**, docker, AWS/Azure container service
- Kubernetes:
  - a **cluster**: for running application
  - an **orchestrator**: run & coordinate cloud-native **microservice** applications.
  - **declarative** management: description of application in **yaml file**.

10.1. Kubernetes Architecture and Components



10.1.1. Master Node / Control Plane

- makes **global decisions** about the cluster as well as **detecting and responding to cluster events**.
- **Kube-APIserver**: front-end

  - client gateway
  - REST interface into control plane and datastore.
  - receives declarative yaml files.

- **cluster store**: cluster data key-value storage, consistency over availability. eg: etcd.
- **controller manager**: independently control nodes(detect & restart fail nodes), jobs, endpoints
- **Kube-scheduler**: watches **newly created pods** with **no assigned node** and select a node to run on.

10.1.2. Node

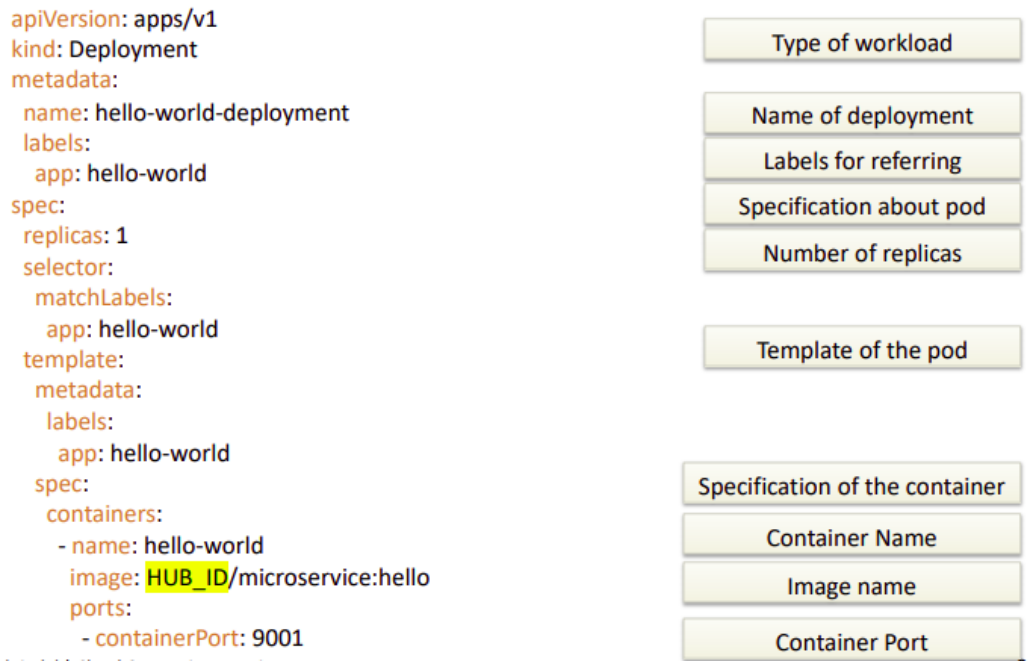
- maintains running pods on the node and provides kubernetes runtime environment.
- **Kubelet**: **node-level manager** between master and node.
  - receives assignment from Kube-APIserver and report states back.
  - management of pods lifecycles.
- **container runtime (CRI)**: performs **container-related tasks**, eg: docker
  - pull images, start/stop containers
  - Each node has only one CRI, different nodes can have different CRIs.
- **kube-proxy**: local cluster networking, maintain network rules and communication
  - ensures each node has **its own IP**
  - routing and load-balancing on the pod network for services

10.1.3. Pods

- the **smallest deployable units** of computing that you can **create and manage** in Kubernetes.

  $\rightarrow$ automic unit for starting/stopping/scaling
- Pod: a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. (resource limits defined in cgroups)
- provides environment for containers:
  - each pod has a unique IP address.
  - each container inside the pod has its own port.
  - containers share memory, volumes and network stack.
- pod-to-pod communication: pod network
  - bridge network
  - routing tables

