

# Cloud Applications Architecture

• • •

Course 1 - Intro (Course & Cloud)

# Why Would You Be Interested?

Cloud is similar to the whole IT industry - relevant for all other industries.

Consistently in top 3 most demanded (hard - as opposed to soft) skills.

It will be useful no matter what you choose to do after.

We will have fun.

# What Does CAA Mean?

Designing and building applications (IT systems) which leverage the possibilities (services) offered by cloud environments to achieve greater:

- Scalability
- Security
- Cost Efficiency
- Availability
- Productivity

In other words, build systems we can trust and have a good time while doing it.

# General Info

## Course

- Focused on theoretical concepts, case studies, and demos
- Probably 50% of final grade
- Involvement is worth more than learning for the exam
- Most likely quiz exam using Moodle (maybe some oral questions?)

## Lab

- Hands-on with AWS
- Migrate an app from **on-premises** to **cloud**
- 50% of final grade
- Architecture design quiz

# General Info

- Topics Overview

# What Is the Traditional Approach?

- For individuals and small companies: develop a monolithic app, contact a hosting company, rent a server, install and configure relevant tools, copy the app with FTP, run the app.
- For large corporations: Discuss the budget, approve the budget, build a data center, go over the budget, hire people to manage the data center, deploy apps, upgrade data center.

# Cloud Intro

# What does Cloud mean?

In simple terms, cloud computing means on-demand availability of various computing resources. Typical benefits of using the cloud include:

- Flexibility
- Cost
- Speed
- Security
- Performance
- Reliability
- ...

# History

First public offering came from a large company (Amazon) as a result of over-provisioning (to handle holiday sales) in 2006.

In 2008, Google launched App Engine, one of the first public PaaS solutions.

# Main Providers



Google Cloud



# Gartner Magic Quadrant

Figure 1. Magic Quadrant for Cloud Infrastructure and Platform Services



Figure 1. Magic Quadrant for Cloud AI Developer Services



# Cloud Classification

## Deployment Models:

- Public
- Private
- Community
- Hybrid

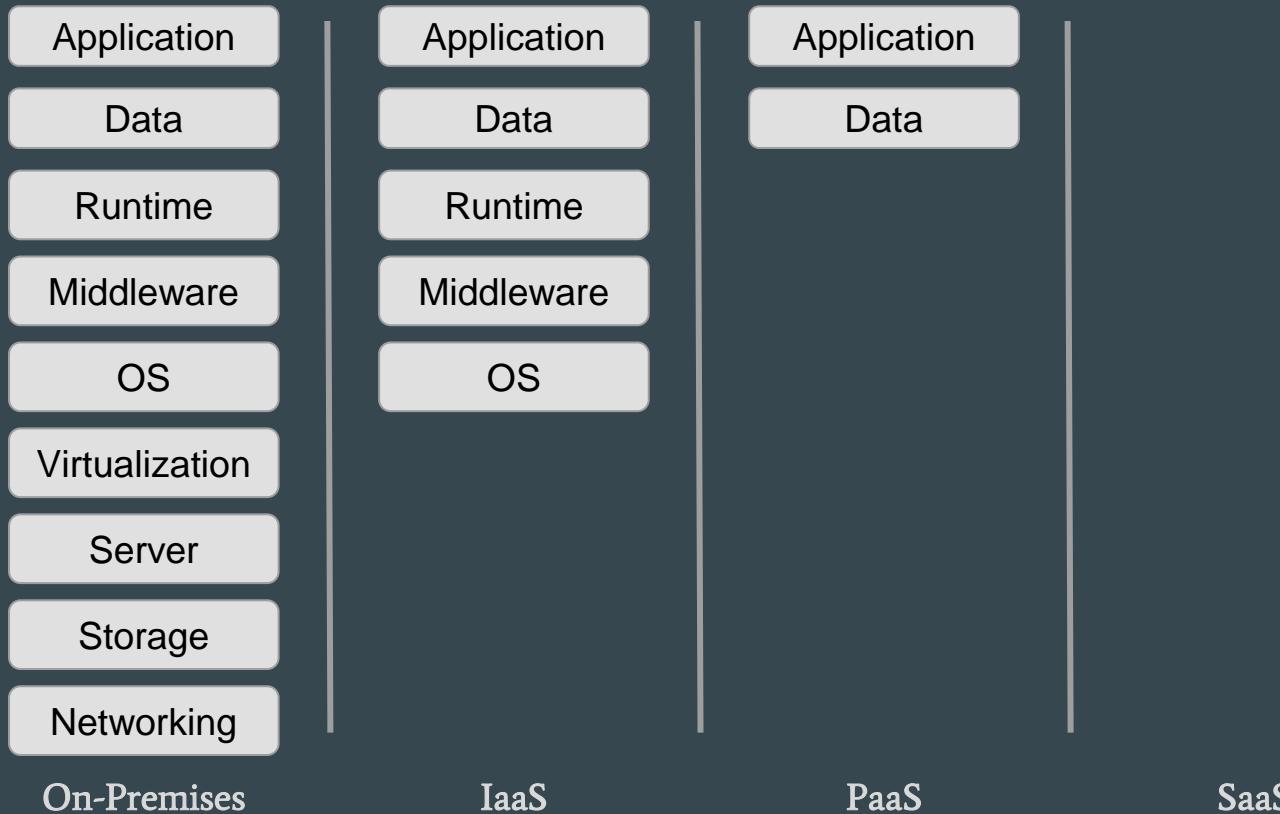
## Service Models:

- **IaaS (infrastructure as a service)**
- **PaaS (platform as a service)**
- **SaaS (software as a service)**

Many \*aaS nowadays: BaaS, DBaaS, FaaS (most of them are PaaS)

The NIST Model

# IaaS, PaaS, SaaS - What We Manage



# Examples for Each Service Model

## IaaS

Usually virtual machine services. Other service models are built on top of them.

E.g. Amazon EC2, Google's GCE etc.

## PaaS

Services that let you develop your solution without worrying about how the system runs.

E.g. Cloudfoundry, Google App Engine, Azure App Service, managed databases (RDS, CloudSQL etc.)

## SaaS

No implementation, you just use the provided functionality. Interact with them through UI or API.

E.g. Google Suite, Office365, Stripe, Dropbox, Mailgun

# Working with the Cloud

1. Create an account
2. Setup billing
  - a. Most providers offer decent free tiers
3. Configure and launch resources
  - a. From the web console
  - b. Using the CLI
  - c. Infrastructure as Code
4. \*Stop/terminate them when not used
  - a. Usually billing stops when stopping the resources

# Standards/Compliance

Assure customers best practices are followed

- CSA (Cloud Security Alliance) - Best practices for cloud security
- ISO 9001 - Global Quality Standard
- ISO 27001 - Information Security Management
- ISO 27017 - 27001 in the context of cloud
- ISO 27018 - Personal data protection in cloud
- PCI DSS - Card payments security
- HIPAA - Health information protection

Running on complaint infrastructure makes our products also compliant

# Pricing Models

Most services are billed based on the actual usage:

- **Time based:** pay per second/hour etc of the service being online
  - Usually applies to IaaS and PaaS offerings
- **Usage based:** pay per processing unit (request, transaction, read, write etc)
  - Mostly for PaaS/SaaS

Prices can be **fixed, tiered** and/or **dynamic**.

There is also the **subscription** based model: agree to pay each month a certain price and have access to the agreed services. Usually applies to SaaS offerings and some providers - e.g. SAP offers both

# Pricing Models - Time vs Usage

## Time Based

Usually applies to virtual machines, (relational) databases and general PaaS services (e.g. Cloudfoundry).

You pay the same for any amount of traffic.

Easy to control costs.

## Usage Based

Usually applies to fully managed services (FaaS, DBaaS - usually NoSQL) and storage services.

You pay based on the traffic.

You cannot throttle the usage without stopping the service.

Some services offer both - e.g.  
Azure API Management

# Pricing Models - Fixed, Tiered, Dynamic

## Fixed

You pay the same no matter what. Usually applies to on-demand virtual machines

## Tiered

You pay less per unit the more you use it. Usually applies to storage services and bandwidth cost.

## Dynamic

Usually based on supply and demand. More spare capacity, smaller prices.

Instance	vCPU(s)	RAM	Temporary storage	Pay as you go
B1S	1	1 GiB	4 GiB	€0.0135/hour
B1MS	1	2 GiB	4 GiB	€0.0234/hour
B2S	2	4 GiB	8 GiB	€0.0473/hour

### Azure Windows Virtual Machines Pricing

Storage pricing		
S3 Standard - General purpose storage for any type of data, typically used for frequently accessed data		
First 50 TB / Month	\$0.023 per GB	
Next 450 TB / Month	\$0.022 per GB	
Over 500 TB / Month	\$0.021 per GB	

	Linux/UNIX Usage	Windows Usage
General Purpose - Current Generation		
a1.medium	\$0.0084 per Hour	N/A*
a1.large	\$0.0234 per Hour	N/A*
a1.xlarge	\$0.0336 per Hour	N/A*
a1.2xlarge	\$0.0672 per Hour	N/A*
a1.4xlarge	\$0.1343 per Hour	N/A*
a1.metal	\$0.13	Linux/UNIX Usage
t2.micro	\$0.00	Windows Usage
General Purpose - Current Generation		
t2.small	\$0.00	
a1.medium	\$0.005 per Hour	N/A*
a1.large	\$0.0098 per Hour	N/A*
a1.xlarge	\$0.0197 per Hour	N/A*
a1.2xlarge	\$0.0394 per Hour	N/A*
a1.4xlarge	\$0.0788 per Hour	N/A*
a1.metal	\$0.0788 per Hour	N/A*
t2.micro	\$0.0035 per Hour	\$0.0081 per Hour
t2.small	\$0.0069 per Hour	\$0.0159 per Hour

# Pricing Models - Discounts

## Credits

Most providers offer **sign-up credits**, **educational credits** and **start-up credits**. You can also earn them from various contests or partnerships.

They also provide always free service tiers.

## Reservation

If you know you will be using a certain service for a long time, you can commit to it and get a considerable price cut.

E.g. AWS Reserved Instances and Savings Plans can reduce the cost by 30 to 70%

## Sustained Use

Some providers will automatically apply discounts to your bill if you use certain resources continuously.

E.g. Google Cloud sustained use discounts for GCE and App Engine (30% if the instance is running the entire month)

# Guidelines & Principles

# Architecture Frameworks

AWS Well-Architected

Google Architecture Framework

Azure Well-Architected

**Operational Excellence**

- Automation, Monitoring, Testing

**Security**

- IAM, Defence in depth, Traceability

**Reliability**

- Monitoring, Scaling, Testing

**Performance**

- Scaling, Choosing right, Design

**Cost Optimization**

- Choosing right, scaling, Understanding the services and offers

# The 12-Factor App

[Link](#)

Codebase

Port Binding

Dependencies

Concurrency

Config

Disposability

Backing Services

Dev/Prod Parity

Build, release, run

Logs

Processes

Admin Processes

[Pragmatic video about it](#)

# Relevant Links

1. [List with engineering blogs from top companies](#) (most relevant: Netflix, Uber, Atlassian)
2. [AWS re:Invent 2019 Playlist](#)
3. [Google Cloud Next `19 Playlist](#)
4. [Microsoft Developer Youtube Channel](#)
5. [Last Week in AWS](#)

# Summary

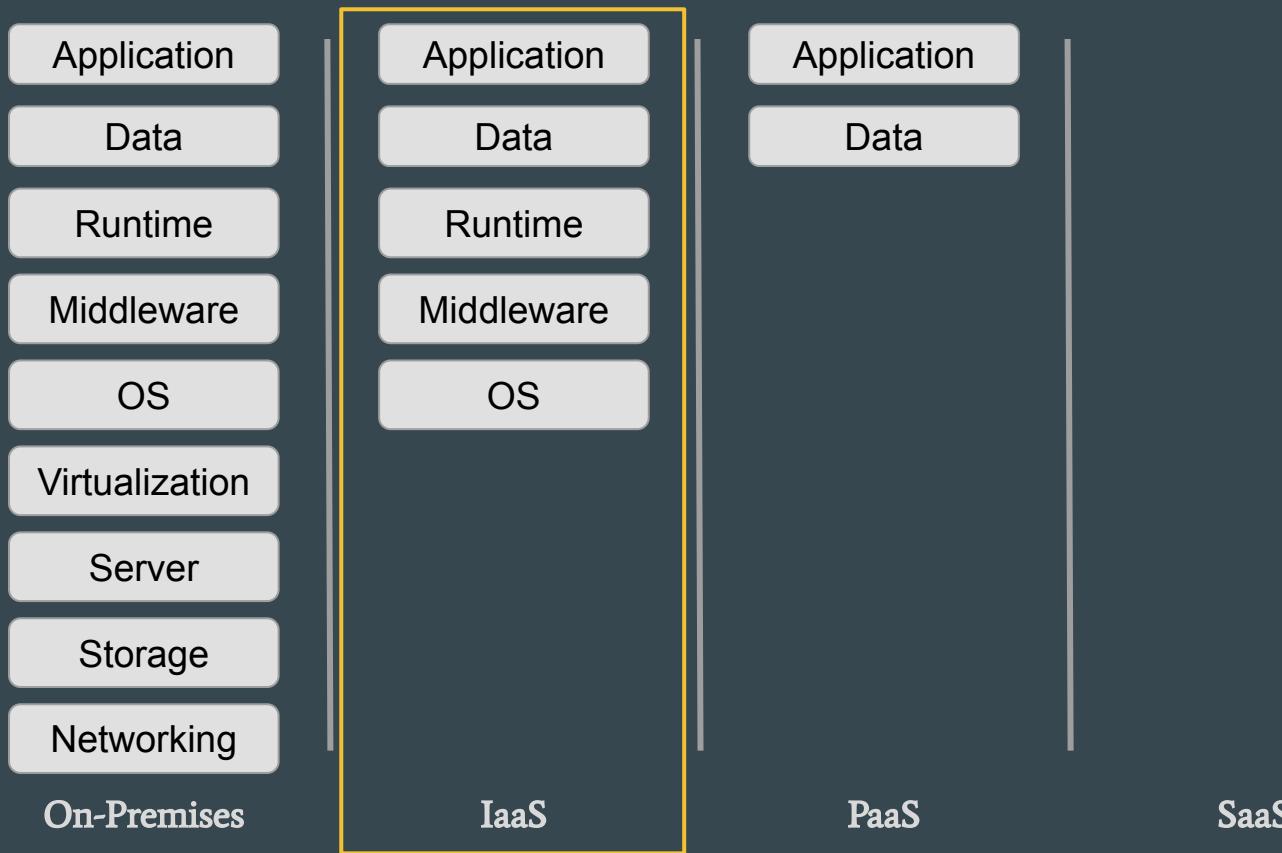
1. Why you would care
2. Involvement is important
3. The cloud and how it compares to on-premises
4. Providers
5. Cloud classification
6. Standards
7. Pricing models
8. Principles

# Cloud Applications Architecture

• • •

Course 2 - Infrastructure as a Service

# IaaS, PaaS, SaaS - What We Manage



# What Really Is IaaS?

- A way to provision and use (mostly virtualized) infrastructure when and while needed (**on-demand**) without having to worry about hardware and maintenance.

We are still responsible for what happens on the infrastructure (e.g. what operating system it runs or whether it is up to date or not).

- The foundation on which other service models are built.
- Datacenter abstraction.

# Virtual Machines (VMs)

“The workhorse of the entire industry”

Everything we will see in this course is more or less based on VMs.

**Emulated computers.**

**A way to create dynamically configured machines on demand.**

Are also useful for **local development**.

# Virtual Machines (VMs)

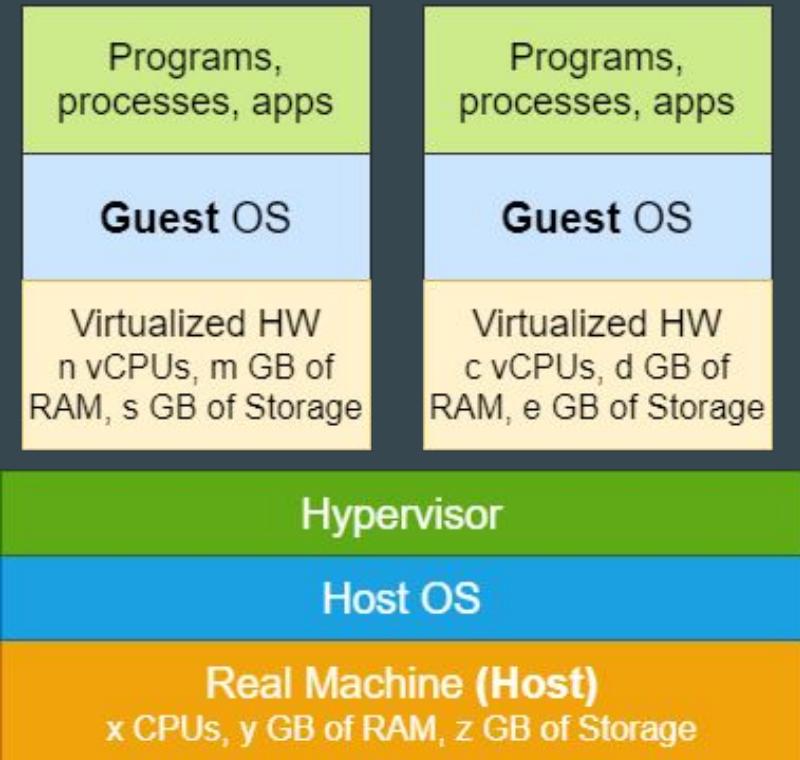
**Host:** the machine on which the VMs will be running.

**Guest:** the virtualized machine

**Hypervisor:** Creates and manages guest machines, translates and limits instructions.

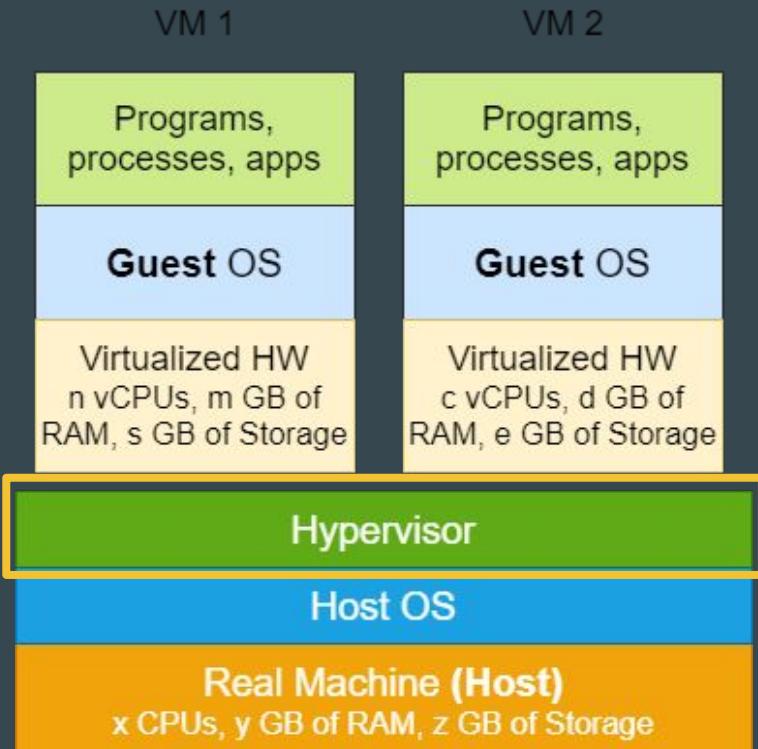
VM 1

VM 2



# The Hypervisor

- Optimize resource usage
  - lower provider cost => lower customer cost
- Enable multi-tenant usage (VMs for multiple clients on the same machine)
  - Optimize costs
  - **Ensure isolation**
- Most providers have custom hypervisors (AWS - Nitro, Azure - Azure Hypervisor/Customized Hyper-V, GCP - KVM).
- Providers might also support other hypervisors, e.g. **VMWare** (easier migration).



# Virtual Machines - Configurations

You choose the configuration based on your needs.

- General purpose
- CPU heavy
- Memory heavy
- GPU

Providers might give a set of pre-configured options or allow you to customize one.

# vCPUs

Highly dependant on the provider.

Might be:

- Timeshares of real CPU cores
- Physical core
- Logical core (think of hyperthreading) (AWS, GCP)

# Virtual Machines - OSs

Linux (most common flavors + provider specific)

Windows Server

Optimized OSs (e.g. container optimized OS)

Even Mac OS (e.g. macincloud)

You still have to buy licenses for certain OSs (e.g. Windows)

# Virtual Machines - Remote Access

1. **SSH** (Linux VMs)
2. **RDP** (Windows VMs)

Usually a security concern.

Production VMs are usually accessed through **bastion hosts**.

- Intermediary VMs with the sole purpose of granting access

# Virtual Machines - Monitoring

Why monitor?

- Business continuity
- Respecting contracts (SLA)
- Know when to scale (up/down, in/out)
- Understand your application
- Understand your users

# Virtual Machines - Monitoring

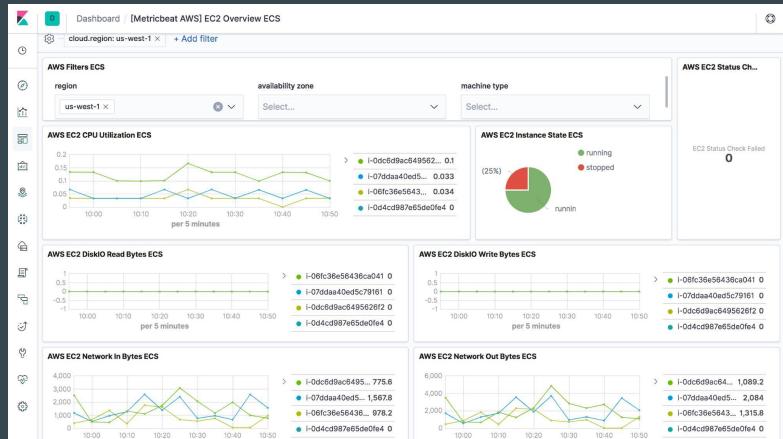
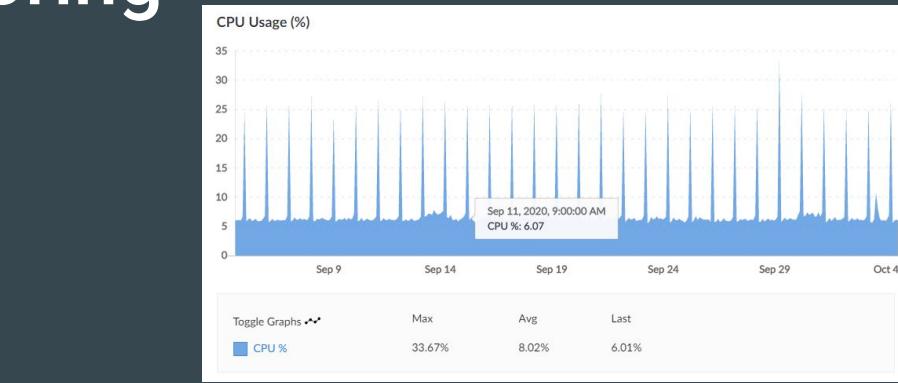
Commonly monitored metrics:

- CPU Usage
- Disk Usage
- Network Traffic (inbound/outbound)
- Memory

# Virtual Machines - Monitoring

## Tools:

- Some are already provided by the hypervisor itself
- Provider specific tools
  - CloudWatch
  - Azure Monitor
  - Google Operations (previously known as Stackdriver)
- The **ELK** Stack



# Virtual Machine Services - Examples



EC2



Azure VM



GCE



Droplets



Oracle VM

...and others

# Virtual Machines - Automation

# Infrastructure as Code (IaC)

**Resources:**

**ExampleEC2Instance:**

**Type:** AWS::EC2::Instance

**Properties:**

**ImageId:** ami-0ff8a91507f77f867

**InstanceType:** t2.micro

AWS CloudFormation

```
{  
    "type": "Microsoft.Network/virtualNetworks",  
    "apiVersion": "2020-06-01",  
    "name": "[variables('virtualNetworkName')]",  
    "location": "[parameters('location')]",  
    "dependsOn": [  
        "[resourceId('Microsoft.Network/networkSecurityGroups',  
variables('networkSecurityGroupName'))]"  
    ],  
    "properties": {  
        "addressSpace": {  
            "addressPrefixes": [  
                "[variables('addressPrefix')]"  
            ]  
        },  
        "subnets": [  
            {  
                "name": "[variables('subnetName')]",  
                "properties": {  
                    "addressPrefix": "[variables('subnetPrefix')]",  
                    "networkSecurityGroup": {  
                        "id":  
                            "[resourceId('Microsoft.Network/networkSecurityGroups',  
variables('networkSecurityGroupName'))]"  
                    }  
                }  
            }  
        ]  
    }  
}
```

Azure ARM Template

```
resource "google_compute_instance" "default" {  
  name          = "test"  
  machine_type = "n1-standard-1"  
  zone         = "us-central1-a"  
  tags          = ["foo", "bar"]  
  boot_disk {  
    initialize_params {  
      image = "debian-cloud/debian-9"  
    }  
  }  
  // Local SSD disk  
  scratch_disk {  
    interface = "SCSI"  
  }  
  network_interface {  
    network = "default"  
  
    access_config {  
      // Ephemeral IP  
    }  
  }  
  metadata = {  
    foo = "bar"  
  }  
  metadata_startup_script = "echo hi > /test.txt"  
  
  service_account {  
    scopes = ["userinfo-email", "compute-ro", "storage-ro"]  
  }  
}
```

Terraform

# Boot Up Scripts

Terminal commands executed during the **first boot** of the VM.

- Usually with admin/root privileges

Used to install various programs and configure the system (e.g. create an user).

Impact the boot time.

# Images

Manually configure a VM, create image from it, spin up identical VMs in seconds.

Pay for the storage

Best practice (aws: Golden AMI)

Can be sold (marketplaces)

# Dedicated Services/Tools

Automatically configure infrastructure and/or system configuration and dependencies.

Usually require an agent to be installed.

Some require a master node (separate VM).



# Storage

Storage is usually separated from the VM.

One of the major differences when migrating to cloud

**VM = CPU + RAM, needs storage**

**Volumes can be attached to different VMs.**

# Storage

Block storage (as opposed to object storage)

Boot disks (where the OS is installed), additional storage

SSD, HDD, Magnetic (can be attached to one VM at a time)

Encryption

Snapshots

Instance store (ultra high throughput, but usually temporary)

NAS (multiple VMs can use it at the same time)

# Networking

Some provide more than others (only public instances to fully customizable network topology). We will focus on the latter.

Similarly to storage, network cards are attached to VMs. Primary and secondary.

More details in next course

# Working with IaaS

1. Create a VM using one of the tools made available by the provider
  - a. Web console
  - b. CLI
  - c. SDK
  - d. API
2. Connect to it (optional)
3. Install the necessary programs (can and should be automatized)
  - a. Application dependencies
  - b. Monitoring software
4. Deploy your application (can and should be automatized)
5. Monitor the VM and app
6. Keep the VM (OS + programs) and app up-to-date

# Other Features

Deploy VMs configured by others

- Can be found on marketplaces (e.g. [AWS](#), [GCP](#), etc.)
- Usually leverage IaC tools

Cost optimization

- Preemptible VMs
- Long-term commitments
- Sustained usage
- Choosing the right specifications
  - Sometimes, the more expensive VM is actually cheaper

# When to Use IaaS?

- First migration phase(s) (lift & shift)
- “Special” programs
  - Usually native programs
- When ultra-high performance is a must
  - You can rent physical machines (actual racks)
- “Special” licenses
- When trying to avoid vendor lock-in at all costs (rarely worth)
- Certain long-running workloads
  - Usually lower cost than alternatives
- When you want to be in control

# Case Study

**OpenAI** (AI for playing and winning Dota 2)

**1v1 (First project) and 5v5 (current)**

Has a win rate of **99.4%** (people could play against it).



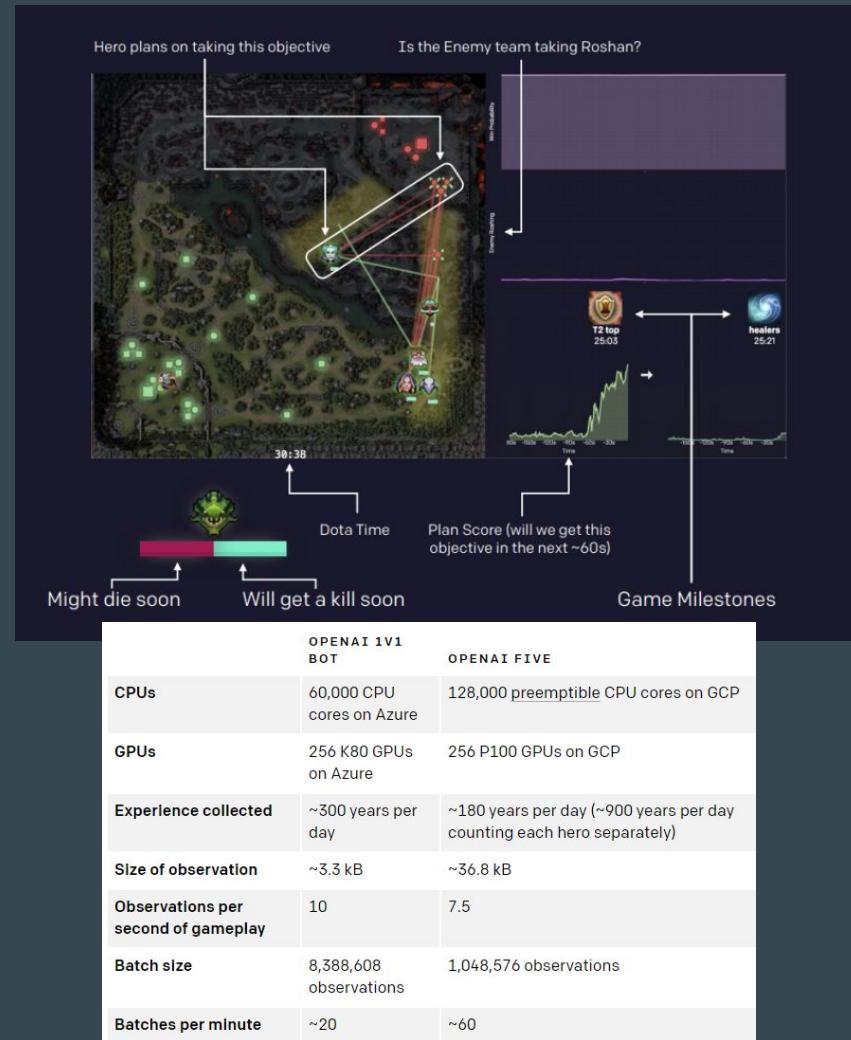
Bill Gates  
@BillGates

#AI bots just beat humans at the video game Dota 2. That's a big deal, because their victory required teamwork and collaboration – a huge milestone in advancing artificial intelligence.

via Twitter

## Timeline of events

~\$25k / day



# Summary

1. The responsibility model
  - a. OS, other tools/programs, for setting up our software, keeping everything up to date
2. The **hypervisor**, the guest, the host machine
3. Configure the VM based on our needs
4. OS (linux, container optimized OSs, windows)
  - a. SSH/RDP for connecting to the VM
5. Monitoring (choose the relevant metric)
  - a. ELK stack to handle monitoring for multiple VMs (allows us to get a better overview)
  - b. Munin
6. Automation: IaC, boot up scripts, images, tools for provisioning software
7. Storage, Networking

# Cloud Applications Architecture

...

Course 3 - Networking in the Cloud

# Networking Concepts

# IP Addresses

Public, private

Static, dynamic

IPv4, IPv6,

# IP Addresses

## IPv4

~4 billion addresses.

We are close to running out of (think of all the IoT devices).

Consists of 4 octets.

e.g. **10.120.70.1**

## IPv6

many orders of magnitude more addresses

`2001:0db8:85a3:0000:0000:8a2e:0370:7334`

A device may have both.

IPv6 is usually optional in cloud.

# IP Addresses

## Public

Unique across the internet.

Not in the private/reserved ranges

## Private

In one of the ranges:

- 10.0.0.0 – 10.255.255.255
- 172.16.0.0 – 172.31.255.255
- 192.168.0.0 – 192.168.255.255

A device may have both.

Public is usually optional in cloud

# IP Addresses

## Static

The device is guaranteed to have the same IP address.

Great for servers.

## Dynamic

The device is **not** guaranteed (and very **unlikely**) to have the same IP address.

Used in most cases.

Assigned by DHCP (dynamic host configuration protocol)

# Routing

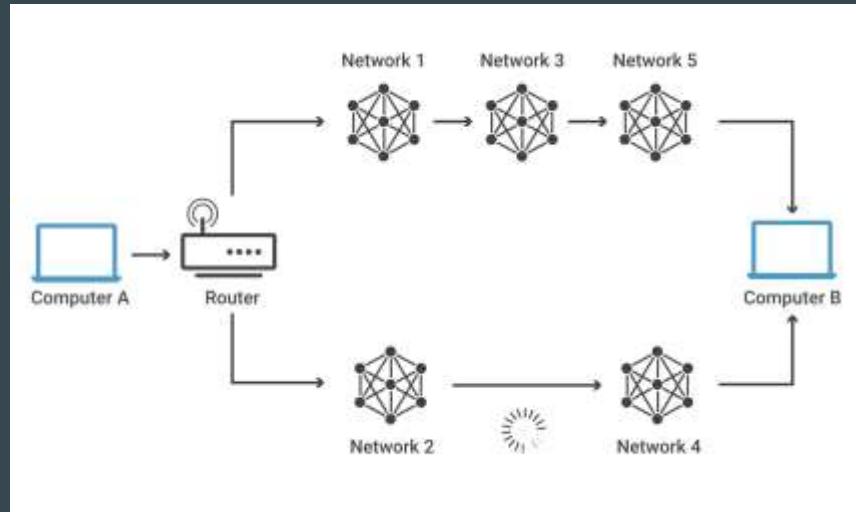
Selecting the path to reach the destination.

Protocols used by the internet:

- IP
- **BGP**
- OSPF
- RIP

Multiple delivery schemes:

- Unicast
- Broadcast
- **Multicast, anycast, geocast**



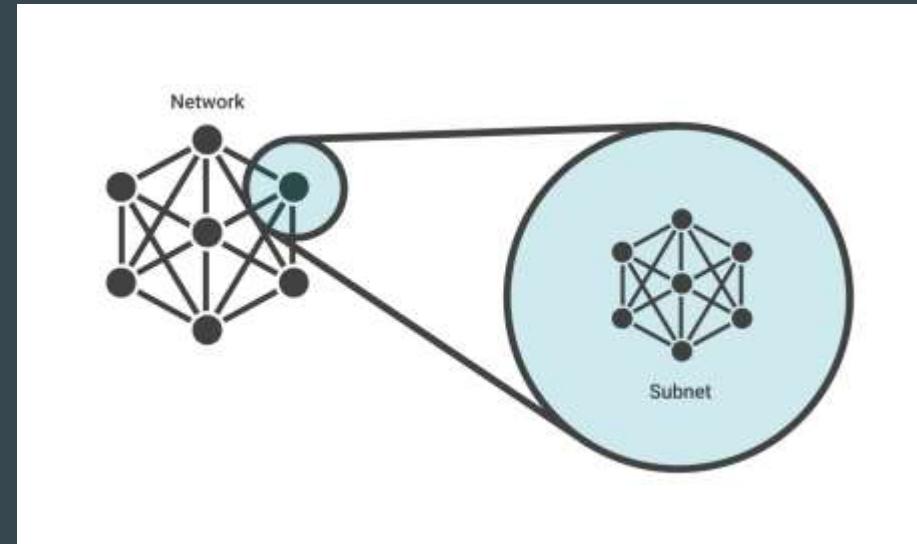
# Subnets

A part (subnetwork) of a network.

Traffic can stay within one subnet

- better performance
- less congestion)

First part of a given IP address indicates the (sub)network. The other part indicates the device.



# CIDR

## Classless Inter-Domain Routing

Replaced classful networks in 1993 (classes A, B, C, D, E)

Example: 192.168.0.1/24

/24 tells us that the the first 24 bits indicate the network. So to find the network address, we apply the **network mask** 255.255.255.0 (bitwise AND).

The last 8 bits indicate the device within the network. (0 and 255 are reserved). This means that we can have 254 devices in this network.

# CIDR - example

/27 -- 8 Subnets -- 30 Hosts/Subnet		
Network #	IP Range	Broadcast
.0	.1-.30	.31
.32	.33-.62	.63
.64	.65-.94	.95
.96	.97-.126	.127
.128	.129-.158	.159
.160	.161-.190	.191
.192	.193-.222	.223
.224	.225-.254	.255

	Addresses	Hosts	Netmask	Amount of a Class C
/30	4	2	255.255.255.252	1/64
/29	8	6	255.255.255.248	1/32
/28	16	14	255.255.255.240	1/16
/27	32	30	255.255.255.224	1/8
/26	64	62	255.255.255.192	1/4
/25	128	126	255.255.255.128	1/2
/24	256	254	255.255.255.0	1
/23	512	510	255.255.254.0	2
/22	1024	1022	255.255.252.0	4
/21	2048	2046	255.255.248.0	8
/20	4096	4094	255.255.240.0	16
/19	8192	8190	255.255.224.0	32
/18	16384	16382	255.255.192.0	64
/17	32768	32766	255.255.128.0	128
/16	65536	65534	255.255.0.0	256

# NAT

**Network Address Translation**

Translate IP addresses

Used for:

- Overlapping networks (networks spanning the same private IPs)
- Accessing the public internet from devices without public IP addresses

Can also translate ports

# Gateways

Allow reaching devices from different networks.

Networks usually have **default gateways** to which traffic is routed when the target is not found within the network.

# VPN

## Virtual Private Network

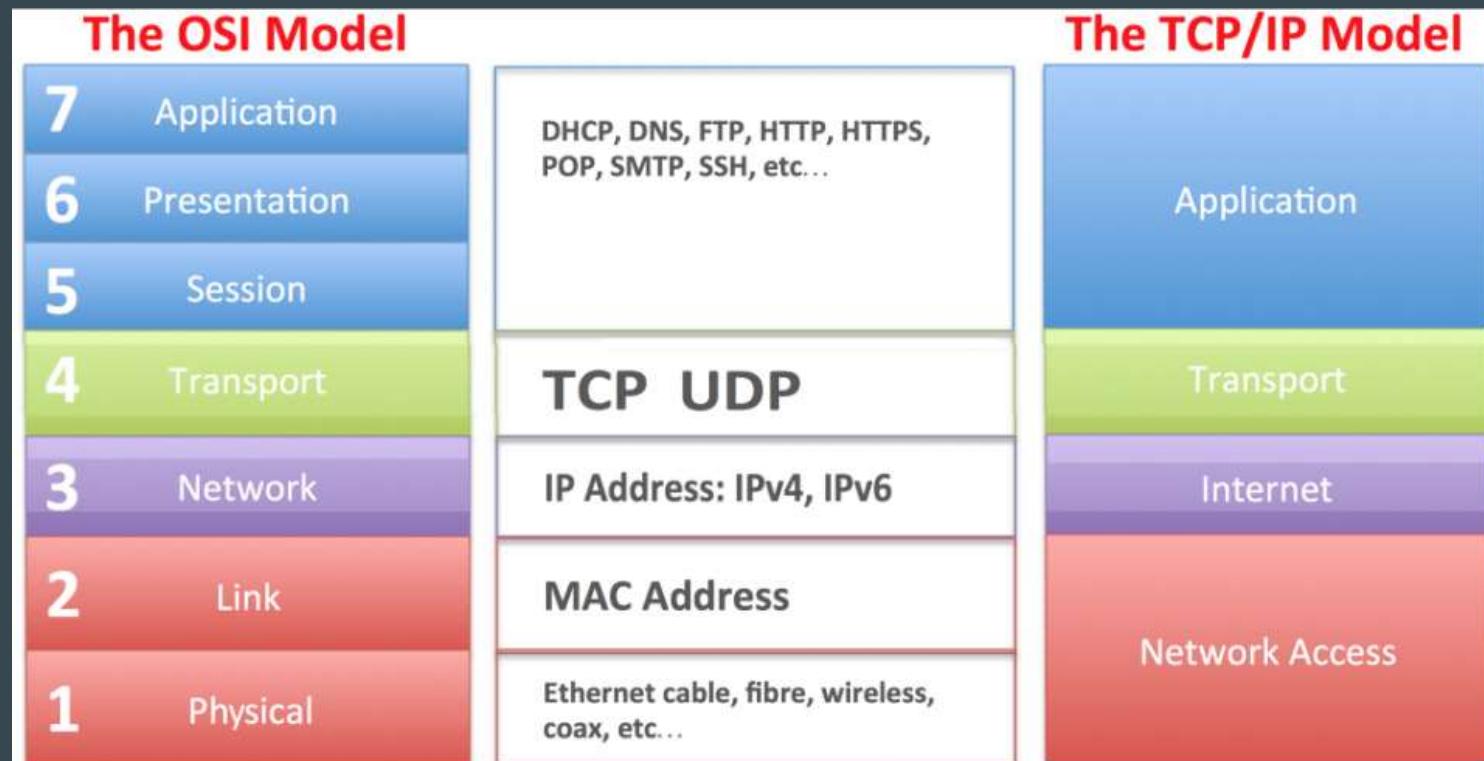
Connect securely to another network

Nowadays used to bypass censorship and watch movies from other regions

Initially designed for businesses

- Access the corporate network from outside (e.g. home, mobile)

# Networking Models



# Cloud Networking

# Context

Most relevant in the context of IaaS.

Constantly evolving.

- E.g. DigitalOcean released **VPCs** (virtual private cloud) in April 2020
- EC2 started without any custom networking support.

Highly dependant on the provider.

# Common Terms

## Virtual Private Cloud (VPC)

- VPC on AWS, GCP, and DigitalOcean, vNet on Azure, etc.
- Logical (**isolated**) network slice where we can deploy resources.
- Works with **private IP addresses** given by a **CIDR**
- Can usually be split into **subnets**

## Peering

- VPC peering usually
- Allows traffic between different networks
- It's our responsibility to avoid/handle **IP overlaps**
- Can also be between on-premises and cloud

# VPC Networking Components

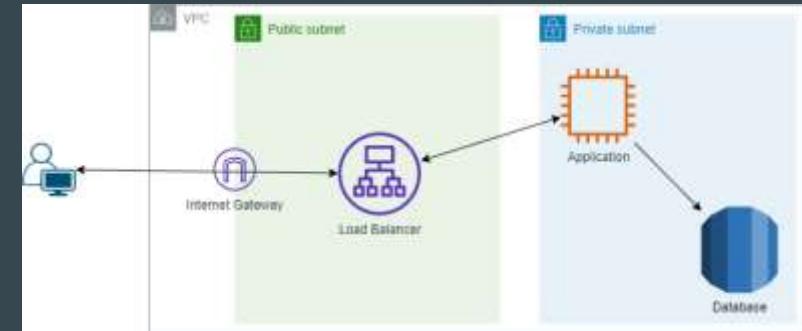
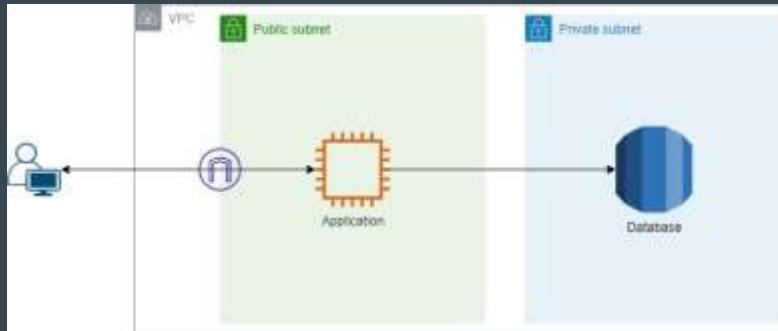
- Network interfaces
- Route tables
- Gateways: Internet, carrier, egress-only Internet
- NAT
- DNS

# Secure by Design

Based on the least privilege principle and minimizing the attack surface.

**Ideally**, a service should be **accessible** (network-wise) only by the services that are intended to access it.

E.g. a dedicated database server should be accessible only by the application server/tier. Any other traffic shouldn't even be possible.



# Common Vulnerabilities

Leaving the provider internal network to access managed services.

We cannot choose to deploy most managed services in a given network.

- They are reached via the public internet

Providers usually offer additional services/features designed to ensure private communication with the managed services.