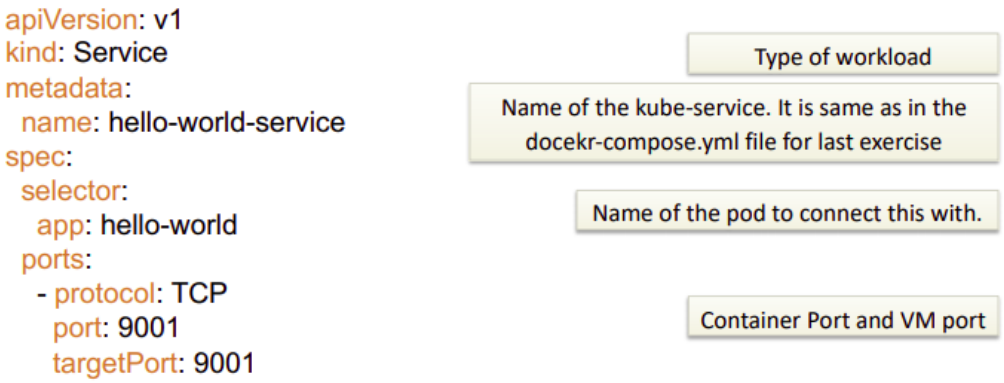
- pod deployment:**declarative description**, provide specification of self-healing(failed pods will be replaced, mortal), scalability and rolling updates.



- managed by higher-level controllers:
  - deployment
  - deamonSet
  - StatefulSet

10.2. Service

- provides reliable **networking for a set of pods**
  - stable DNS name, IP address and port
  - load balance across a set of pods
- types:
  - clusterIP service
  - NodePort service
  - LoadBalancer service
  - ExternalName service
- service discovery:
  - service **registers** automatically with **cluster DNS**.
  - request handled by **rewriting the service IP address** from cluster DNS **to IP address of pod**.



10.3. Kubernetes Storage: Persistent Volume

- **Persistent Volume**: a piece of **storage in the cluster** that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
  - different properties: capacity, storage class, access mode, etc.
  - persistent volume object : map external storage onto cluster
  - **persistent volume claims** (PVC): a request for storage by a user. pods act through PVC to access PV.
  - storage class: automates creation of PV

10.4. Configuration: ConfigMap

- ConfigMap: stores **configuration data** outside of a pod, which can be **dynamically injected into pod at runtime**
  - key-value pairs
  - injected by: environment variables, files in volume, etc.

10.5. Autoscaling & Monitoring in Kubernetes

10.5.1. Autoscaling

- **horizontal Pod autoscaler**
  - modifies number of **replicaSet**, desired number defined by rules/thresholds
- **vertical Pod autoscaler**
  - specification: update policy, resource policy, Deployment/StatefulSet
- **cluster-level autoscaler**: adapts the **#nodes**

10.5.2. Monitoring

- Monitoring platform: Prometheus
- Monitoring metrics:
  - application monitoring: CPU usage, response time
  - infrastructure monitoring: node monitoring
  - Kubernetes monitoring: metrics about objects(deployment, pods), apiserver workload, kubelet container metrics