# Network Performance

Subnetting helps

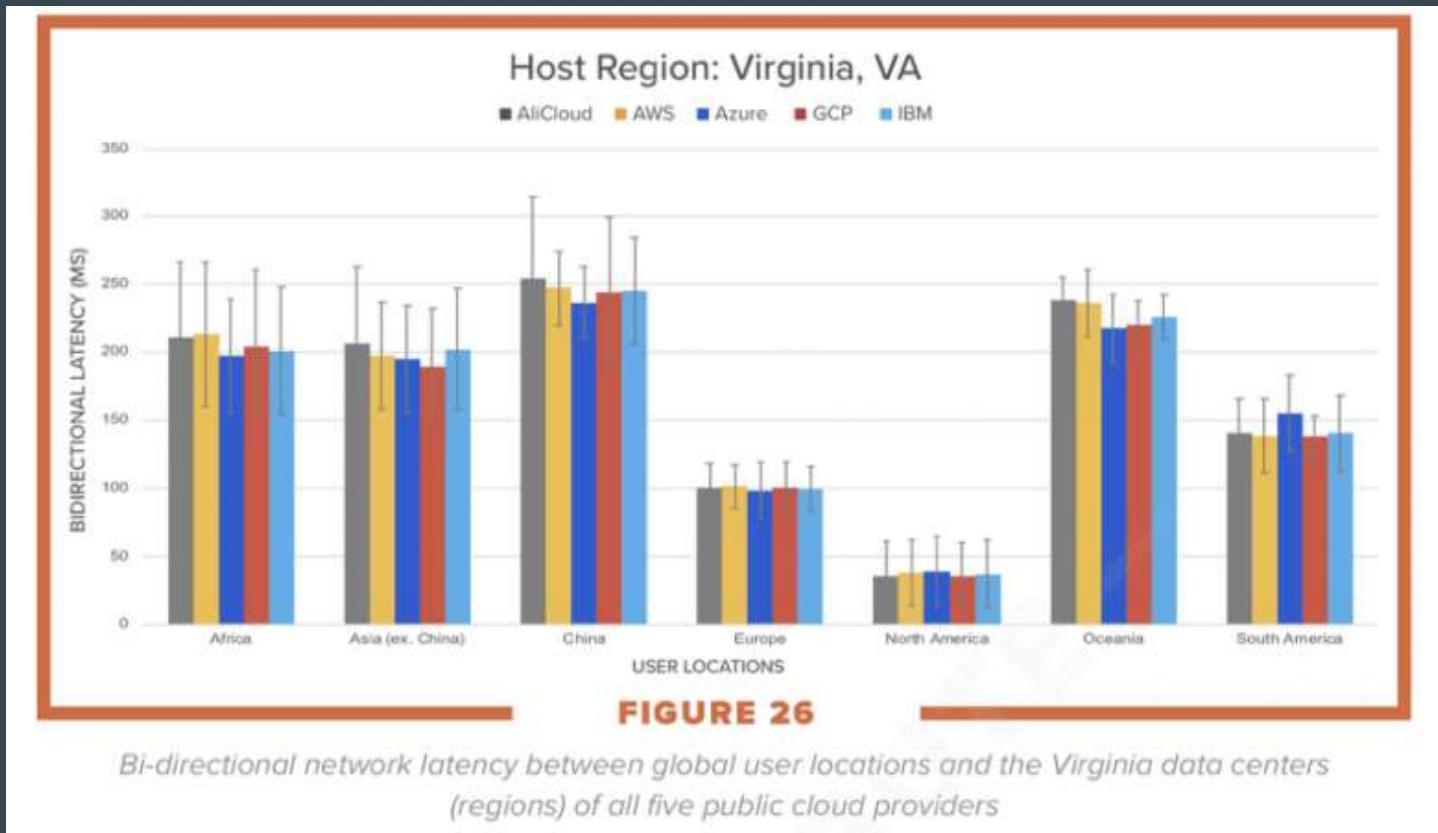
Bigger VMs tend to have better (virtual) networking cards.

Some providers support clustering (running multiple VMs on the same rack)

Our designs should strive to leverage the provider infrastructure

- Packets should enter the provider network as soon as possible

# Network Performance



source

# Firewalls

Security groups in AWS, Network security groups in Azure, VPC firewall rules in GCP... similar names for the other providers.

Used to allow or explicitly deny inbound/outbound traffic.

Can be configured per protocol (e.g. TCP, UDP, ICMP - HTTP, SSH etc are based on these 3), port, and source (e.g. IP address).

**Are attached to resources.**

They are used to filter external (e.g. from the internet) and internal traffic (e.g. from other VMs).

# Security Groups

Act as a virtual firewall for your instance to control inbound and outbound traffic

For each security group, you add *rules* that control the inbound traffic to instances, and a separate set of rules that control the outbound traffic

AWS:

You can specify allow rules, but not deny rules.

You can specify separate rules for inbound and outbound traffic.

Security group rules enable you to filter traffic based on protocols and port numbers.

# Network Access Lists (NACLs)

Are **stateless** (as opposed to firewalls which are usually stateful)

- I.e. both inbound and outbound must be allowed

Simpler, based on IP addresses

Usually support priority (lower = higher priority)

# Security Groups vs NACLs

Security Group	Network ACL
Operates at the instance level	Operates at the subnet level
Supports allow rules only	Supports allow rules and deny rules
Is stateful: Return traffic is automatically allowed, regardless of any rules	Is stateless: Return traffic must be explicitly allowed by rules
We evaluate all rules before deciding whether to allow traffic	We process rules in order, starting with the lowest numbered rule, when deciding whether to allow traffic
Applies to an instance only if someone specifies the security group when launching the instance, or associates the security group with the instance later on	Automatically applies to all instances in the subnets that it's associated with (therefore, it provides an additional layer of defense if the security group rules are too permissive)

# DNS

Domain Name System

Translates domains to IP addresses

Cloud providers usually offer their own services

- E.g. Google's Cloud DNS - one of the few services with 100% availability

Many companies have internal DNS

- Which might have to be migrated

More on DNS in course 5

# Packet Mirroring

Some providers offer services for this

- E.g. Google Cloud

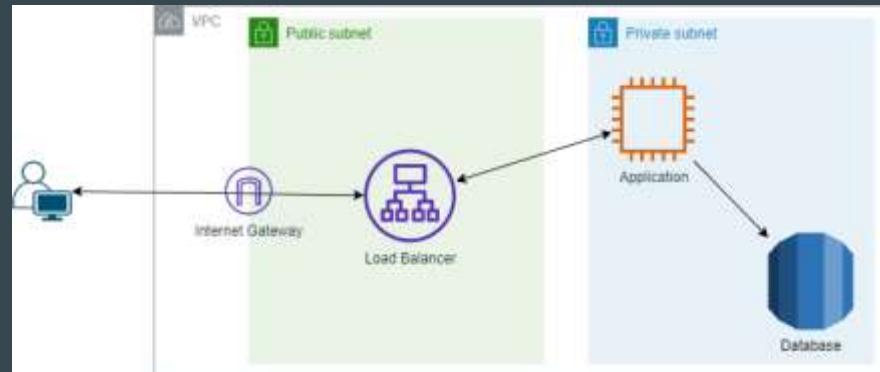
Can provide immense benefits for:

- Testing
- Recreating bugs
- Security analysis
- Anomaly detection

Does not impact traffic performance.

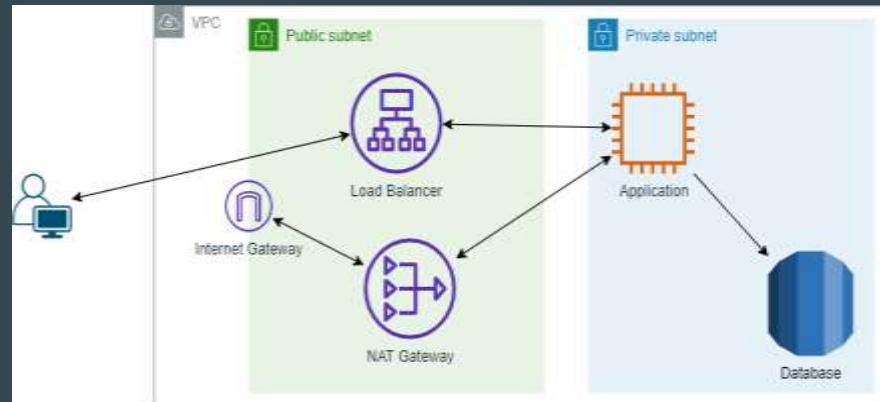
# Accessing the Public Internet

Problem: the instance has to path to reach the internet (e.g. to download updates)



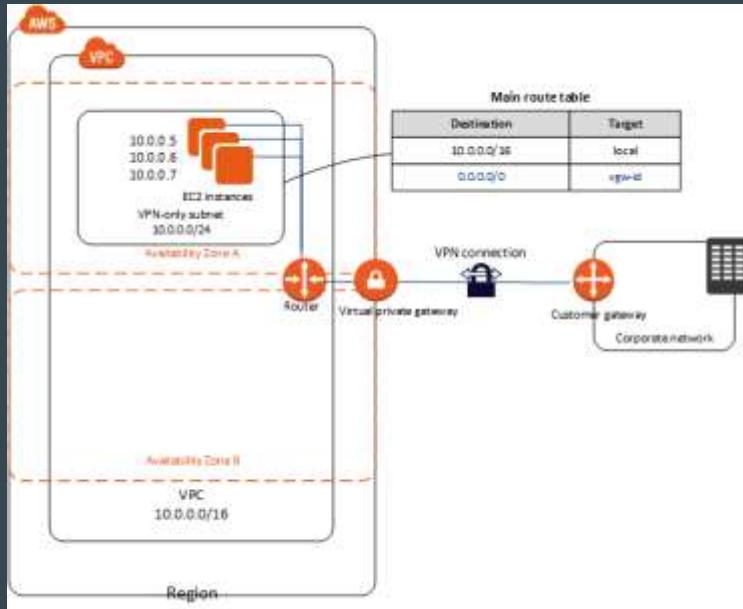
Solution: NAT

- Dedicated NAT services
  - E.g. NAT Gateway
- Configure a NAT instance

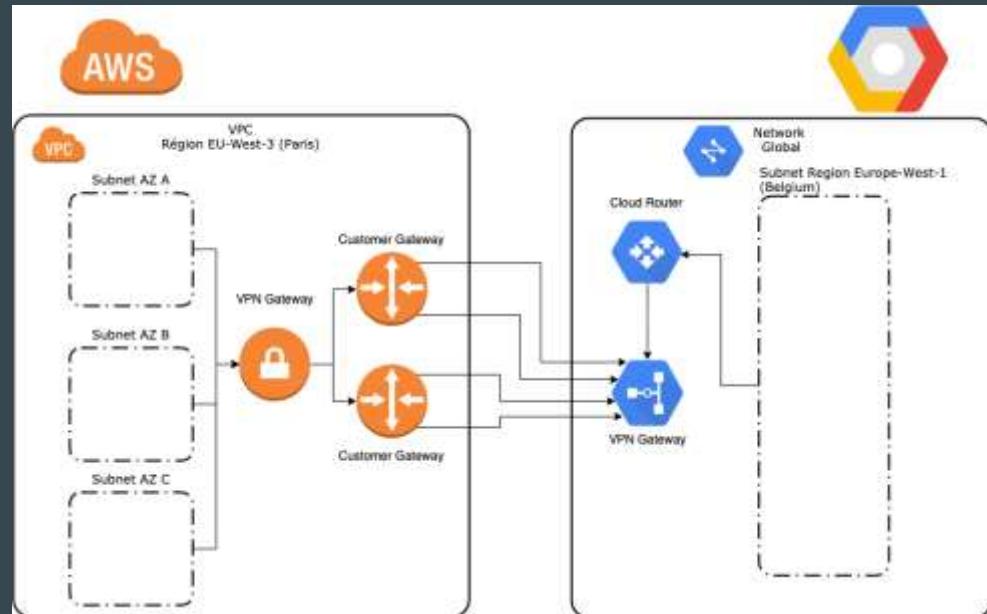


# Connecting to the On-premises Systems (or other Clouds)

# VPN Services



Example of VPN from On-Prem to AWS



Example of VPN from AWS to GCP

# Direct Connection Services

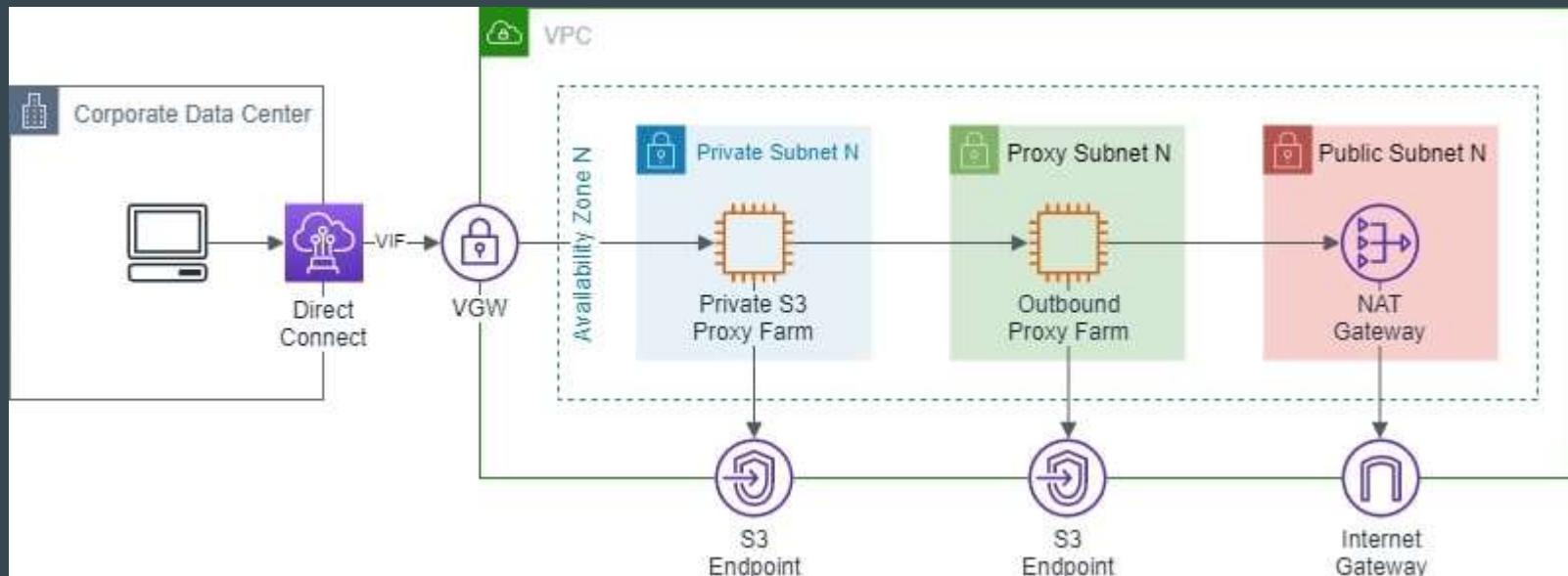
When VPN is not enough (usually performance-wise).

Is a physical connection between customer network and cloud provider.

Highest performance achieved when connecting directly to the cloud infrastructure, but partners also offer this (enough in most cases) (e.g. Orange and Vodafone).

# Network Architecture

Example of securely accessing public storage services from the corporate network



# Cloud Applications Architecture

...

Course 4 - Scalability

# What is a Scalable System?

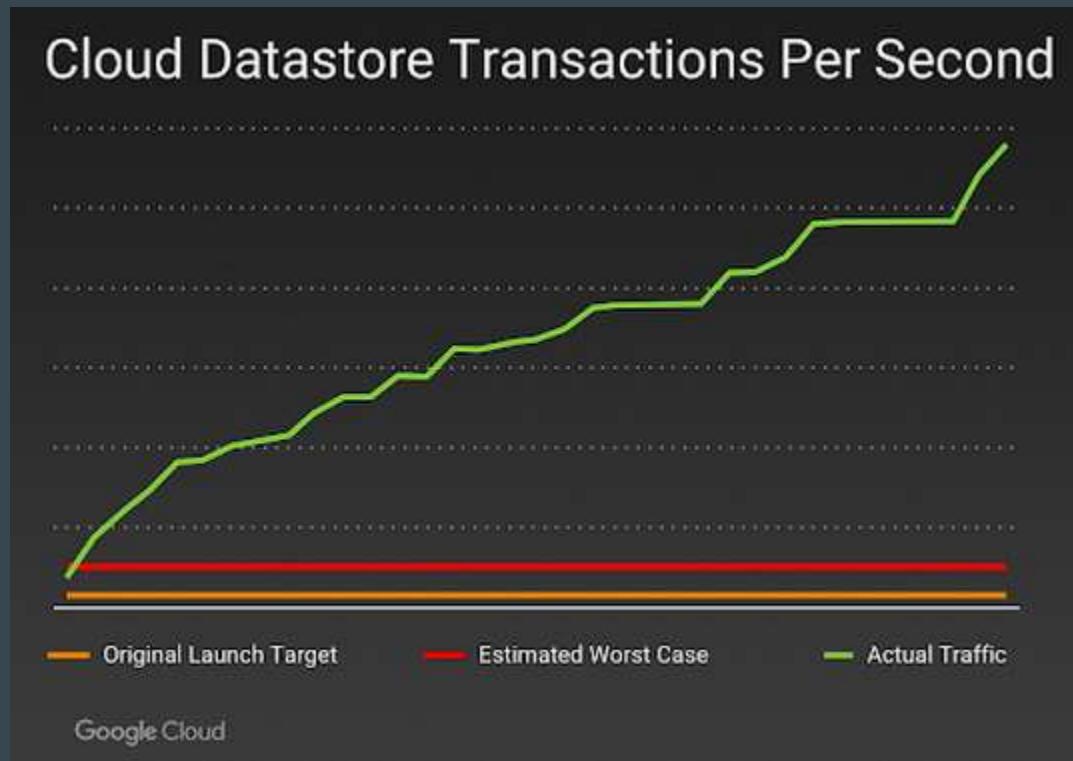
# Match the Required Load

How do we handle traffic spikes? (e.g. Black Friday)

How do we handle gradual growth of the product?

How can we minimize cost if our system is used only during certain timeframes?

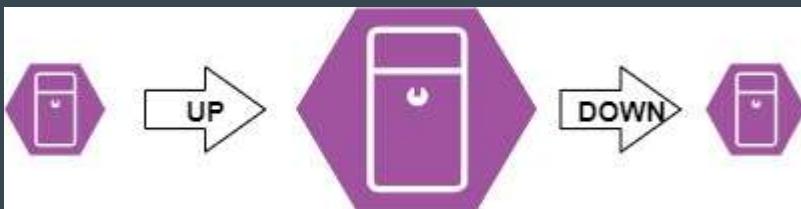
# Load Testing



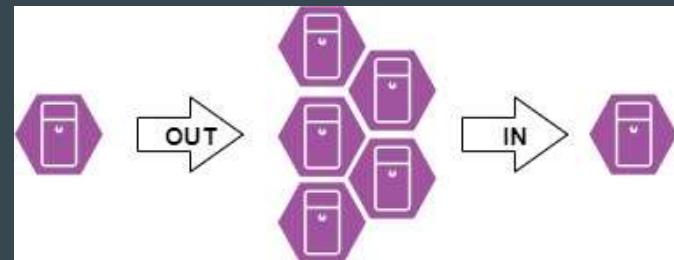
Pokemon Go launch day  
source

# Types of Scalability

**Vertical  
(Up/Down)**



**Horizontal  
(In/Out)**



# Comparison

## Vertical

- Usually easier/straightforward to implement (update the VM).
- Compatible with any app architecture (e.g. stateful apps).
- Doesn't require additional components.
- Usually requires restarting the service.
- Might be the only approach for certain components (e.g. RDBMS)

## Horizontal

- Infinite scale.
- Requires stateless apps (promotes better architectures).
- Requires load balancers (not always).
- Does not touch already existing VMs (i.e. no downtime) (unless when scaling in).

# Context

We will focus on IaaS mostly (i.e. working with VMs).

While vertical scaling can definitely help, we will focus on horizontal scaling.

- Possibly unlimited scale
- Doesn't require restart
- More interesting to talk about

# Scaling Strategies - Dynamic

**Scale based on various metrics**  
provided by the monitoring  
software:

- CPU, # of requests, memory,  
etc.

Choose the right metric for your  
system.

E.g. for a chat app, # of  
connections might be more  
relevant than CPU utilization.



# Scaling Strategies - Dynamic

Aim to keep the load under 50% to have a buffer until additional VMs are ready

- Monitoring delay
- Scaling takes place only after sustained load.
  - E.g. scale up only if the load was above the threshold for 2 minutes
- Resource provisioning/initialization/setup
- The load balancer might take some time acknowledge the new VM
  - E.g. waits for the health check to pass at least 3 times

# Scaling Strategies - Scheduled

Useful for predictable load patterns such as:

- Low usage at night
- High usage from Wednesday until Friday
- Events

Should be used when possible since it avoids scaling delays

- A.k.a. Resources are pre-warmed
- Providers can also pre-warm certain managed resources
  - E.g. managed load balancers

# Traffic Routing

How do we route traffic when working with horizontal scaling?

Multiple approaches:

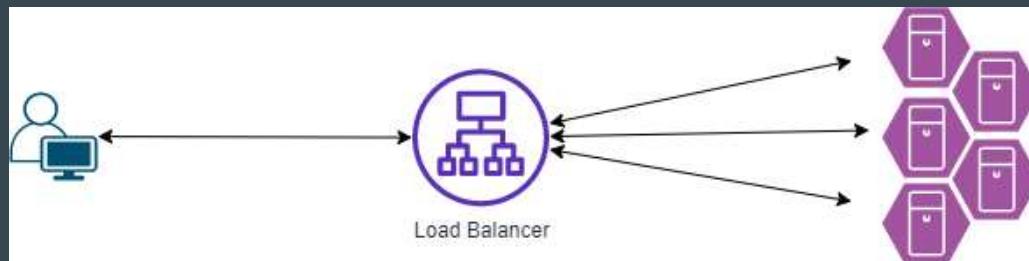
- Have a load balancer in front of the VMs
  - I.e. a single point of entry
- DNS load balancing
  - I.e. rely on the DNS service to redirect traffic to VMs
- Make the app aware of scaling
  - Rare, usually used for scaling databases

# Load Balancers

# Load Balancers

Serve multiple purposes:

1. Balance traffic across multiple targets
2. Monitor the health of target resources
3. Serve as a single point of entry
4. (Some) Caching
5. Others such as SSL offloading, authentication, A/B testing, etc.



# (Some) Examples

NGINX

HAProxy

Traefik

AWS Elastic Load Balancer (Application LB/Network LB)

Google Load Balancing

Azure's several (and confusing) services

# Types of Load Balancers

## HTTP (AKA Layer 7 LB)

- Works at layer 7 of the OSI model
- Offers most features
- Adds significant latency (~400ms)
- Behaves as a **Proxy** (**Sends new requests**)
- Good default choice

## TCP/UDP (AKA Layer 4 LB)

- Works at layer 4 of the OSI model - i.e. works with packets
- Behaves similar to a network router (the request is not decomposed)
- Ultra-high performance
- Not so many features

## DNS

- Also works at layer 7
- Useful for global load balancing
- A domain might be pointing to several IP addresses. Clients usually connect to the first one - send IPs in desired order.
- **Issues with caching/TTL**

# Benefits & Common Features

HTTP, HTTPS, WebSockets / UDP, TCP

SSL/TLS Termination

Monitoring

Single point of access

Enables auto-scaling

# Benefits & Common Features

Routing based on path, query strings, subdomains

Health checks

Port mapping

Integration with other services (e.g. authentication)

Caching

Cross-region

# (Some) Routing Strategies

## Round-Robin

- Simplest, route traffic in a loop. Good enough for simple use cases

## Weighted Round-Robin

- Prefer certain targets more than others (useful when target vary in sizes)

## Least Connections

- Useful when requests vary in load

# Application Layer Scalability

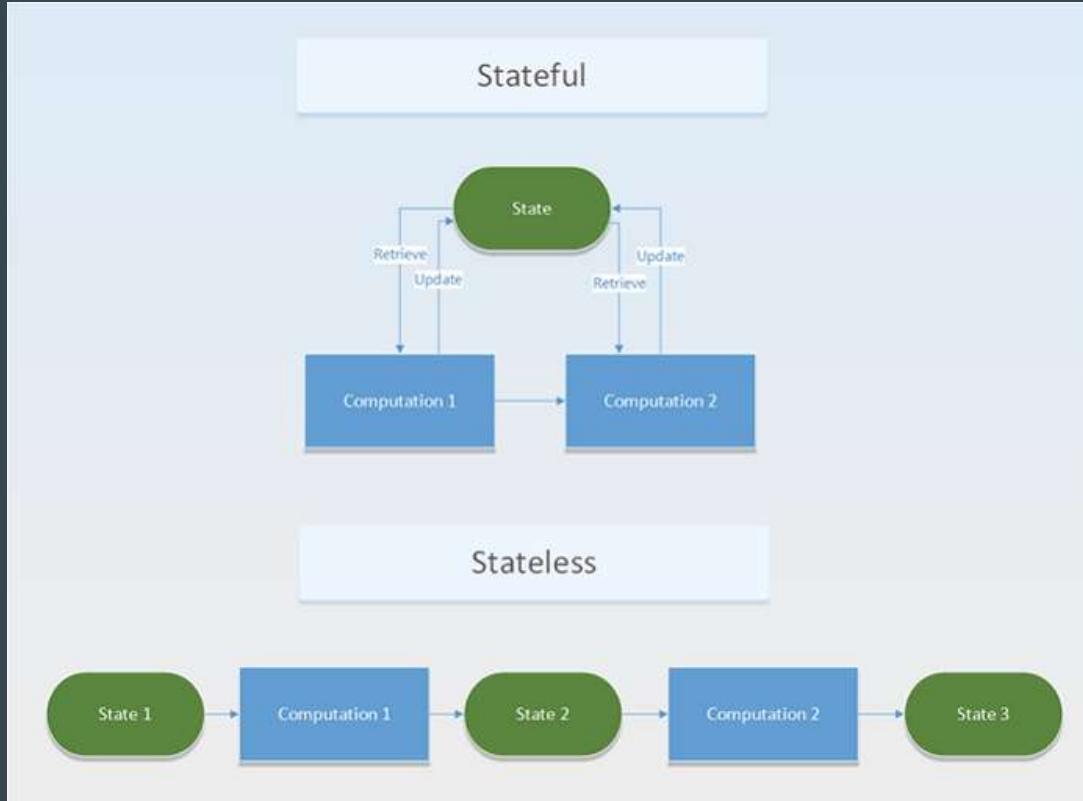
# Stateless Applications

Applications were traditionally stateful (e.g. authentication sessions/shopping carts handled by the application layer).

Stateless applications handle requests independently.

- They don't rely on previous requests
- The entire context is deduced based on the current request).

# Stateless Applications



[source](#)

# Pets vs Cattle

Rather cruel, but good analogy.

Pets are hand-built servers that we cannot just get rid of.

Cattle, on the other hand, can easily be replaced.

**Automation** is the main point.

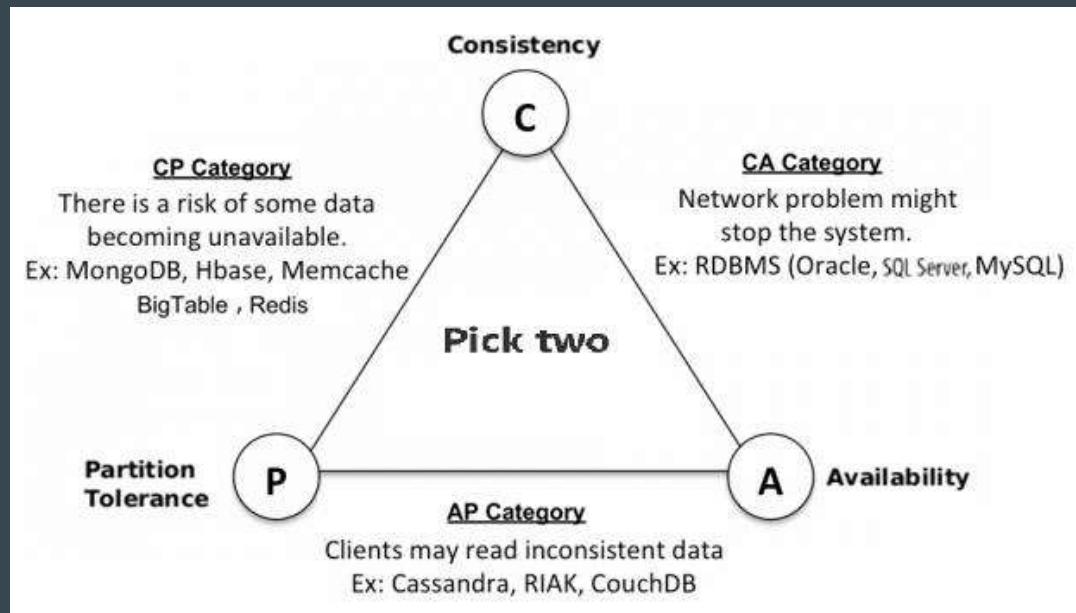
# Database Layer Scalability

# Context

NoSQL scale much easier than SQL databases (their main selling point).

We will cover different DB types in a future lecture.

## The CAP Theorem



# Consistency Types

## Strong consistency

- When data is changed, every client sees the new data immediately.

## Eventual consistency

- Some clients might see the new data with some delay
- ACID vs BASE

Certain DBs can be tuned to favor different consistency types.

Check out the [Cosmos DB consistency levels](#)

# Scaling for Reads

Nodes dedicated just for reading.

Still a single node for writing.

Data replication is usually asynchronous (i.e. leading to eventual consistency).

Clients must be aware of it - i.e. read from nodes dedicated for reading.

# Partitioning

Split the data into multiple subsets/groups/**partitions**.

Each item in a given partition has the same partition key.

Doesn't imply multiple databases/nodes.

Still **improves performance as queries are usually executed within one partition**.

Queries should contain the partition key otherwise the DB engine has to look through all partitions.

# Sharding

Similar to partitioning, but actually involves distributing the data across multiple nodes.

# Bonus - Types of Traffic

## North-South

- Refers to traffic from outside - i.e. from clients/external systems
- Requires more attention
  - Firewalls
  - NACLs
  - Authentication
  - Authorization
  - Rate Limiting
  - Schema verification

## East-West

- Refers to traffic within the system's network - i.e. across servers/services
- Usually more relaxed checks

# Summary

# Cloud Applications Architecture

...

Course 5 - Web Capabilities

# Topics

(Technologies that help us build and consume services on the web)

- HTTP
- Content
- Hosting
- CDNs
- Domains
- DNS

# **URI/URL/URN**

# URI/URL/URN

**Universal Resource Identifier:** used to identify a resource (doesn't have to be internet related).

E.g. *ISBN 1234*

**Universal Resource Locator:** a more specific URI that also describes how to get the resource over a given protocol.

E.g. *http://books.com/books/1234*

**Universal Resource Name:** a more specific and standardized URI.

E.g. *urn:isbn1234*

# URL

`https://www.caacourse.com:443/courses?status=published`

Protocol

Subdomain

Domain

Port

Path

(Query) Parameters

# HTTP

# HTTP Concepts

## Hypertext Transfer Protocol

Defines how messages should be structured.

Layer 7 (Application).

Read more from some well-known sources: [Mozilla](#), [Cloudflare](#)

Stateless protocol (requests are independent)

- As opposed to TCP which is stateful for example

Common actors: client (user-agent), server, proxies

# HTTP Versions

## HTTP/1.0

- Initial version, required a new TCP connection for each request

## HTTP/1.1

- Allowed TCP connection reuse
- Still widely used today

## HTTP/2

- Derived from Google's SPDY protocol
- Incremental changes over HTTP/1.1; most notable being Server Push

## HTTP/3

- Entirely different than previous versions. Works over UDP

# HTTP Methods

**Safe** methods (no change expected)

GET

HEAD

OPTIONS

These methods are also **idempotent**.

**Not safe** methods (change expected)

POST

PUT (**Idempotent**)

DELETE (**Idempotent**)

PATCH

**It's still our job to use these methods as intended and expected**

# HTTP Headers

Commonly used headers

- **Content-type/Accept:** *application/json, text/plain, \*/\**
- **Cache-Control:** *no-cache, max-age=<seconds>*
- **Authorization:** *Basic <base64 credentials>, Bearer <token>, etc*
- **Access-Control-Allow-Origin:** *https://caacourse.com* (needed for CORS)
- **Origin:** *https://caacourse.com*

Some headers are automatically handled by browsers (e.g. CORS).

A common practice is to prefix custom headers with **x-**, but it's not really necessary nor encouraged.

Watch [this](#) video for some great capabilities headers offer us.

# HTTP Header Fields

Can be grouped as

- **General:** Request URL, Request Method, Status Code, Remote Address
- **Request:** Cookies, Accept-xx, Content-Type, Content Length, Authorization, User-Agent
- **Response:** Server, Set-Cookie, Content-Type, Content-Length, Date

# Content

# Types of Content

Images

**Static**

Video

Hello!

User data

Text

**Dynamic**

Binary

Hello, *John*!

Applications (i.e. html, css, js)

# Content Storage

Common cloud services

- Amazon S3 (Simple Storage Service)
- Google Cloud Storage
- Azure Storage Account (different sub-services: Blob, File, Queue)

# Content Caching

Content can be cached at multiple levels

- Server
- CDN (Content Distribution Network)
- Browser

CDN and Browser caching is usually controlled through Header Cache-Control

Both static and dynamic content can be cached, but with different **TTL** (time to live).

Various techniques to refresh the content such as purging and cache-busting.

# Content Distribution

## Common issues

- Latency (for global distribution)
  - Solved by storing the content across multiple locations, closer to more users
- Paid content
  - Storage services usually support **signed requests/URLs**.

# Website Hosting

# Website Hosting

## “Static” Websites

I.e. no processing is required to serve the website (html, css, js files).

Cheap/Free hosting. Can leverage storage services (e.g. S3).

## Jamstack

(These also include react/angular/vue etc. apps)

## “Dynamic” Websites

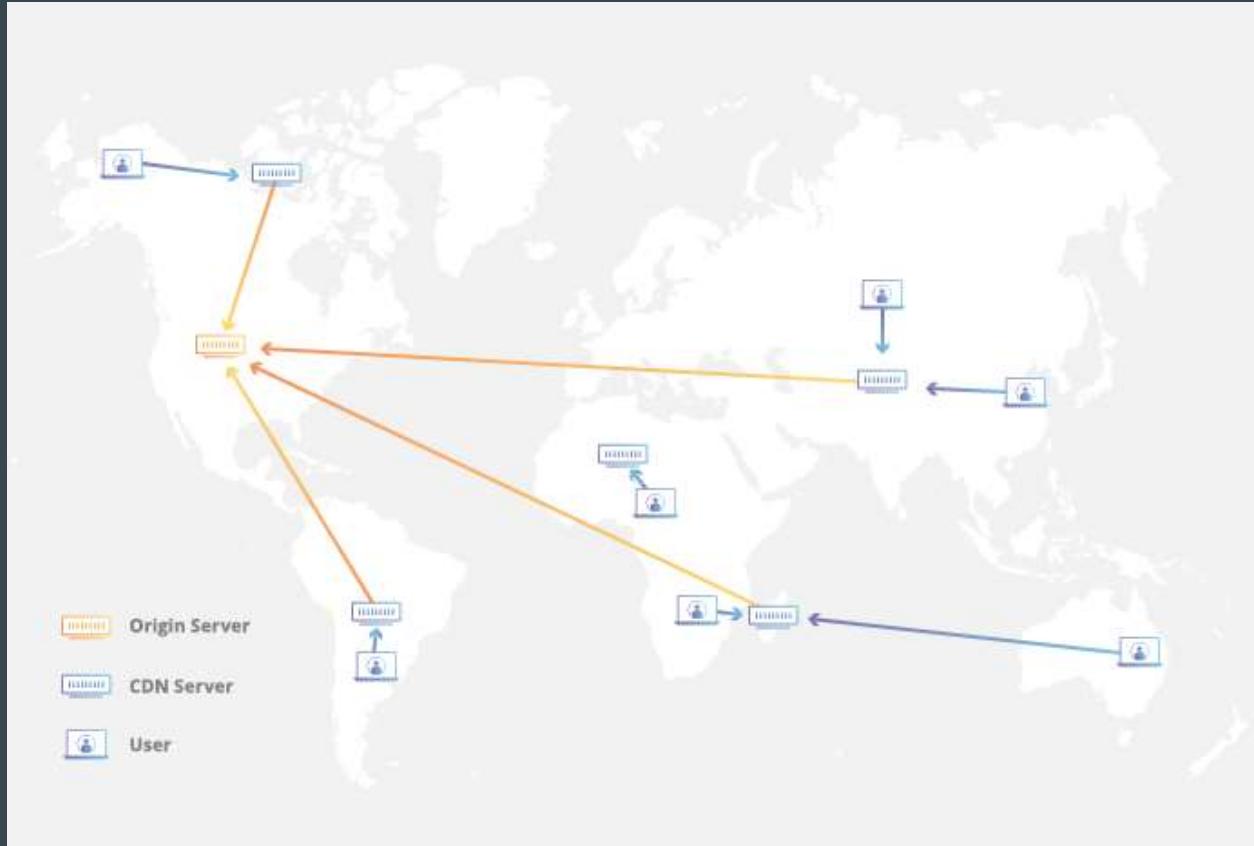
I.e. The server generates the website files when requested.

E.g. Wordpress sites.

Require a server with dedicated CPU.

# Content Delivery Networks (CDN)

# What is a CDN?



**Edge caching**

Source: [Cloudflare](#)

# Why CDNs?

## Improve load times

- Content is physically closer to the end-user

## Optimize costs

- For content heavy applications, traffic is one major cost driver

## Reduce pressure on critical resources

- Our system only loads the content into the CDN

## Improved security

- Most CDNs provide features such as DDoS protection

# CDN Examples

- [AWS CloudFront](#)
- [Google Cloud CDN](#)
- [Azure CDN](#)
- [Cloudflare CDN](#)
- [Akamai CDN](#)

Some providers also support edge computing. E.g. [Lambda@Edge](#)

# Domains

# Domains

Domains, subdomains, domain registrars

Relative paths, local development, virtual hosts.

Multiple environments (can be deployed differently): dev, stage, prod.

# Domain Name System (DNS)

# DNS

Translates **domains/hostnames** to **IP addresses**.

Multiple layers. Companies might even have internal DNS.

DNS Actors:

- DNS Recursor
- Root nameserver
- TLD nameserver
- Authoritative nameserver

# DNS Records

- **NS (Name Server)**
  - Indicates the DNS server handling DNS requests
- **A Record**
  - Points the domain to the server (actual IP address)
  - 172.217.17.238 -> google.com
- **CNAME (Canonical Name)**
  - Points an “alias” (sub)domain to the real (sub)domain
  - www.google.com -> google.com
- **MX (Mail Exchange)**
  - @ -> mxa.mailgun.org (these also usually have a priority, e.g. 10)
- **TXT**
  - General purpose. Usually used to verify ownership of the domain

# DNS Services

AWS Route53

Google Cloud DNS

Google DNS (8.8.8.8) (Public)

Azure DNS

Cloudflare DNS

Cloudflare 1.1.1.1 (Public)

# DNS TTL

Clients usually cache the association between a domain and its IP address.

The caching time is given by the DNS Record TTL.

Relevant especially when we change DNS records.

# Summary

# Cloud Applications Architecture

• • •

Course 6 - High Availability

# Highly Available (HA) Systems

# What Makes a System Highly Available?

- Hardware
- Software
- Data
- Network

# Why is Availability Important?

Business continuity

Loss of revenue, customers, lives

SLAs (certain SLAs might allow “uncounted” downtime if the service recovers within a certain time)

High Availability vs Continuous Availability

# Availability Formula

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime + Downtime}}$$

<https://uptime.is/>

Availability %	Downtime per year <sup>[note 1]</sup>	Downtime per month	Downtime per week	Downtime per day
90% ("one nine")	36.53 days	73.05 hours	16.80 hours	2.40 hours
95% ("one and a half nines")	18.26 days	36.53 hours	8.40 hours	1.20 hours
97%	10.96 days	21.92 hours	5.04 hours	43.20 minutes
98%	7.31 days	14.61 hours	3.36 hours	28.80 minutes
<b>99% ("two nines")</b>	<b>3.65 days</b>	<b>7.31 hours</b>	<b>1.68 hours</b>	<b>14.40 minutes</b>
99.5% ("two and a half nines")	1.83 days	3.65 hours	50.40 minutes	7.20 minutes
99.8%	17.53 hours	87.66 minutes	20.16 minutes	2.88 minutes
<b>99.9% ("three nines")</b>	<b>8.77 hours</b>	<b>43.83 minutes</b>	<b>10.08 minutes</b>	<b>1.44 minutes</b>
99.95% ("three and a half nines")	4.38 hours	21.92 minutes	5.04 minutes	43.20 seconds
<b>99.99% ("four nines")</b>	<b>52.60 minutes</b>	<b>4.38 minutes</b>	<b>1.01 minutes</b>	<b>8.64 seconds</b>
99.995% ("four and a half nines")	26.30 minutes	2.19 minutes	30.24 seconds	4.32 seconds
<b>99.999% ("five nines")</b>	<b>5.26 minutes</b>	<b>26.30 seconds</b>	<b>6.05 seconds</b>	<b>864.00 milliseconds</b>
<b>99.9999% ("six nines")</b>	<b>31.56 seconds</b>	<b>2.63 seconds</b>	<b>604.80 milliseconds</b>	<b>86.40 milliseconds</b>
<b>99.99999% ("seven nines")</b>	<b>3.16 seconds</b>	<b>262.98 milliseconds</b>	<b>60.48 milliseconds</b>	<b>8.64 milliseconds</b>
<b>99.999999% ("eight nines")</b>	<b>315.58 milliseconds</b>	<b>26.30 milliseconds</b>	<b>6.05 milliseconds</b>	<b>864.00 microseconds</b>
<b>99.9999999% ("nine nines")</b>	<b>31.56 milliseconds</b>	<b>2.63 milliseconds</b>	<b>604.80 microseconds</b>	<b>86.40 microseconds</b>

Data from [Wikipedia](#)

# Common Availability Tiers

95%

99%

99.9%

99.95%

99.99%

99.999%

# Availability Concerns - **Nature**



# Availability Concerns - Technical

## **Unplanned**

Usually due to human error

## **Planned** (maintenance)

You can define the maintenance window for certain services

Usually third-party APIs notify you

# Techniques to Achieve HA

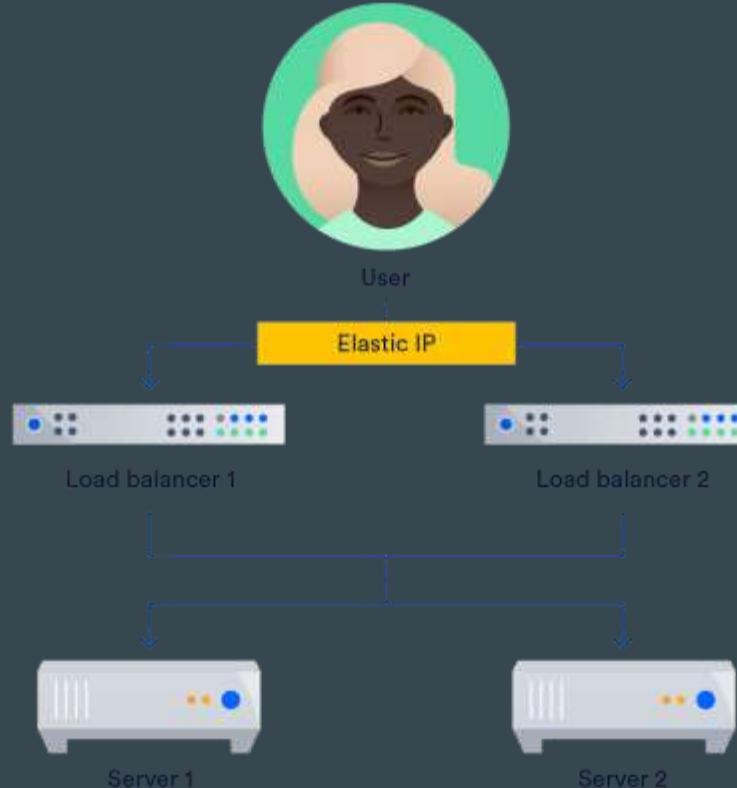
# Techniques

Infrastructure level

Application level

# Floating/Elastic IP

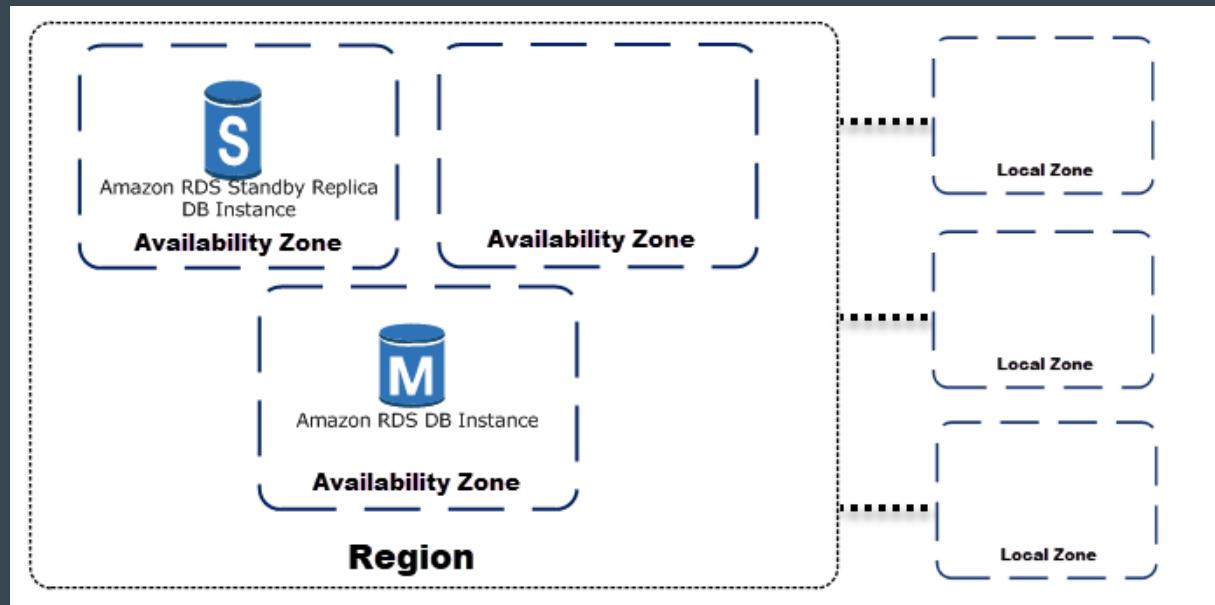
Eliminate Single Point of Failure



# Multi-AZ

Many services support it natively

Can be hard to maintain otherwise



# Multi-AZ

How to achieve it?