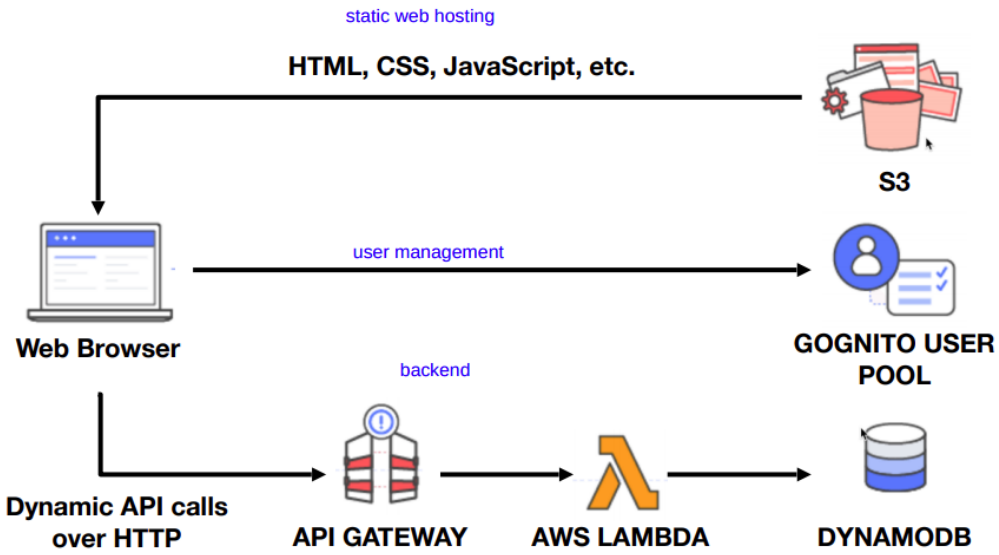
11. Function as a Service – FaaS

- extension of **Platform-as-a-Service** model, provides **platform for serverless computing**.
- **Serverless Computing**: develop, run, and manage application **functionalities without worrying about the underlying infrastructure/scaling** (physical/virtual servers).
  - FaaS: **event-driven** computing, functions are **triggered** by events/http-requests
  - Backend-as-a-Service(BaaS): third-party API services that replace core subsets of functionality in application.
- Advantages:
  - **concentration on development of application** instead of deployment/allocation/scaling of resources.
  - no provisioning of servers, automatic scaling(no autoscaling policies)
  - cost reduction. **Pay per function invocation**. (Microservice: one instance when service is not used will be paid.)

  Disadvantages:
  - focuses on **stateless functions**
  - unsuitable for heavy workloads → too many invocations, expensive
  - limited security: shared VMs, no control over network
  - vendor lock-in
- When to FaaS:
  - easily **decomposable** functions
  - highly-**variable demand** – autoscaling being taken cared.
  - **low/medium frequency function triggering**, overhead in running instance is high (pricing)
  - need for tight integration to cloud events
- Use-case: Amazon Lambda, OpenWhisk(open source)

11.1. AWS Lambda

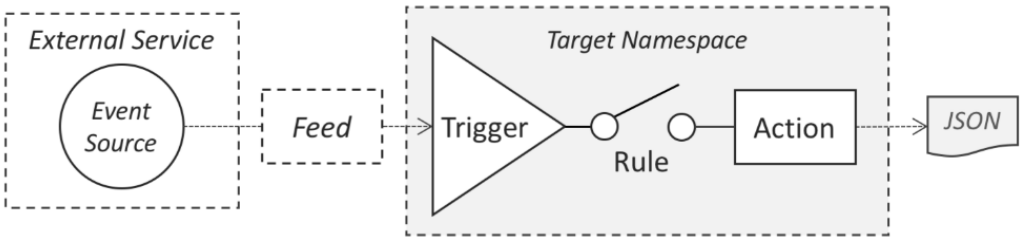
- lambda functions: anonymous functions, often used as **arguments** being passed to higher-order functions or as result of a higher-order function.
- event sources:
  - data stores(insert/delete in dymnamo DB, image upload on S3),
  - endpoints(API requests, IoT measurements)
  - development and management tools(CloudWatch monitoring results)
  - event/message services
- execution model: synchronous, asynchronous(event queue), stream-based
- Pricing: #requests, compute time
- Architecture:



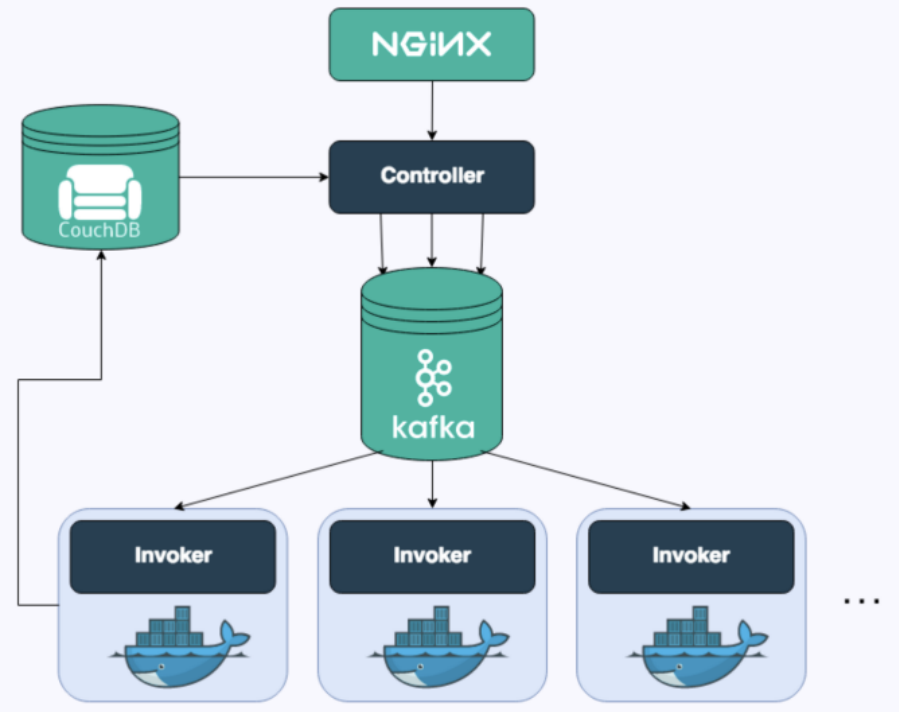
  - static web hosting in S3: **static web content** for user interface are **stored into a bucket and registered** in S3 → gets a public URL
  - **user management** with Cognito
  - serverless service backend: **handles requests**
    - requests from browser –> API Gateway invokes lambda function –> Amazon lambda function implementation –> access/request in Dynamo DB in the cloud
    - Dynamo DB: create table, Identity and Access Management role for lambda function (permission to write into Dynamo DB using lambda function)