- Cluster-aware routing (load balancers, DNS)
  - Health checks/probes - natively supported by many services
- Data replication
  - Sync/Async
  - Available solutions
- Packet mirroring (Traffic duplication)

# Multi-Region

**Regions are usually entirely different clouds.**

Netflix users didn't even notice an entire AWS region went offline

# Minimizing Impact Radius

(Decoupled) Microservices

# Proper Processes

i.e. try to avoid human errors

Code reviews (4-eye principle)

CI/CD



# Proper Monitoring

React quickly

What to monitor:

- Database
- Website
- Virtual network
- Storage
- VM

# “Embrace the Chaos”

Netflix Chaos Monkey

<https://principlesofchaos.org>

# Resiliency

# Resiliency

Capability of a system to remain functional/useful even if parts of it become unavailable.

No matter how perfect a system is, failure is certain.

Resilient system is:

- Adaptive
- Self healing
- Predictable

Requires

- Investment
- Automation
- Monitoring
- Simplicity

# Disaster Recovery (DR)

# Recovery Time Objective (**RTO**)

How long it takes to bring the system back.

Highly dependant on the **DR Strategy**.

Lower RTO usually means (considerably) increased cost.

# Recovery Point Objective (**RPO**)

How much data was lost.

I.e. How much time has passed since the last backup.

Usually easier to improve:

- More frequent backups
- Leverage incremental backups to reduce costs

# RTO & RPO



# DR Strategies

## Backup and Restore

- In case of disaster, restart/recreate everything based on the latest backup

Worse RTO,  
cheaper

## Pilot Light

- Have the critical components prepared
- E.g. have a database replica ready for DR (but no compute)



Better/lower RTO,  
(much) more  
expensive

## Warm Standby

- Full system replica ready, reduced size (e.g. smaller VMs)

## Hot Site

- Exact replica ready

# Summary

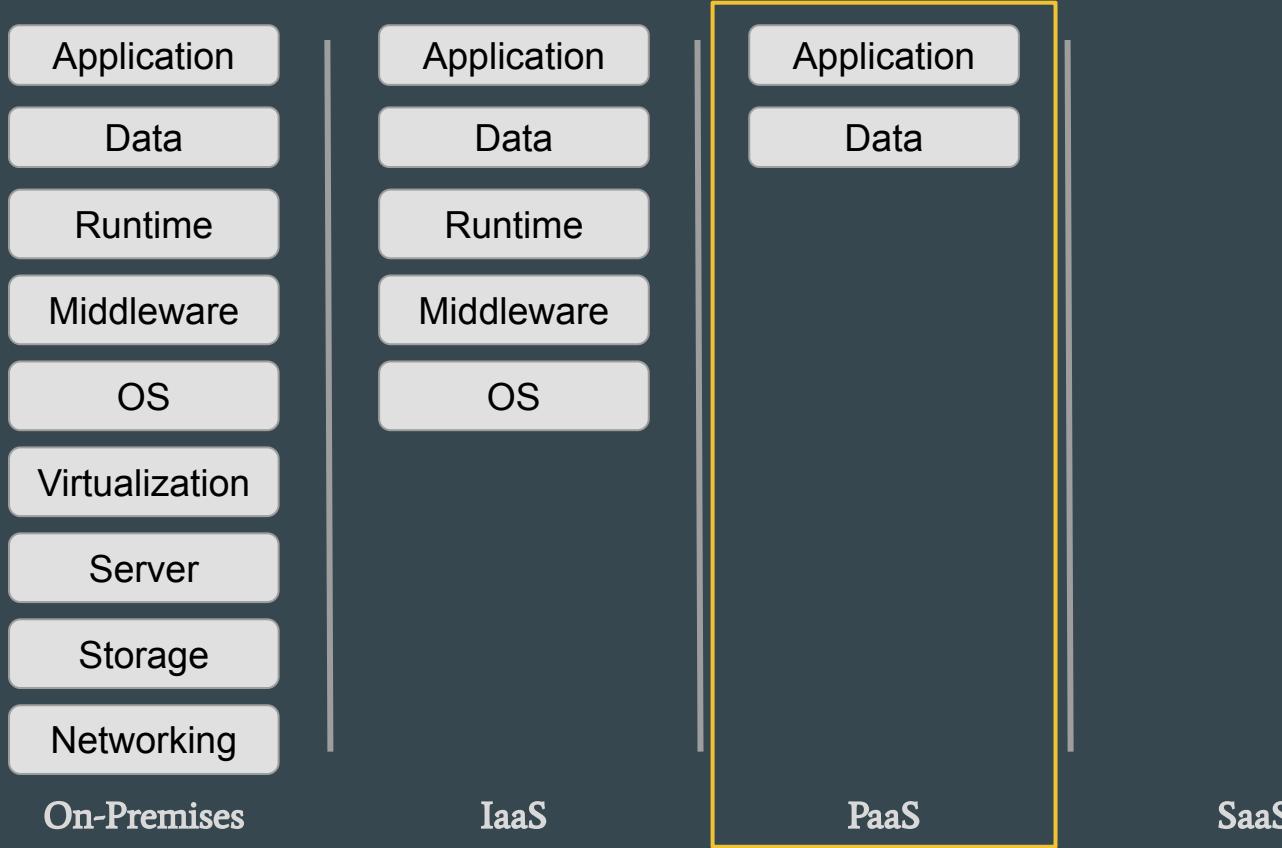
# Cloud Applications Architecture

• • •

Course 8 - PaaS & Managed Services

# What is PaaS?

# IaaS, PaaS, SaaS - What We Manage



# IaaS, PaaS, SaaS - What We Manage

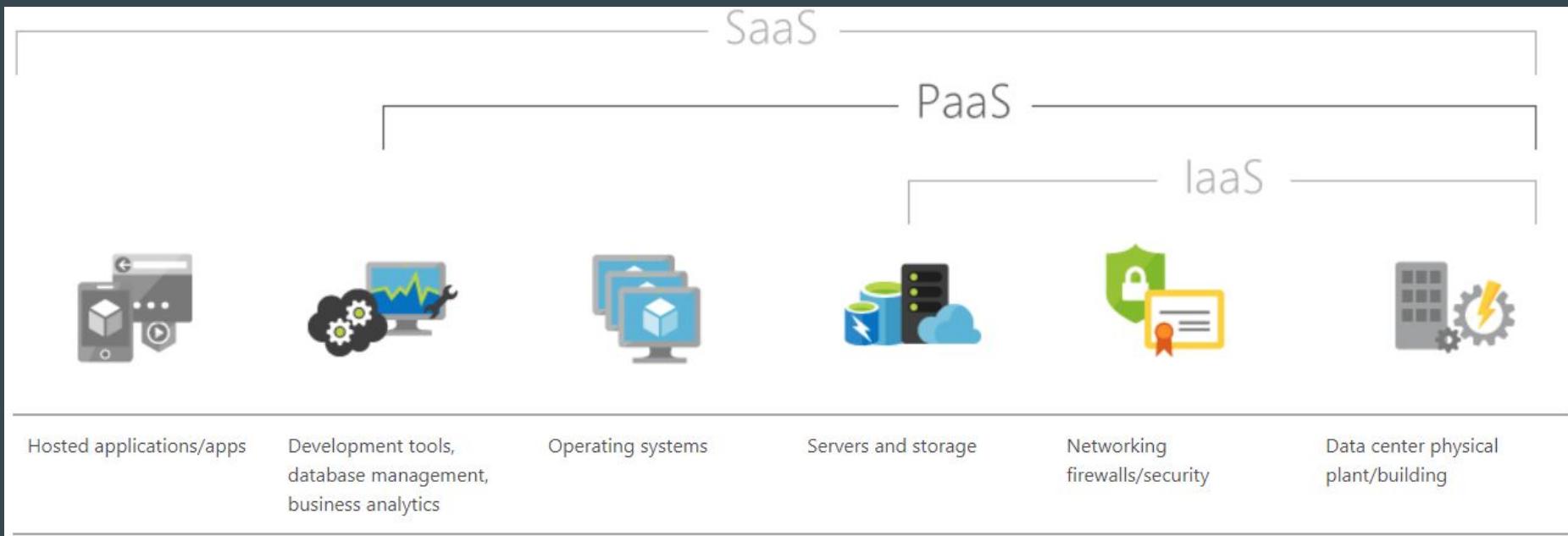


Image from [Azure](#)

# What is PaaS?

- Based on IaaS
- Services for common challenges/requirements (e.g. DB)
- Managed services
- **Wide** range of services
  - Compute
  - Data storage
  - Monitoring
  - User management
  - Encryption
  - Combination of some or all

# Pros of PaaS

- Less to worry about => focus on business and actual features
- Dedicated/specialized teams manage them
- Different billing
- Can be cheaper
  - Billing might be more favorable
  - No administration effort/cost

# Cons of PaaS

- Opinionated, you have to work with what the provider gives you
- Can be more costly to run
- Vendor lock-in

# When to Use PaaS

- Fast prototyping/product validation
- When the upsides outweigh the downsides
- When working with a small team
- When you understand and accept the lock-in to a decent degree

# When Not to Use PaaS

- When building “exotic” products
  - E.g. banking systems
- When building ultra-high performance systems
  - Especially if the latency is decisive
  - E.g. high-frequency trading
- When control/security is a must
  - Most PaaS will provide better security than we can achieve, however we cannot completely audit it.
- When working with “special” licenses
  - Some require dedicated hardware

# Examples - General/Compute



Google App Engine



Azure App Service

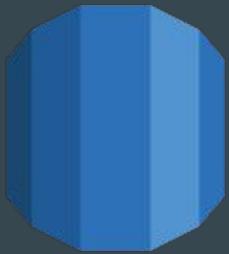


AWS Elasticbeanstalk



CLOUD FOUNDRY  
Cloud Foundry

# Examples - Databases



AWS RDS



Azure CosmosDB



MongoDB Atlas

# Environment Parity

*“It works on my machine”*

# Environment Parity

One of the 12 factors.

Local, dev, stage, ..., prod should be as similar as possible.

More and more relevant as we start using more services.

Achieved by using the right tools.

# Containers

“Lightweight VMs”

Unit containing the application and all its dependencies.

Useful for:

- Creating identical/similar environments
- Quickly provisioning dependencies locally (DBs, caches, queues etc)
- Keeping your computer clean

Docker's explanation

# Docker

One container implementation.

Leverages **Linux** tools for creating the “lightweight VMs”.

- **Namespaces**
  - Makes the container look like a complete VM
  - Isolates it from other containers/processes
- **Control Groups**
  - Limits resources
- **UnionFS**
  - Keep the size small
  - Copy on write

[Docker docs](#)

# Docker - Terminology

## Images

- Similar to OOP classes that make heavy use of inheritance
- Stored locally and/or on container registries (e.g. [Docker Hub](#), [GCR](#), [ECR](#), etc.)

## Containers

- Processes running based on given images. Similar to OOP objects.
- Can be created, started, stopped, removed.
- Allows us to execute commands within it (similar to a VM).

## Services

- Manage, scale and load balance containers across multiple hosts

# Docker - Terminology

## Volumes

- Needed since containers are ephemeral and isolated
- A way to persist data separate of the container's lifecycle

## Networks

- Enable communication between multiple containers.
- Allow us to control network isolation separately

# Demo

Useful tutorials:

- The official getting started guide from Docker

# Docker Compose

Useful for running and managing multiple containers (on the same host)

Has a more declarative approach - [\*docker-compose.yml\*](#)

# Kubernetes

Container orchestrator

Works with Docker and other container engines ([link](#))

Useful for:

- Running (heterogeneous) containers in a cluster (and handling communication)
- Optimizing resource utilization
- Large/variable workloads
- Rolling updates

Offers what cloud providers already offers, but in a cloud/provider-agnostic way.

# Kubernetes - Terminology

## Cluster

- The underlying infrastructure (can be any IaaS and on-premises)

## Nodes

- VMs available to kubernetes (i.e. having kubelet installed)

## Pods

- Smallest deployment unit.
- Runs one or more (related) containers
- Multiple pods can be deployed on a node
- Pods are usually managed by other components
  - Deployment, StatefulSet, DaemonSet

# Kubernetes - Terminology

## **Services**

- Pods cannot be reached by default
- Services expose pods to other pods or to the outside world
- Multiple types: ClusterIP, NodePort, LoadBalancer (this will create the corresponding service offered by the cloud provider)

## **PersistentVolumes**

- Represent a physical storage volume (e.g. EBS on AWS)
- Consumed by Pods through PersistentVolumeClaims

# Relevant Resources

## Tools

- Helm ~ package manager for kubernetes
- Skaffold ~ streamlines local development (e.g. hot-reload) and deployment

## Tutorials

- The official kubernetes tutorials
  - E.g. Deploy Wordpress

# Cloud Applications Architecture

...

Course 8 - Databases

# Introduction

What is a database?

What is a DBMS?

# Types of Workloads

# Online Transactional Processing (OLTP)

The most common workload

Examples: banking, ERP, booking, billing

Characteristics:

- **Transactional behavior**
  - If 2 people try to book the same place, only one succeeds.
- **Known access patterns**
  - The use cases of the system are well-defined
- **Write-heavy**
- **Known data schema**
- **Short operations (~milliseconds)**
- **Back-up is highly important**

# Online Analytical Processing (OLAP)

Used to be relevant only for large businesses.

Based on data warehouses (central data repository).

Examples: analyze sales in a certain period/location.

Characteristics:

- Access patterns/use cases not known entirely (Ad-hoc)
- Read heavy
- Large data volumes
- Possibly long operation (minutes, hours)
- Data consistency & back-up not crucial

# Decision Support System (DSS)

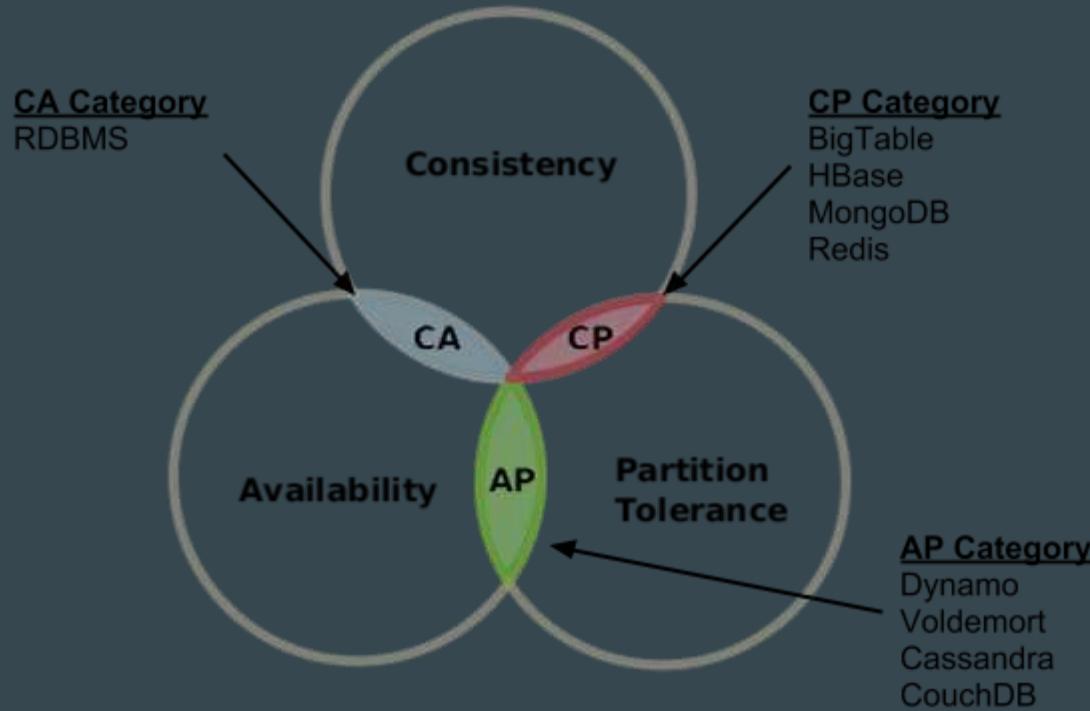
Similar characteristics to OLAP

- Queries might be known since the beginning
- Data spans longer timeframes (more historical data)

# Types of Databases

# Recall

## CAP Theorem



## ACID vs BASE

# Relational

Based on **SQL** (Structured Query Language)

- Declarative

## RDBMS

Optimized for storage (reduced data duplication - normalization).

Examples:

- Open source: PostgreSQL (`postgres`), MySQL, MariaDB, SQLite, H2
- Commercial: SQL Server, Oracle, DB2, Cloud Spanner, HANA, Aurora
- Commercial extensions: Percona (for MySQL), Citus (for Postgres clusters)

# NoSQL

Started as a twitter tag

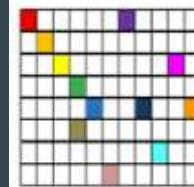
Stands for Not Only SQL

Covers a wide range of database types

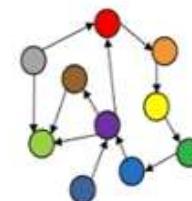
Optimized for compute & scalability

All are based on partition key

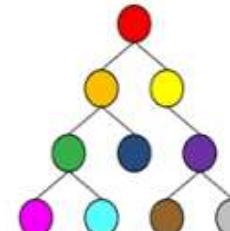
Column-Family



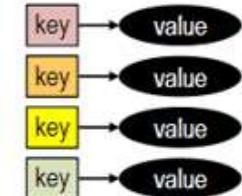
Graph



Document



Key-Value



# Key-Value

Data is always accessed based on a key.

DBs can be scaled infinitely (since there is no relation between items).

**Ultra low and consistent latency.** (sub-millisecond)

- Especially for some in-memory databases

**High and consistent throughput.**

Examples:

- Caching systems: Redis, Memcached
- Amazon Dynamo

# Column

There are 2 (somehow similar) types:

- Column stores

Stores data tables by column rather than by row

Serializes all of the values of a column together, similar to an index organization for row oriented dbs

Reduces the amount of data read from disk by compressing the similar columnar data and by reading only the necessary data

Examples

- HBase, Cassandra
- MariaDB ColumnStore

# Column

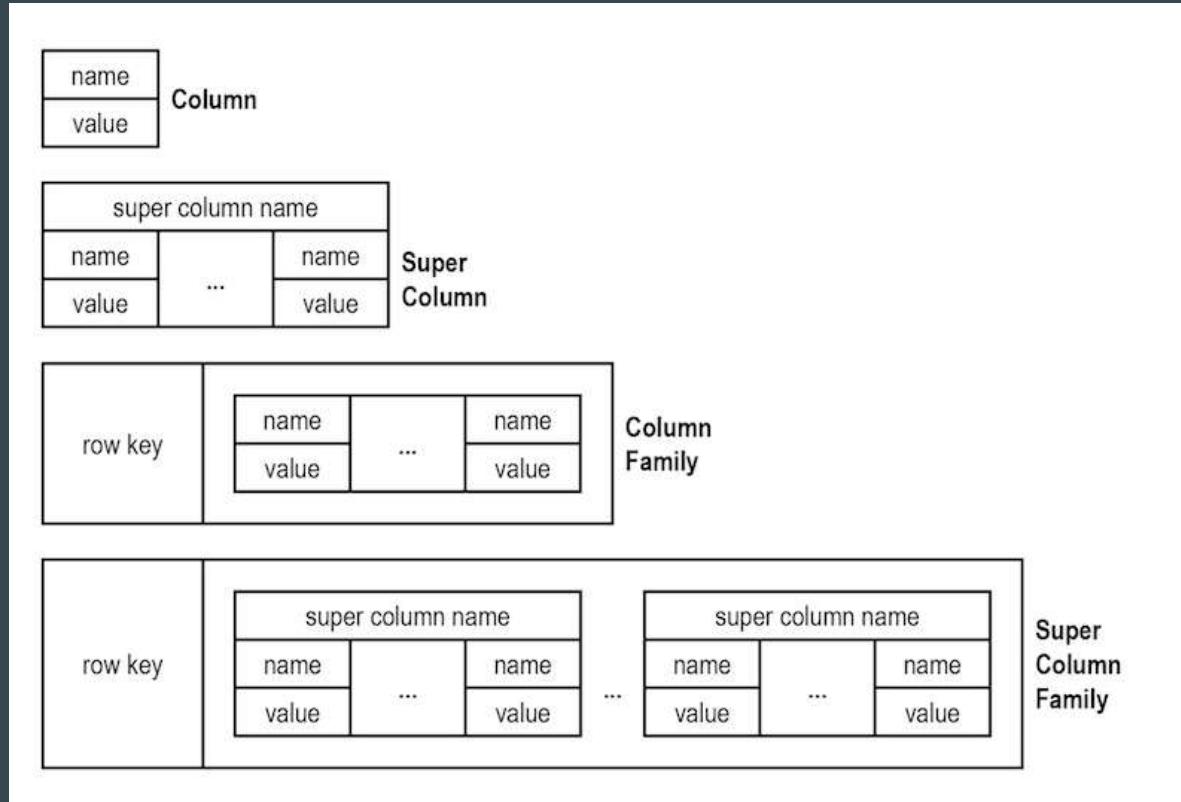
- Wide column databases

Has an architecture that uses persistent, sparse matrix, multi-dimensional mapping (row-value, column-value, and timestamp) in a tabular format meant for massive scalability (over and above the petabyte scale)

Examples:

- BigTable, Cassandra, Accumulo

# Column



# Document

Focus on storage and access methods optimized for documents

Data modeled as collections of documents containing key-value pairs.

Values can be scalar values but also nested documents or lists as well as

Attribute names are dynamically defined for each document (no schema enforcement)

## Examples

- MongoDB
- CouchDB
- Elasticsearch !

# Graph

Use topographical data models to store data

Connect specific data points called nodes

Create relationships called edges in the form of graphs

- contains nodes and relationships
- nodes contain properties (key-value pairs)
- relationships are named and directed, and always have a start and end node
- relationships can also contain properties

Examples

- Neo4J
- OrientDB

# Time-Series

Optimized for time-stamped data

- The data that arrives is almost always recorded as a new entry
- The data typically arrives in time order
- Time is a primary axis (time-intervals can be either regular or irregular)

Time-series data workloads are generally “append-only.”

Data could be measurements or events tracked, monitored and aggregated over time (server metrics, network data, sensor data, events, clicks, etc)

## Examples

- InfluxDB
- TimescaleDB

# Search

Dedicated to the search of data content

Use indexes to categorize the similar characteristics among data and facilitate search capability

Optimized for dealing with data that may be long, semistructured, or unstructured

Offer specialized methods such as full-text search, complex search expressions, and ranking of search results

## Examples

- Elasticsearch
- Solr

# Ledger

Key characteristics of a ledger database:

- **Immutable**: the past doesn't change. Each write to the database is appended to the past.
- **Transparent**: there is access to the past. The log information can be retrieved.
- **Verifiable**: offers a way to validate the complete history of changes.

The database contains the log and tables which are a view into the log's data

It's not enough to have access to historical data, you must be able to verify the authenticity of that history (cryptographically verifiable transaction log owned by a central authority).

# Understand Security

- Administrative user or authentication is not enabled by default.
- It has a very weak password storage
- Client communicates with server via plaintext(MongoDB)
- Cannot use external encryption tools like LDAP, Kerberos etc
- Lack of encryption support for the data files
- Weak authentication both between client and the servers
- Vulnerability to SQL injection
- Denial of service attacks

## Examples

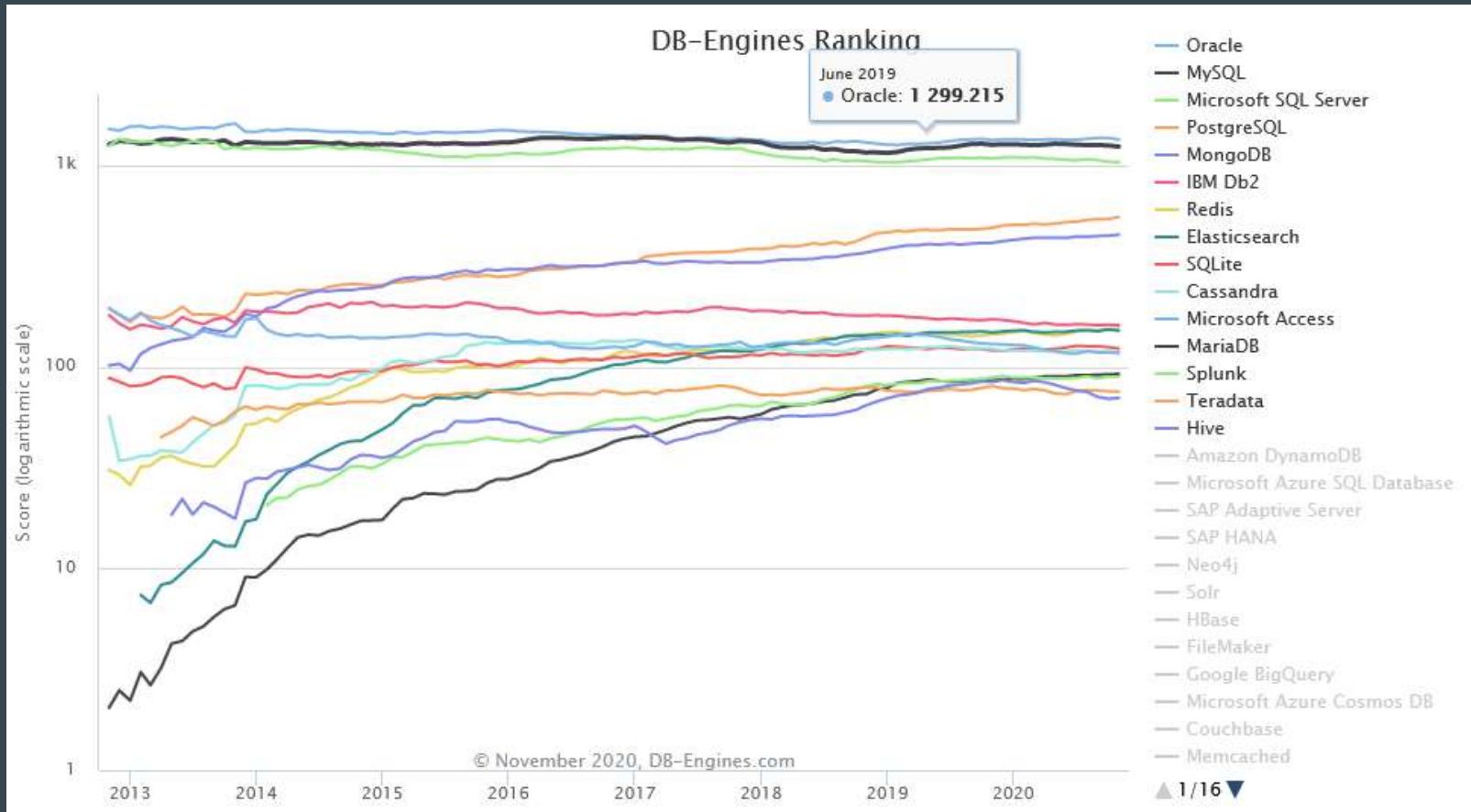
Data breach1

Data breach 2

360 systems in ranking, November 2020

Rank			DBMS	Database Model	Score		
Nov 2020	Oct 2020	Nov 2019			Nov 2020	Oct 2020	Nov 2019
1.	1.	1.	Oracle 	Relational, Multi-model 	1345.00	-23.77	+8.93
2.	2.	2.	MySQL 	Relational, Multi-model 	1241.64	-14.74	-24.64
3.	3.	3.	Microsoft SQL Server 	Relational, Multi-model 	1037.64	-5.48	-44.27
4.	4.	4.	PostgreSQL 	Relational, Multi-model 	555.06	+12.66	+63.99
5.	5.	5.	MongoDB 	Document, Multi-model 	453.83	+5.81	+40.64
6.	6.	6.	IBM Db2 	Relational, Multi-model 	161.62	-0.28	-10.98
7.	↑ 8.	↑ 8.	Redis 	Key-value, Multi-model 	155.42	+2.14	+10.18
8.	↓ 7.	↓ 7.	Elasticsearch 	Search engine, Multi-model 	151.55	-2.29	+3.15
9.	9.	↑ 11.	SQLite 	Relational	123.31	-2.11	+2.29
10.	10.	10.	Cassandra 	Wide column	118.75	-0.35	-4.47
11.	11.	↓ 9.	Microsoft Access	Relational	117.23	-1.02	-12.84
12.	12.	↑ 13.	MariaDB 	Relational, Multi-model 	92.29	+0.52	+6.72
13.	13.	↓ 12.	Splunk	Search engine	89.71	+0.30	+0.64
14.	14.	↑ 15.	Teradata 	Relational, Multi-model 	75.60	-0.19	-4.75
15.	15.	↓ 14.	Hive	Relational	70.26	+0.71	-13.96
16.	16.	16.	Amazon DynamoDB 	Multi-model 	68.89	+0.48	+7.52

Source: <https://db-engines.com/en/ranking>



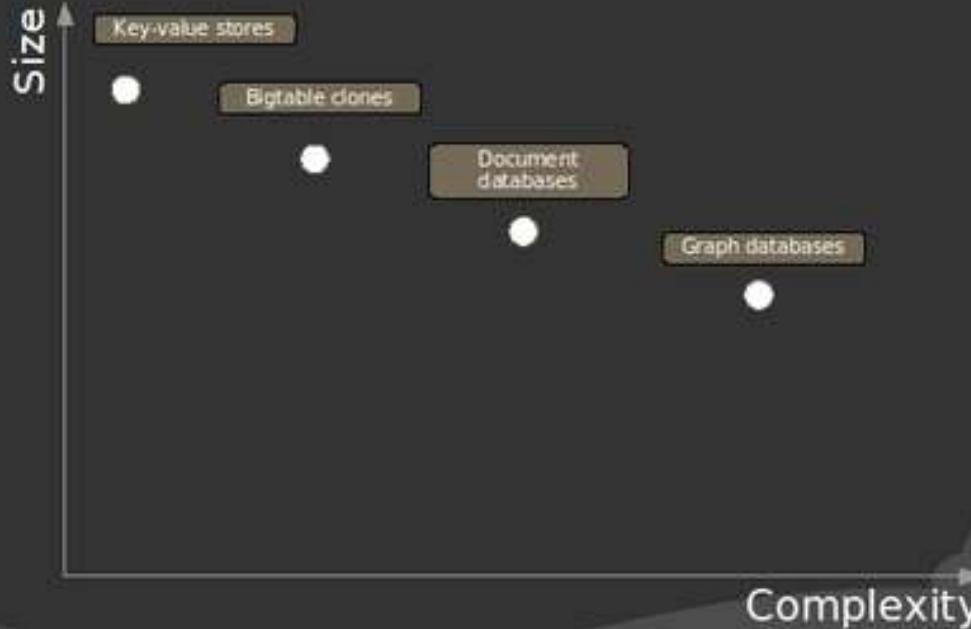
Source: <https://db-engines.com/en/ranking>

# Choosing the Right Database(s)

# Considerations

1. The purpose of the system (e.g. is it a social network?)
2. Are most access patterns known?
3. Performance
4. Volume
5. Does a user work mostly with his/her data? (as opposed to accessing the same data as everyone else - e.g. a leaderboard)
6. Scale/intended reach (local, regional, global)
7. Does one model fit all cases?

# NOSQL data models



# Managed Databases

# Migrating to Cloud

## Lift & Shift

Each cloud provider has its own tools to facilitate migration. Examples:

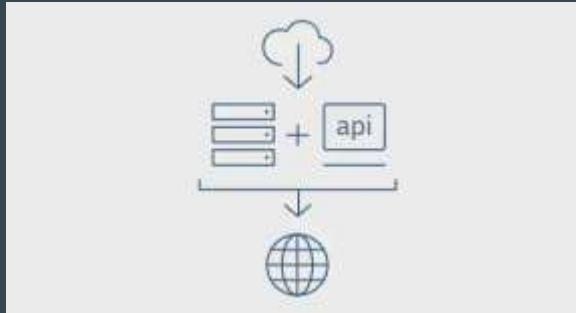
- AWS Database Migration Service
- Azure Database Migration Service

Usually support both homogeneous and heterogeneous source/target DBs

Disaster recovery strategies/services can also be used

# Advantages

Easy access



Scalable



Disaster safety



# Must Watch

[Martin Fowler's Intro to NoSQL](#)

AWS re:Invent talks:

- [DAT301](#)
- [DAT403-R1](#)
- [DAT205-R1](#)

# Cloud Applications Architecture

...

Course 9 - Infrastructure Security

# Security

# Why is Security Important?

More data (value) than ever (higher stakes)

- Data is the new oil

We do more (especially sensitive) things online.

Stricter regulations.

- More concerning fines

# CIA Triad

## Confidentiality

- only the authorized entities (people and/or systems) have access to the data

## Integrity

- only the authorized entities can modify the data through well-defined procedures

## Availability

- there is no point in having a well-guarded system if no one can use it.
- Also, earthquakes tend to overrule highly sophisticated security measures



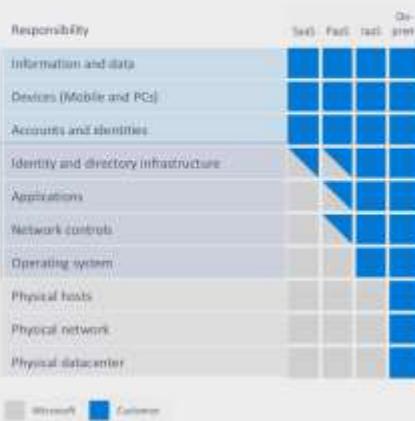
# Standards

Many cloud services are certified making your product also compliant

- [CSA](#) (Cloud Security Alliance) - Best practices for cloud security
- [ISO 27001](#) - Information Security Management
- [ISO 27017](#) - 27001 in the context of cloud
- [ISO 27018](#) - Personal data protection in cloud
- [PCI DSS](#) - Card payments security
- [HIPAA](#) - Health information protection

# Shared Responsibility Model

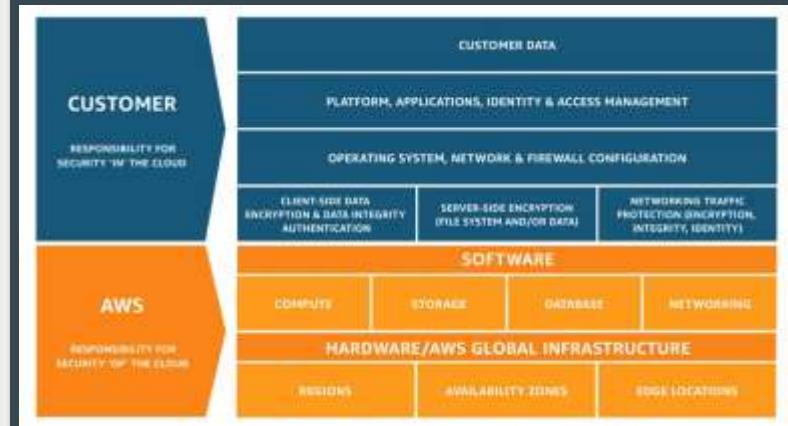
## Shared responsibility model



RESPONSIBILITY ALWAYS RETAINED BY CUSTOMER

RESPONSIBILITY VARIES BY SERVICE TYPE

RESPONSIBILITY TRANSFERS TO CLOUD PROVIDER



AWS Model

Azure Model

# Infrastructure Security

(as opposed to application security)

Concerned with:

- Network architecture
- Cloud users (as opposed to application users)
- Application/system permissions
- Resource access permissions

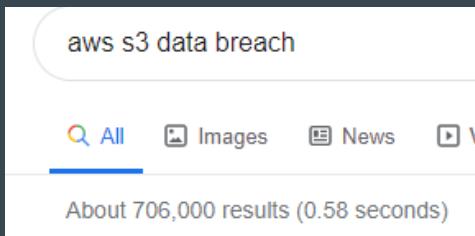
An entirely different paradigm compared to traditional approaches, main difference being the **user** concept.

# Infrastructure Security - Common Issues

## Unauthorized access

- Mine bitcoin while someone else is paying
- My AWS account was hacked and I have a \$50,000 bill

## Leaky buckets - examples



# Secure by Design

- Not something you add later
  - DevSecOps approach
- Least privilege principle
  - Allow users and applications to perform only what they have to and no more
  - Avoid using root/admin accounts
- Reduced attack surface
  - Keep the number of publicly available resources as low as possible
  - Remove redundant users and policies (e.g. after an employee leaves)
- Multi-factor authentication
  - Most (if not all) providers support this, but it has to be enabled
- Zero trust architecture [Azure][Cloudflare]
  - Basically add security at each layer (not only at the edge)

# Secure by Design

Cloud enables us to make security part of the architecture.

- Networks and firewalls/security groups are resources just like any other.
- Users, roles and services accounts are also resources.
- Fine-grained authorization controls/policies are enforced immediately and are also resources in most cases.
- Whitelist approach - e.g. by default, a new user cannot do anything
- Most services behave similar to users
  - If a service must use another service, explicit permission must given.
- In cloud, most requests are HTTP based - each request is evaluated

# User Access and Permissions

# IAM

Most providers offer an IAM (Identity and Access Management) service:

- [AWS IAM](#)
- [Google Cloud IAM](#)
- [Azure Active Directory](#)

This service facilitates authentication and authorization to the cloud infrastructure

# Users

Most providers allow multiple users within one account.

These are usually developers or other stakeholders (DBAs, consultants, auditors).

Some providers also offer **groups**.

- Common examples

<input type="checkbox"/>	DE	developers
<input type="checkbox"/>	ST	stakeholders
<input type="checkbox"/>	RE	readers

# User Permissions

Managed through roles and/or policies

- Might also be assigned at the group level
- A user might be part of multiple groups

Least privilege principle

- Operations that require elevated privileges should leverage just in time privileges
  - [AWS enables users to assume certain roles](#)
  - [Azure AD Privileged Identity Management](#)

Can (should) vary based on environment

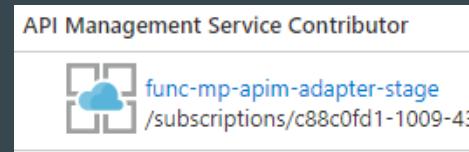
- E.g. a developer might have access to the database service in the development environment, but should not have access in the production environment

# Application Permissions

(Enable/allow a service to call another service)

Highly dependant on the provider. E.g.:

- In Azure, each service has its own roles (e.g. API Management Contributor) to which applications can be added.
- In Google Cloud, applications use service accounts through keys (one service account can have multiple keys).



# Logging & Monitoring

Since all actions in cloud are HTTP calls performed by a certain identity, they can be monitored - e.g. [AWS CloudTrail](#)

- Send notifications on certain events (e.g. VM creation)
- Help with audits
- Detect anomalies

# Confidentiality

- Enforce and democratize **encryption**
  - Facilitated by key management services: [Google CKM](#), [AWS KMS](#), [Azure Key Vault](#)
  - At least for sensitive and/or PII data
- Social engineering is still a risk nowadays
  - MFA and short lived access tokens help
- Most managed database services will offer encryption at rest by default.

# Integrity

Highly relevant especially in the context of distributed systems which are most likely eventual consistent.

# Availability

## Attacks

- DDoS: Krebs (620 Gbps), Mirai, Dyn, Project Shield (saved Krebs), AWS Shield (02.2020, 2.3 Tbps), Github (1.3 Tbps, memcache) -

Plan with availability in mind:

- Use dedicated services (that also make high availability easier to achieve)
- Leverage protection services/offerings (several network services from main providers offer DDoS protection by default) - e.g. Azure CDN.
- Attacks from today are just the normal traffic from tomorrow.

# Further Readings

[CNCF Security Whitepaper](#) (focuses on containers)

[AWS Well-Architected Framework - Security Chapter](#)

[Google Architecture Framework - Security, Privacy, and Compliance](#)

[Azure Well-Architected Framework - Security Chapter](#)

# Cloud Applications Architecture

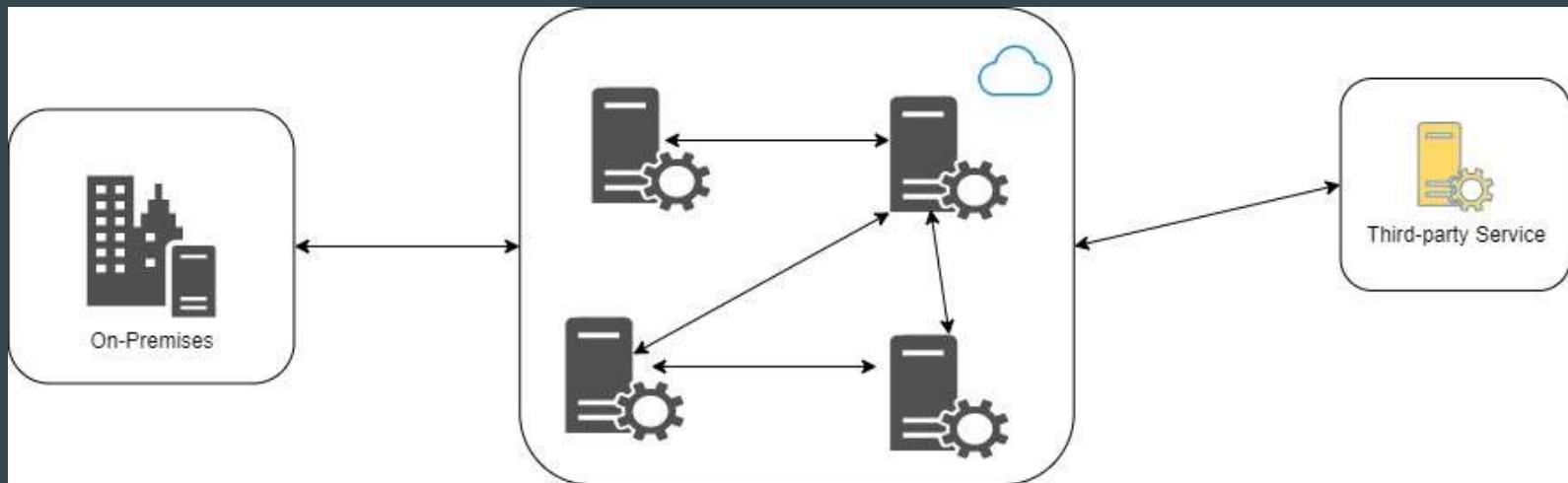
• • •

Course 10 - Integration Services

# Integration

Integrate 2 or more systems.

Can be microservices, 3rd party APIs, or on-premises systems



# Coupling

Degree of dependency between entities. I.e. how much entities must know about each other in order to function.

Strive for loose coupling (decoupled services/code).

## Benefits

- Better maintenance and development/extensibility
- Higher cohesion
- More efficient work split (even among different teams)

## Challenges

- Require additional tools/services
- Harder to plan, monitor and debug

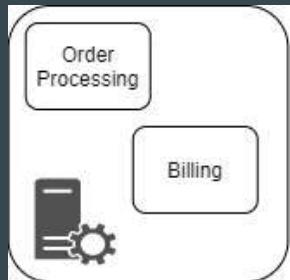
# Decoupling Example - Code

```
function processTransaction(amount: number) {  
    // the actual processing  
  
    const notificationService = new EmailNotificationService();  
    notificationService.send(`Sent ${amount} successfully`);  
}
```

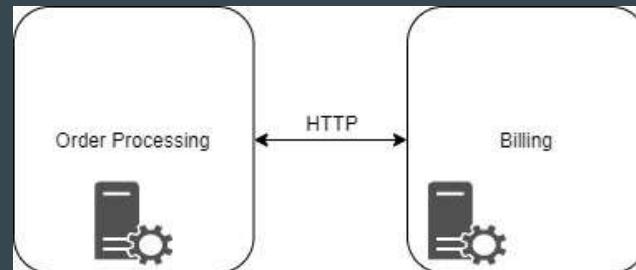
```
function processTransaction(amount: number, notificationService: NotificationService) {  
    // the actual processing  
  
    notificationService.send(`Sent ${amount} successfully`);  
}
```

Our function is no longer responsible for managing the dependencies. We just program for a certain interface/contract.