11.2. OpenWhisk

- open source FaaS platform
- Concept: **event sources trigger action codes**
  - trigger: named channels for classes or kinds of events sent from Event Sources
  - rules: associate one trigger with one action. After an association is created, each time a trigger event is fired, the action is invoked.
  - action invocation:
    - blocking: calls action, wait until it's executed, return value delivered
    - non-blocking: action trigger is recognized, returns immediately(might be no value if not available) without waiting for action execution
  - action chaining



- Architecture & Process:
  - **entering the system – nginx**: front-end, a HTTP user API into the system. **Requests send into system**.
  - **entering the system – controller**: **user authentication and authorization**(gets from **CouchDB**). when succeeds, translates user **requests into an invocation of actions**.
  - **getting the action – CouchDB**: actions (codes + default parameters) are loaded into CouchDB. It merges with the user request parameters.
  - **choosing action invoker – load balancer(controller)**: checks available executors(**invokers**)
  - **queue – Kafka**: handles two issues: losing invocation(system crashes) and waiting(heavy workload). Controller publishes **message**(action to invoke, user parameters, allocated invoker) to Kafka. User gets activationID. **Asynchronous execution**.
  - **invoking codes – invoker**: invokes actions. Execution of actions are isolated **inside a container**(Docker).
  - **storing the results – CouchDB**: action results saved as **JSON object into CouchDB**
  - **getting the result – nginx**: get result using activationID.



- Optimization: **reduce cold start time** – time to prepare container to start
  - **cold container**: container has to be **created from scratch** → longest cold start time.
  - **pre-warmed container**: pre-allocated container with **initialized runtime environment**(eg: nodejs). It needs to initialize and run the function.
  - **warm container**: container with **initialized function and runtime environment**. It can directly **rerun same subsequent** function invocations.
  - **hot container**: container with **running function and runtime environment**. Within the grace-period, it **executes same subsequent** functions invocations.
- function composition(chaining):
  - target: small, simple, stateless functions
  - approaches:
    - client side: calls one function that executes both
    - server side: double billing problem
    - event-driven: first function triggers second
    - **premitive sequence**: define a sequence in command-line.

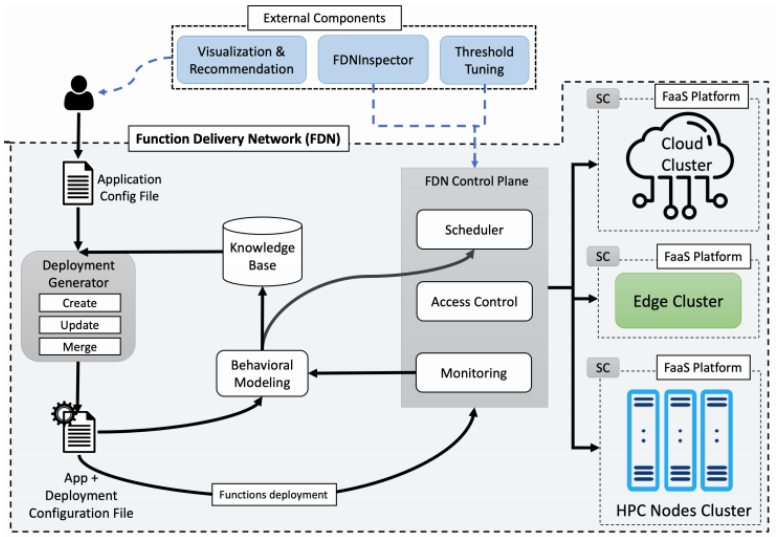
      → no special code required, no knowledge required about function relationship, no double costs.