# Decoupling Example - Systems

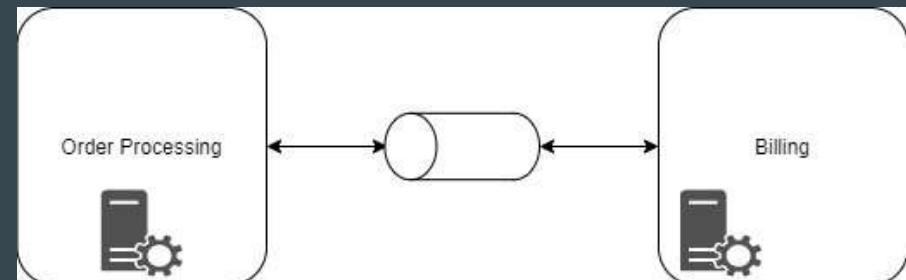


Everything built and deployed together

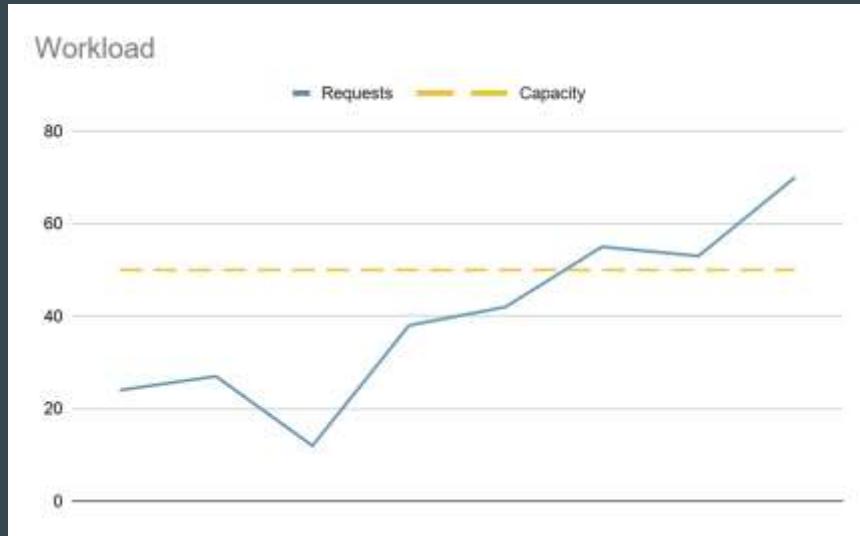


Independently built and deployed.  
Order service still must know how to  
communicate with the billing service

Independently built and deployed.  
Order service is no longer concerned  
with how the message reaches the  
billing service.



# Unpredictable Workloads

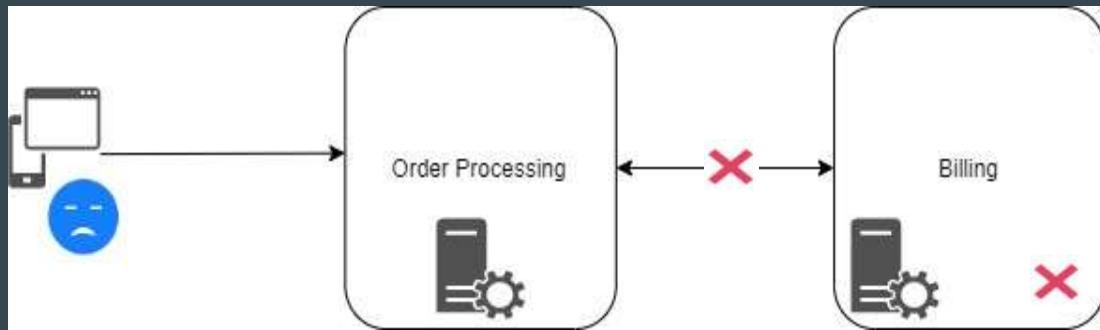


Instead of handling directly all the incoming requests, have a buffer in front.

What happens when the demand exceeds the capacity?



# Temporary Issues



- Temporary connection issues
- Billing service is down (server issues, planned maintenance)
- Rate limiting/throttling

The client has to perform the action again (e.g. fill in a long form)

# New Features

Considerations when delivering new features:

- Development (ripple effects/changes)
- Testing (how much to test to cover the new feature?)
- Deployment (speed, green/blue, canary)
- Minimizing the impact radius (in case something goes wrong)

# (A)Synchronicity

From the user's perspective, it's about when the response is received

- Synchronous communication - immediate response
- Async communication - response is sent separately (e.g. notifications)

Synchronous communication works when everything works. In practice, things tend to break.

A good portion of use cases can be handled asynchronously

There are ways to overcome async communication downsides: e.g. server push, polling.

# Flow Control

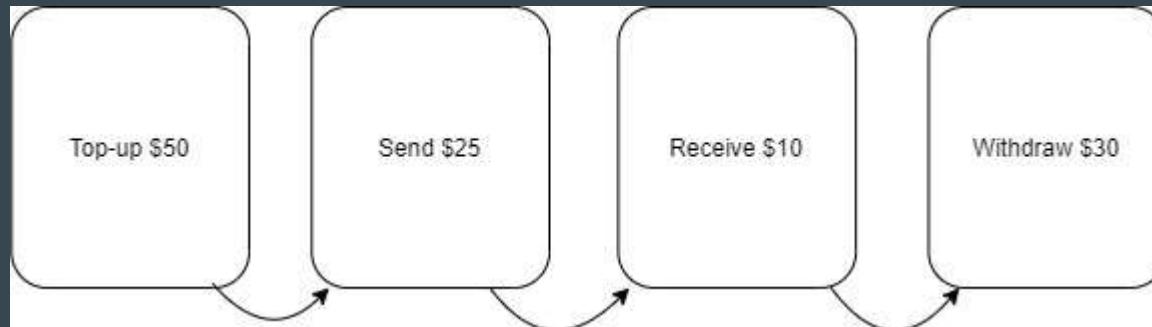
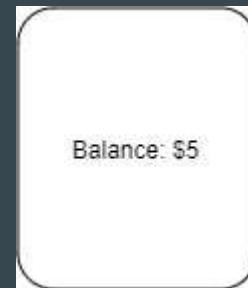
There might be cases in which requests must be processed in sequence.

We might want to temporary stop processing requests.

Certain requests might have higher priority.

# Persistence of Change

We are used to persisting the state.

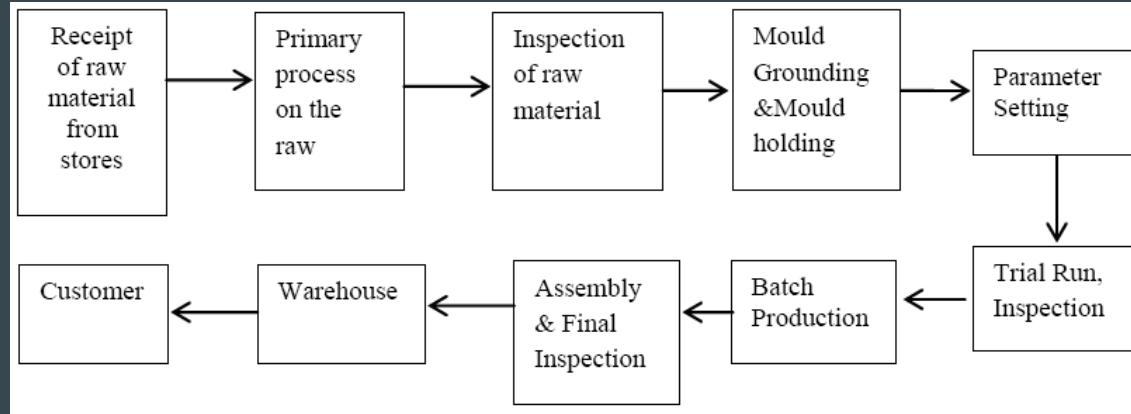


We can reconstruct the state.

We can analyze the history.

A.k.a. Event-sourcing

# Workflow Orchestration



Combine different services to achieve one result.

Maintain state.

Support different triggers (schedule, on a certain action, under certain conditions)

# Contract Definition

Separating the interface from the implementation might be beneficial:

- Easier refactoring/reworking/changing the architecture (e.g. transition from monolith to microservices)
- Other services are concerned only with the contract.
- Easier to achieve a common/unified interface.
- More in course 13 (API Design)

# Common Integration Services

# Common Integration Services

Queues

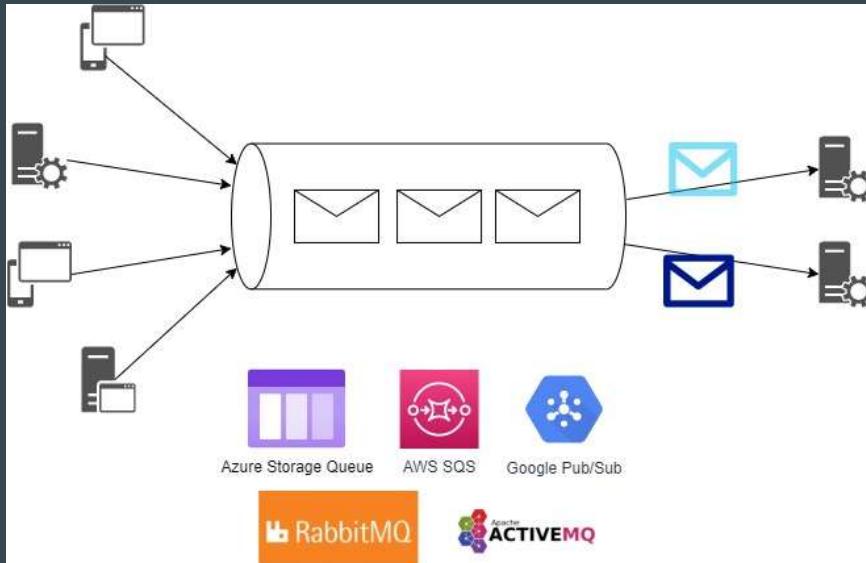
Publish/Subscribe Services

Streaming Services

API Gateways

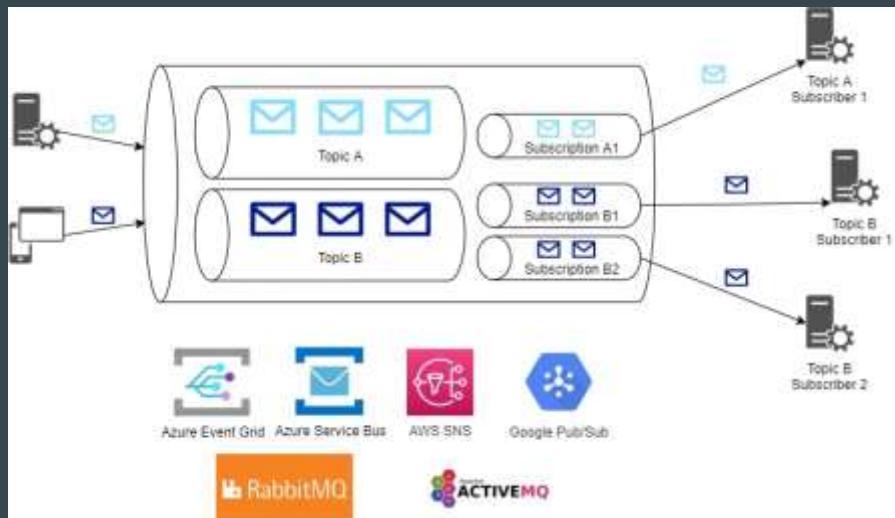
Workflow Services

# Queues



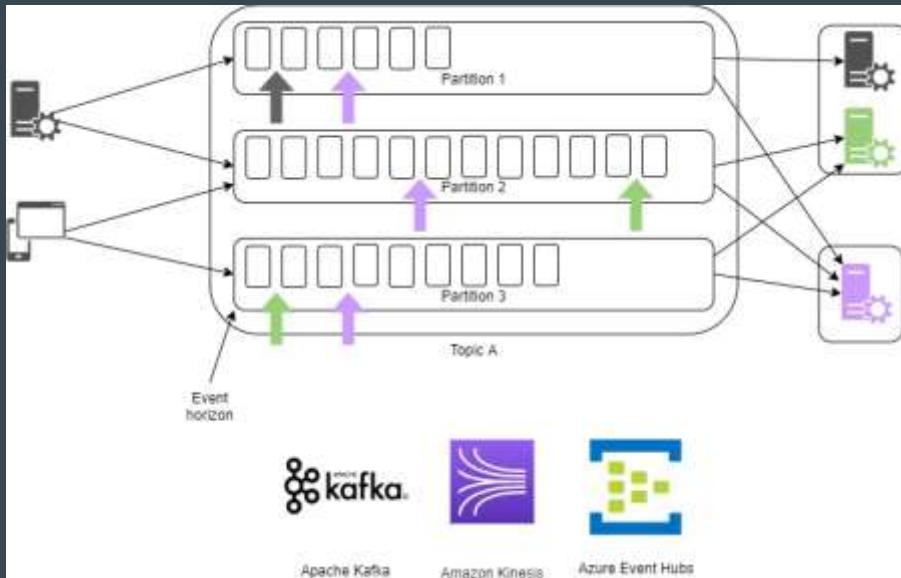
- Producers send messages to the queue
- Brokers manage the messages
- Consumers poll the queue (**pull**)
- Any consumer can receive any message(s) (batch)
- Once processed, the consumer sends the acknowledgement (usually deletes the message)
- Messages might be out of order.
- Usually **at least once** delivery.
- Queues help smoothing out the traffic and avoiding data loss
- Unprocessed message usually go to a **dead-letter queue**

# Publish/Subscribe



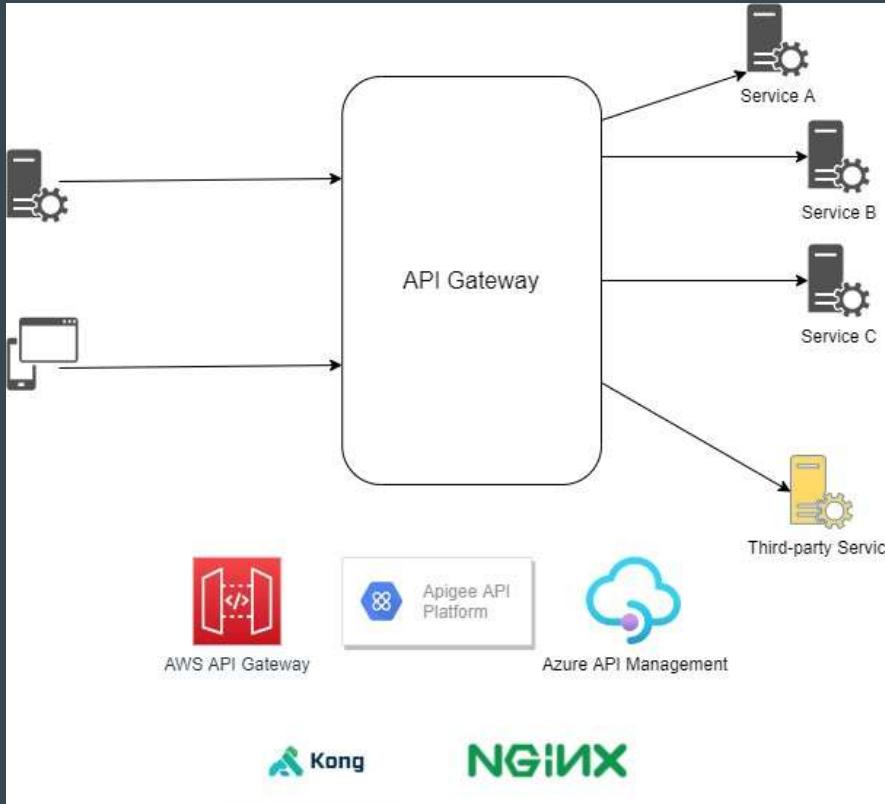
- Publishers publish messages
- Brokers manage messages
- **Subscribers receive messages (push)**
- All subscribers receive all messages (relevant to them)
- Some services store the messages, some don't (e.g. if there are no subscribers for AWS SNS, the message is lost)
- Some services offer message deduplication (to achieve **at most once delivery**) and strict ordering (e.g. Azure Service Bus)
- Usually offer dead-letter queues

# Streaming Services



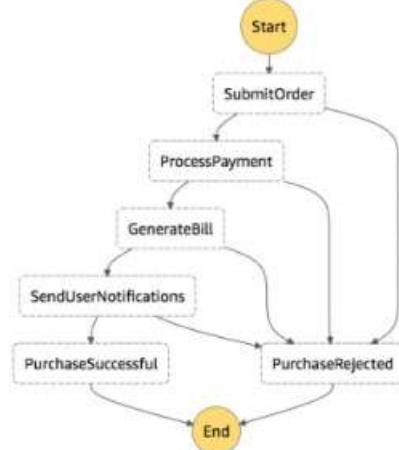
- **Producers** publish messages
- **Brokers** manage messages
- **Consumers** read messages from topics
- Each **consumer group** receives a copy of the message. Partitions are distributed across the consumers within a group.
- Each consumer is responsible to keep track of the messages (checkpoints)
- Partitions can be seen as **append-only files**
- Messages are not deleted, they expire (or they might be kept indefinitely - e.g. New York Times keeps all posts ever created in Kafka)

# API Gateways



- Usually based on **HTTP**
- Act as an entrypoint/front door/reverse proxy
- Handle request routing and protocol translation if necessary
- Usually provide additional features such as rate limiting and caching
- Common design pattern when working with microservices
- Also useful when monetizing APIs
  - API Management services

# Workflow Services



AWS Step Functions



Azure Logic Apps



Google Workflows



- Orchestrate processes involving multiple services
- Can be seen as a **finite state machine**
- Usually come with lots of connectors/integrations. E.g. dropbox, slack, stripe, google suite, office365
- Can span various time intervals (from seconds to years)
- Are usually based on FaaS and leverage other data stores for state tracking

# Cloud Applications Architecture

...

Course 11 - Application Security

# Security

## Confidentiality

- only the authorized entities (people and/or systems) have access to the data

## Integrity

- only the authorized entities can modify the data through well-defined procedures

## Availability

- there is no point in having a well-guarded system if no one can use it.
- Also, earthquakes tend to overrule highly sophisticated security measures



# Application Security

Create a secure environment for the user and his/her data.

Common processes/techniques:

- Authentication
- Authorization
- Encryption
- Input validation
- Hashing

# CWE (Common Weakness Enumeration)

1. Cross-site scripting (XSS) (46.82)
2. Out-of-bounds write (46.17)
3. Improper input validation (33.47)
4. Out-of-bounds read (26.5)
5. Improper restriction of operations within the bounds of a memory buffer (23.73)
6. SQL injection (20.69)
7. Exposure of sensitive information to an unauthorized actor (19.16)
8. Use after free (18.87)
9. Cross-site request forgery (CSRF) (17.29)
10. OS command injection (16.44)

See complete list (top 25) [here](#)

# CWE (Common Weakness Enumeration)

Most of these vulnerabilities are addressed by modern frameworks, libraries, and even languages.

Our duty is to keep them updated and to choose well-tested ones.

Packages managers can help - e.g. [npm audit](#).

There are also dedicated tools such as [snyk](#)

# Authentication (AuthN)

Identify and ensure that the users are indeed whom they pretend to be.

Usually involves something the user:

- Has (e.g. smart card, code/token generator)
- Knows (e.g. password, the name of the first pet)
- Is (e.g. biometrics)

Multi-factor authentication = using 2 or 3 factors from above.

Flows and approaches are standardized - allows us to delegate it

# Authorization (AuthZ)

Grant access to a specific **resource** based on a set of **roles, permissions or privileges**.

Takes place on each action.

Flows and processes are partially standardized. E.g. authorization on [google](#) and [github](#) works the same, but uses different **scopes** (defined by each of them).

# Identity Providers (IdP)

Whenever we register on a website/platform, we create a new identity for ourselves on the internet.

Certain providers are more authoritative:

- We can use our gmail address on both Google and Facebook, but only Google can guarantee that we are indeed the owner of that address.

Facilitate **identity federation (social login)**.

E.g. Google, Facebook, Twitter, Linkedin, Microsoft, Apple, etc.

# OAuth 2.0

Link

**Authorization** protocol based on several widely adopted flows.

Enables us to delegate authorization without providing our credentials.

Implemented by most providers and platforms.

## Are your friends already on Yelp?

Many of your friends may already be here, now you can find out. Just log in and we'll display all your contacts, and you can select which ones to invite! And don't worry, we don't keep your email password or your friends' addresses. We loathe spam, too.

Your Email Service



Your Email Address



Your Gmail Password



Welcome  
75451442@qq.com

(our Gmail email)

storage.com wants to

View and manage the files in your Google Drive

View and manage Google Drive files and folders that you have opened or created with this app

View the activity history of your Google apps

Allow storage.com to do this?