12. Function Delivery Network – Serverless Computing for Heterogeneous Platforms

- Motivation:
  - current **FaaS platform** are limited to **homogeneous nodes/clusters** (eg: same architecture/VM/GPU)
  - **FaaS landscape** however very **heterogeneous**: software platforms, architecture different, computation requirements
  - when scheduling functions, **no consideration in data and computation requirements** (eg: machine learning – more GPUs, vector processing – accelerators)

  → need for **heterogeneous FaaS platform**

  → **Function Delivery Network**
- Function Delivery Network: integrates multiple heterogeneous FaaS clusters into one network.
  - deployment of functions across clusters based on:
    - data requirements: edge devices(locality, low latency)
    - compute requirements: HPC nodes(high demanding functions)
    - cluster capabilities
    - cluster resource usage
    - response time: edge devices(low latency)
  - **migration of data** based on function requirements: data migration from cloud to edge cluster and execution in edge cluster.
  - fault-tolerance across clusters
- Architecture:
  - control plane schedules function deployment to the best suitable FaaS platform.



Part IV.
Monitoring and Autoscaling

13. Cloud Monitoring

13.1. Introduction & Definitions

- Motivation: get data about the running system to make **best use of resources** at **lowest cost**.
  - cloud infrastructure: resource management, root cause analysis, accounting & metering, incident detection
  - cloud applications: performance analysis, resource management, failure detection and debugging, SLA verification
- Definition:

Monitoring the process of **collecting status information** of applications and resources to observe application and infrastructure.

Monitoring System components for gathering monitoring data at runtime – **the target system(HW/SW), raw data and data storage**.

- requirements: comprehensive, low intrusion, extensibility, scalability, elasticity, accuracy, resilience

Target System system to be monitored. an **online system**

- status depends not only on the **codes** but also the **environment** where it runs(#VMs running on physical machines, bandwidth, cloud storage)

  → execution is **not reproducible**
- **real-time analysis** and **continuous data collection**

Analysis Tools tools that read monitoring data from data storage.

Informations

- Alerts: alarm of exceeding thresholds. eg: CPU utilization 100%, full storage
- Dashboard: overview of resources
- Costs: billing and usage of resources
- Bottleneck: most critical service/program that slows down response time/ most CPU time
- Root cause: once bottleneck defined, find out the root cause
- etc.

Blackbox Monitoring **no data** are gained from **inside** of the monitored system(blackbox). A request interface, no visibility of internal system structure. eg: state of service like CPU/disk usage

Whitebox Monitoring data **also gained** from **inside** of the monitored system. eg: metrics specific to application #requests

13.2. Three Pillars/ Data Sources of Monitoring: Logs, Metrics, Traces

13.2.1. Logs

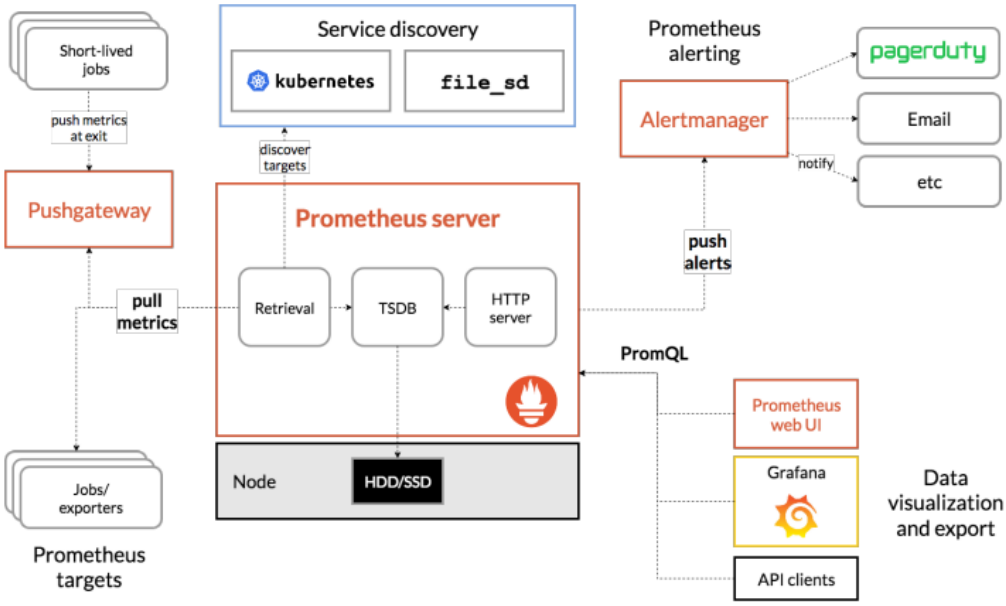
- Definition: a sequence of immutable records of discrete events.
- Characteristics:
  - forms: plain texts, structured (eg: JSON), binary
  - high level of detail, different levels possible
  - hard to read and analyze
- Techniques of Processing and Analyzing Logs: **ELK Stack**
  - Beats & Logstash: data ingestion. read, process logs and insert into Elastic Search.
  - Elastic Search: search, analyze and store data
  - Kibana: data visualization
- Other providers: AWS CloudWatch Log Insights

13.2.2. Metrics

- Typical metrics:
  - **latency**: response time
  - **throughput**: #requests/sec, network I/O rate, concurrent sessions, #transactions/sec
  - **error rate**: rate of fail requests.
  - utilization: CPU time/percentage, memory percentage, disk queue length(I/O)
- monitoring in different cloud layers:

	Context	Metric Aggregation: no, min, max, mean, percentiles	Purpose
Client Requests	Request type	#requests, latency, Availability	SLA check Alerting
Application Microservices	Service name Service id	#requests, request rate Latency, #replicas CPU time, memory usage	Autoscaling Performance tuning
Platform Kubernetes Docker	Container id	CPU & memory quota, utilization, incoming & outgoing bytes	Autoscaling
Infrastructure VM, volumes Queueing services	VM id, volume id Service name	CPU & memory #read/write, I/O latency #requests, size of requests of infrastructure service	Root cause analysis
Hardware Servers, network SAN, disks	Server id, switch id	Disk utilization, traffic	Management of VMs

- Techniques for Processing and Analyzing Metrics: **Prometheus** (open source)
  - data collection, storage & querying: **time series** data
  - Architecture:
    - **server**: retrieved metrics saved in TSDB/storage, HTTP-server for data querying and analysis
    - **target discovery**: define monitored targets
    - **metric retrival**: scraping at endpoint /metrics from target, pull for long-lived jobs, push gateway for short-lived jobs
    - **alert manager**: receives alerts from server and send to various channels
    - **data visualization**: Grafana
  - Scalability: **hierarchical**. Global DC stores aggregated data, local DC stores more detailed data.



  - Other providers: AWS CloudWatch (custom and user-defined metrics)

13.2.3. Traces

- Definition: Finding out **interaction** of different services/events and **association of sequence**.
- Techniques for Processing and Analyzing Traces: **Google Dapper**
  - **global request ID**: each event has a **trace ID** created at front-end service. It's **passed down** to subrequests.
  - representation: **Dapper trace tree**
    - nodes: **spans**, lifetime of a request. It has a **trace ID, parent ID and span ID**.
    - edges: temporal relationship of calls
  - trace collection: logs –> dapper collectors –> into central bigtable(trace IDs and span IDs)
  - possible challenges:
    - clock skew from different servers in distributed system.
    - **overheads**: caused by the massive amount of requests, trace generation and collection. Reduction by coalescing, adaptive sampling(application/trace id), asynchronous writes

13.3. Challenges in Monitoring: Overheads

- Definition: overhead is any **combination of excess or indirect computation time, memory, bandwidth, or other resources** that are required to perform a specific

task. Such resource **doesn't contribute directly** to the result, but it's **required** to make it work.

- Reasons:
  - collection of data: **all available**
  - instrumentation: insert additional binary calls/codes in the program and use these calls to trace native calls and achieve measurement goals
  - aggregation computation
  - memory overhead for buffering
  - storage overhead for long-term storage
- Reduction techniques:
  - reduce **number** of metrics: collect only important metrics
  - reduce measurement **frequency**
  - representation: choose **binary** instead of ASCII
  - message **coalescing**: package multiple messages into one message
  - **sampling**
  - long-term **coarsening**: after some time period, only store **aggregated** values

14. Autoscaling

- Motivation:
  - **Scalability**: ability when resource ↑, application performance ↑, throughput ↑, latency ↓
  - **Elasticity**: ability to **dynamically adapt**(up/down) the resource scale to the actual workload **without reboot**.

    → No under-/overprovisioning, cost ↓, customer satisfaction ↑
  - **Automation**: automatically, without interaction with the application owner
- Capacity planning in cloud: avoid under-/overprovisioning by **dynmaic resource management** and **pay-per-use cost model**

14.1. Scaling

- scaling performance:
  - speedup:

$$\text{speedup(p processors)} = \frac{\text{performance(p processors)}}{\text{performance(1 processor)}}$$

  - efficiency:

$$\text{efficiency(p processors)} = \frac{\text{speedup(p processors)}}{p}$$

- scalabilty limit: restricted by **app design**, maximum **application capacity** or maximum **resource capacity**

  → scaling with infinite servers impossible

  - **efficiency** of scaling **drops** when **more resources** are provisioned. → **bottleneck** of application gets dominant
  - **poor application design** has lower scalability limit.
  - more servers, communication latency increases.

- resource to be scaled: **CPU (#CPU, CPU time/percentage)**, memory, disk storage(size, bandwidth), network(throughput, bandwidth)

14.1.1. Vertical Scaling – Scaling Up

- increase capacity of a service instance by **increasing its allocated resources**. eg: CPU time percentage ↑, clock frequency ↑, cores ↑
- Advantages:
  - no re-design of app required
  - easy to replace, less management complexity

  Disadvantages:
  - powerful resources are expensive
  - **limited** scalability due to resource capacity limit

14.1.2. Horizontal Scaling – Scaling Out

- increase capacity of a service instance by **increasing the amount of same resources**.
- Advantages:
  - lower cost, implementation of commodity hardware
  - larger scalability possibility (just increase the amount of resource)

  Disadvantages:
  - more management overhead
  - distributed architecture required, load balancing, replica handling.

14.2. Autoscaling

- Goal: scale automatically to fulfill **Service Level Objectives**(SLO) while **minimizing costs**
  - latency of request
  - failed request rate
  - service availability

- Process: monitoring data –> analyzer –> scheduler takes scaling actions according to autoscaling policy –> executor gives cloud commands to scale up/down resources
- Autoscaling approaches:
  - **reactive**: **real-time** scaling according to policy(thresholds). detection of under/overloaded service
  - **scheduled**: **periodical** scaling, time-stamped scaling events. eg: banking system, food ordering app
  - **predictive**: predict future workload and scale **ahead of time**. It adapts proactively the **minimum** of autoscaling group.
- Autoscaling implementation:
  - **resource-centric**: scaling action directly modifies **resource allocation**(#VM, CPU, memory, bandwidth).
    - Use-case: AWS Reactive Autoscaling – scaling modifies #VMs.
  - **service-centric**: scaling action directly modifies **#service instances**.
- Autoscaling Policies (AWS):
  - **target tracking scaling**: a **target value** of a **metric** is predefined, eg: CPU utilization at 40%, #requests per target at 1000.

    Metrics that decrease when capacity increases and increase when capacity decreases can be used to **proportionally scale out or in** the number of instances using target tracking.
  - **simple scaling**: scaling action is triggered based on metric, threshold or condition.
    - **cooldown period exists**: responses to **no additional alarms** until cooldown period expires, wait until scaling result visible and health check.
  - **step scaling**: scaling action is triggered based on metric, threshold or condition with **step adjustments**. The adjustments vary based on the **size of alarm breach**.
    - **no cooldown period**: continues to **respond to additional alarms**, even while a scaling activity or health check replacement is in progress. Therefore, **all alarms that are breached are evaluated**

14.3. Load Balancing

- for **scaling out**(horizontal scaling)
- Goals:
  - efficient resource utilization
  - increase service availability (health check, restart nodes)
  - increase aggregated capability of replicas → reduce response time and failure rate
- Implementation: **multi-layer** load balancing

  - different **layers** for different **functions**: service layer(request distribution), virtual machine layer(VM distribution), physical layer(server, CPU, storage, bandwidth distribution)
  - **static VS. dynamic**:
    * static: scheduling according to **weighted round-robin**, **no feedback** about current load of servers
    * dynamic: **feedback** from server to load balancer, able to adapt according to feedback.
  - scalability: destributed design for load balancer

- Algorithms:
  - class-aware: **classification of requests** into sensitive, best-effort
  - content-aware: direct **similar request content** to same server
  - client-aware: based on **packet source**. Clients from similar area might share similar information, system can already **cache** the info.