By clicking Allow, you allow this app to use your information in accordance with their terms of service and privacy policies. You can remove this or any other app connected to your account in My Account

CANCEL

ALLOW

# OAuth 2.0 Terminology

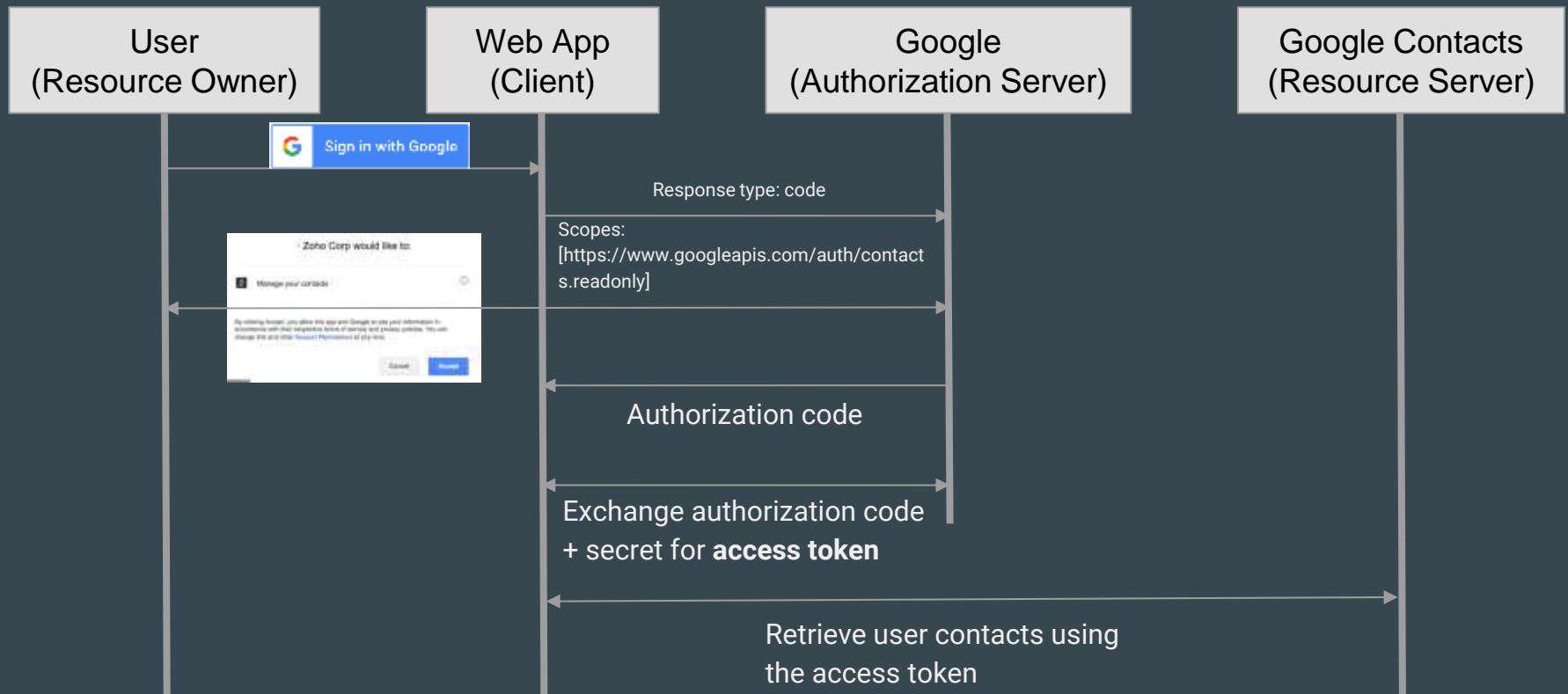
- **Resource Owner:** Entity that can grant access to a protected resource. Typically, this is the end-user.
- **Client:** Application requesting access to a protected resource on behalf of the Owner.
- **Resource Server:** The API you want to access.
- **Authorization Server:** Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization.
- **User Agent:** Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).

Source - [Auth0](#)

# OAuth 2.0 - Flows/Grants

- **Authorization Code**: Used by web applications (ones built on frameworks such as Spring or Laravel). Requires a server to securely exchange information (including a **secret**) with the authorization server.
- **Implicit**: Less secure, but doesn't require a server (used for Single Page Applications and mobile apps) (obsolete).
- **Authorization Code with PKCE**: Improved version the implicit flow.
- **Resource Owner Password Credentials (ROPC)**: Involves the user providing the username and password. Intended only for special/legacy scenarios.
- **Client Credentials**: Intended for secure environments (e.g. server to server)

# OAuth 2.0 - Authorization Code



# OAuth 2.0 - Other Flows

With **implicit flow**, the access token is provided directly by the authorization server without requiring the authorization code and secret.

**Authorization Code with PKCE** is similar to simple authorization code, but doesn't involve a server to run the client (and store the secret). Instead it relies on the SPA/mobile app to generate a **code challenge** (sent with the initial request to the authorization server) and a **code verifier** (sent alongside the authorization code).

**ROPC** and **Client Credentials** simply involve exchanging the credentials for the access token.

# OpenID Connect

Built on top of OAuth 2.0. [Link](#)

Facilitates **authentication**. (the access token contains no information about the user)

Works with [JWT tokens](#) (base64 encoded; can contain arbitrary data) - known as idToken (as opposed to access token).

To retrieve an idToken, simply add *openid* to the scopes and *id\_token* to the response type. The authorization server will provide the idToken alongside the access token (and optionally *refresh\_token*).

# Working with JWT Tokens

The JWT tokens can be validated **offline**

- The resource server does not have to call the authorization server in order to validate the token.
- Downside is that tokens might be out of sync/**stale** (in case any information contained in the token was changed since it was issued).
- Online validation means calling the authorization server each time.

Compared to session based authentication (usually implemented using cookies), token based authentication is stateless (since the entire info about the user can be stored in the token). This means better/easier horizontal scalability. For highly secured scenarios, we might store the tokens on the server and keep track of invalidated ones.

**We can include other info about the user such as custom roles and permissions.**

# Managed Authentication Services

Firebase Authentication

AWS Cognito

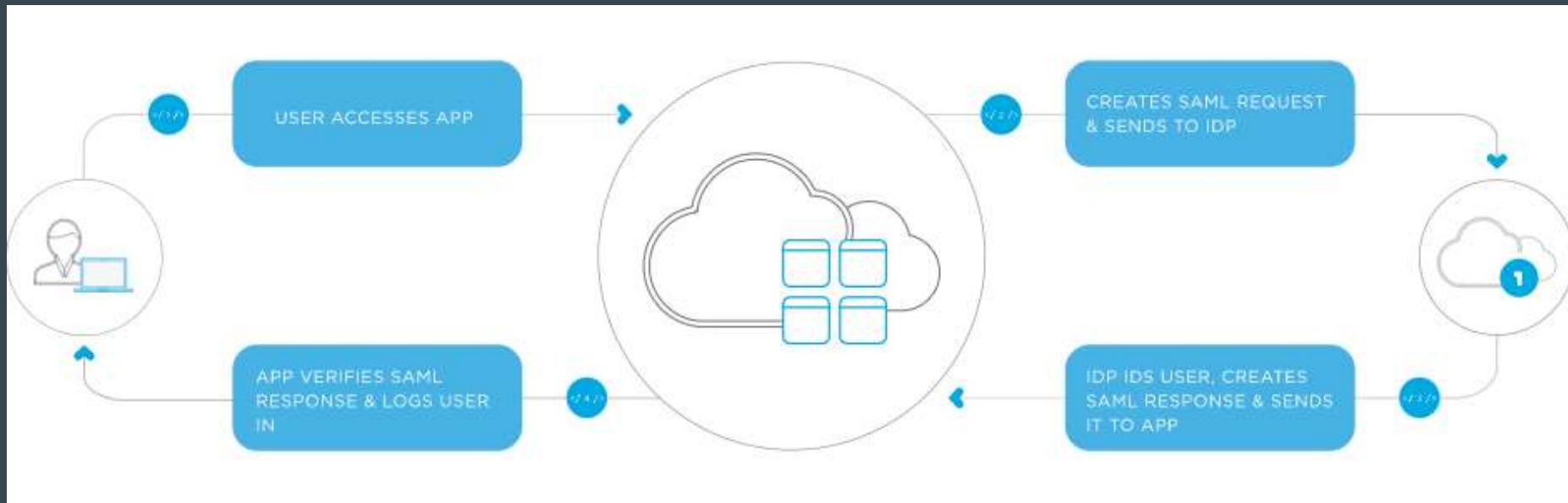
Azure Active Directory B2B (standard) and B2C

Auth0

Okta

OneLogin

# SSO and SAML



# Encoding, Hashing, Encryption

**Encoding**/decoding is used to facilitate communication

- E.g. in order to store some data in an URL, it has to be URL encoded (i.e. escape certain characters such as spaces and ?).
- Can be reversed (decoding) - **provides no security**.

**Hashing** is used to verify the integrity of data and to store passwords.

- Integrity is verified by comparing a given checksum to the one generated by ourselves.  
E.g. to ensure that a file was completely received and authentic
- Cannot be reversed.

**Encryption** is used to safely send and store data.

- Can be reversed (decrypted) if both the algorithm (cypher) and the key are known.

# Encryption (Cryptography)

One of the pillars that enable us to do more online (work, make purchases, etc).

- Enables secure **transfer** of data (TLS, IPSec).
- Enables secure **storage** of data (data at rest).
- Enables **digital signatures**.

2 types of encryption:

- **Symmetric**: one key, works with large data (e.g. [AES](#))
- **Asymmetric or public key**: 2 keys - **public** (encryption) and **private** (decryption) (e.g. [RSA](#)). Cannot handle large data.

There is also **envelope** encryption which combines both (e.g. AES for the actual data and RSA for the AES key).

# Encryption in Cloud

Offered by all providers for most storage services.

Most of the times enabled by default and transparent to developers and users.

Dedicated services (main role is to store encryption keys) such as AWS KMS, Google KMS or Azure Key Vault.

Can be entirely handled by the provider (generating and managing the keys and data encryption/decryption) or we can choose to **supply/provide** our own keys.  
E.g. AWS S3 encryption, Google storage encryption.

# HTTPS, SSL, TLS

**SSL (secure sockets layer)** - Developed by Netscape in 1995.

**TLS (transport layer security)** - the successor of SSL standardized by IETF (i.e. SSL v4).

The SSL term is still used today especially for **SSL Certificates**.

A website/server that supports TLS is HTTPS enabled.

**Based on RSA** - allows browsers to verify the authenticity of the origin/server.

Secure connection established through the **SSL/TLS handshake**.

# Secrets Management

Similar to cryptographic key management.

Where do we store API keys, client secrets (e.g. for OAuth), and passwords?

# Active Directory

Users, groups, domains, forests

Azure Active Directory

Hybrid Identity

AAD B2B vs B2C

# Resources

[Authentication on the Web \(video\)](#)

[OAuth 2.0 and OpenID Connect \(video\)](#)

[Transport Layer Security \(TLS\) - Computerphile](#)

# Cloud Applications Architecture

...

Course 12 - Serverless

# What is Serverless?

There still are servers, just entirely abstracted by the providers.

A service can be considered serverless **if and only if** (iff):

- you don't have to setup any infrastructure
- you don't have to provision/scale capacity
- you pay only for what you use
- the service can scale down to 0

# Pros/Cons

- **you don't have to setup any infrastructure**
  - Less to manage ✓ ✘
  - Getting started quicker ✓ ✘
  - Less control ✘
  - Vendor lock-in ✘
- **you don't have to provision/scale capacity**
  - Can scale incredibly high ✓ ✘
  - You might want to limit the scale (native ways to limit scale were introduced)
  - Might overload dependencies (e.g. databases) ✘

# Pros/Cons

- **you pay only for what you use**
  - Scales with your business ✓ ✘
  - You have to pay to keep it running
    - even if it ends up costing more than expected ✘
- **the service can scale down to 0**
  - Great for multiple environments and prototypes ✓ ✘
    - Dev resources are not needed outside of the business hours
  - **Cold starts** ✘

# Classification

Usually refers to functions as a service (FaaS), but there is more

- **Compute**
- **Workflows**
- **Databases**
- **Integration**
- **Storage**
- **Analytics**
- **Monitoring**
- **Development/build tools**

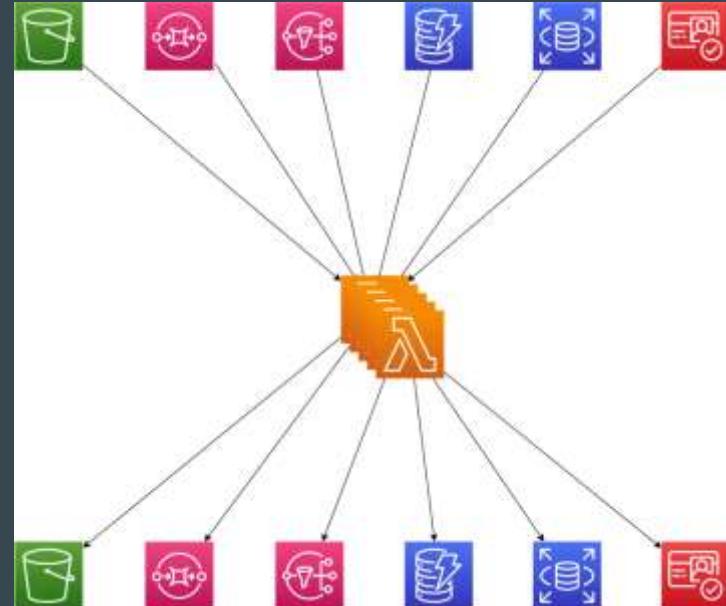
# Serverless Compute

- FaaS (functions as a service)
  - E.g. [AWS Lambda](#), [Azure Functions](#), [Google Functions](#), [Cloudflare Workers](#)
- Certain container services
  - E.g. [Google Cloud Run](#), [AWS Fargate](#)
- Certain PaaS compute services
  - E.g. [Google App Engine](#), [Azure App Service](#)

# FaaS

Excellent for handling events (especially background events)

- Messages from integration services
- Authentication/user related actions
  - E.g. send custom email on sign-up, store user info in DB
- Changes in the database
  - Especially useful for NoSQL databases in which data consistency must be maintained programmatically due to its denormalized nature.
- File uploads



# FaaS for HTTP

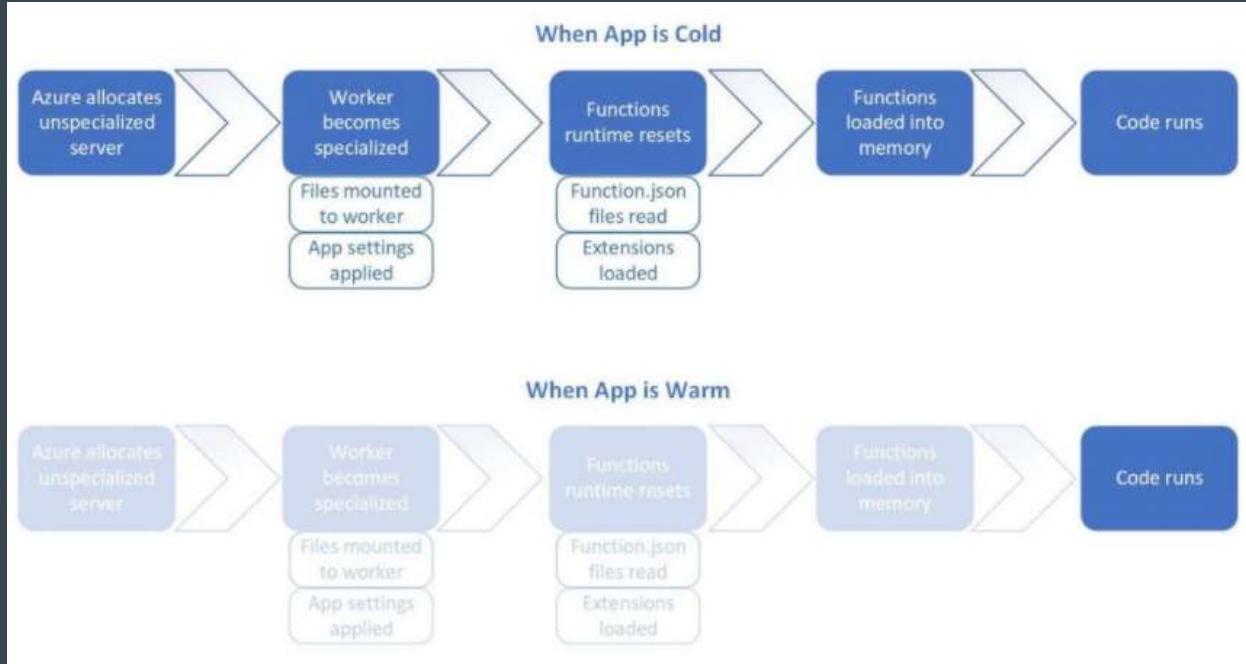
HTTP requests (e.g. through a REST API) are still events, but the user waits for a response.

Some FaaS requires additional integration services (e.g. Lambda requires API Gateway to be exposed as HTTP endpoints), while others provide URL endpoints automatically (Google and Azure).

- Both Google and Azure leveraged their PaaS services for FaaS (Google App Engine, Azure App Service). AWS built it from the ground up on [Firecracker](#).

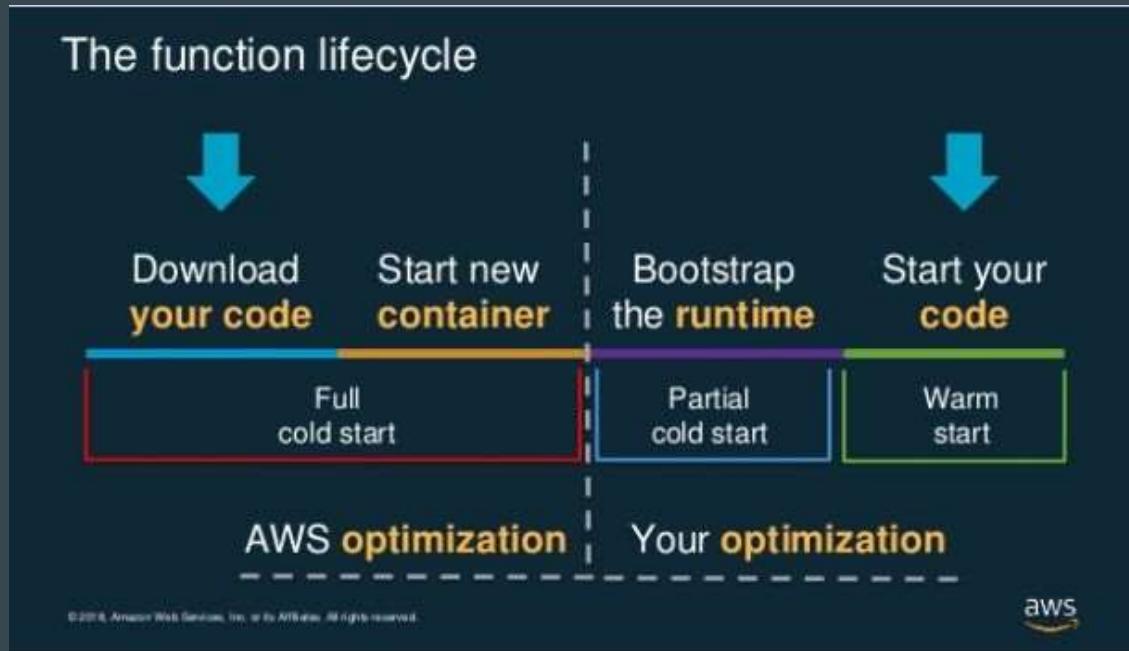
Cold-start is a common issue for HTTP.

# Cold Starts



Azure Functions Lifecycle

# Cold Starts



AWS Lambda Lifecycle

# Cold Starts - Improvements

We can try to keep the functions warm:

- Call the function on a schedule (e.g. every 15 minutes)
- Good enough solution, but might still run into cold-starts when scaling up the underlying servers
  - One server runs multiple functions
- Can be scheduled either using dedicated services (e.g. [Google Cloud Scheduler](#)) or using features of Load Balancers or Monitoring Software.

Some providers might offer a way to provision a minimum amount of functions that are always ready (deviates a bit from the serverless principles):

- [AWS Lambda Provisioned Concurrency](#)
- Azure Functions [Premium Plan](#) and [App Service Plan](#)

# FaaS - Common Limits/Quotas

## Resources

- Memory
  - We can usually control the amount of memory available to each function
  - CPU scales proportionally
  - **Maximum memory is limited** (e.g. 10GB for Lambda, 4GB for Google Functions)
- Deployment size
  - Usually limited to a few hundred MBs.
  - Dependencies might cause issues
- Concurrency
  - New functions will be instantiated if needed, but up to a point (commonly 1000).
- Input
  - The size of the input data is usually limited to a few MBs

# FaaS - Common Limits/Quotas

## Duration

- **Functions were not designed for long running processes**
  - E.g. you cannot run a Wordpress website on them
- Commonly limited to a few minutes (e.g. max 15 minutes for AWS Lambda)
- They also become really expensive when running for long (longer than a few seconds).

See the complete service quotas here:

- [AWS Lambda](#)
- [Google Cloud Functions](#)
- [Azure Functions](#)

# Serverless Databases

- “Serverless-Native” - designed with the serverless paradigm in mind (NoSQL)
  - E.g. [Google \(Firebase\) Firestore](#), [FaunaDB](#)
  - No provisioning, just pay for data storage and operations (reads, writes, deletes)
- Hybrid, NoSQL - have a serverless mode alongside a provisioned mode
  - E.g. [AWS DynamoDB](#), [Azure Cosmos DB](#) (still in preview)
  - Usually based on some capacity units (e.g. Cosmos DB has request units) that are managed by other services (e.g. AWS Cloudwatch).
- Hybrid, SQL
  - E.g. [AWS Aurora Serverless](#), [Azure SQL Database](#) (Serverless Tier)
  - Scaling still takes a bit of time, but services are getting better (e.g. Aurora Serverless v2)

Serverless databases tend to be more expensive than their provisioned counterparts (when running continuously)

# Serverless Storage

Most object storage services are serverless

- E.g. AWS S3, Azure Blob Storage, Google Cloud Storage

Are usually leveraged by serverless compute services for storing the application files.

Can act as a trigger for compute - e.g. when an image is uploaded, a function is invoked automatically to resize the image (be careful to not run into infinite loops)

# Closing Remarks

- Trying out ideas and building is more accessible than ever
  - Most projects can stay within free tiers
- Massive scale - can be good, can be bad
- Infinite loops are costlier than ever
- NoSQL and serverless is a great match
- There is room for both serverful and serverless paradigms

# Resources

AWS My Architecture - Nordstrom

David Schmitz' Presentation

Where Should I Run My Code? ('19) (also see the '18 one)

Azure Functions Cold Starts

Serverless Horror Stories

# Cloud Applications Architecture

...

Course 13 - API Design

# Definition

## **API = Application Programming Interface**

Might refer to different layers:

- Operating system functionality (i.e. system call) (e.g. for reading files)
- Communication between tiers (e.g. ORMs for interacting with the DB)
- Communication between services/applications over the network
  - This is the subject of the course

# Motivation

## Adoption/Popularity of APIs in general

- Both consumption and production
  - Many SaaS products offer APIs (e.g. Stripe)
- Even for large corporations
  - Traditionally they implemented their own software

## Microservices architecture

- Common approach for new projects
- Services need a way to communicate to each other

# Consideration

While there are multiple alternatives for designing APIs, there is no approach or technology that fits all cases.

Our duty is to choose the approach that makes the most sense for our current use case/needs.

Some designs might facilitate performance, others discoverability while others might be built to support a wide range of use cases.

# Contexts/Use-Cases

Mobile applications

Websites

Inter-service communication (e.g. microservices)

Legacy systems (e.g. older ERPs/CRMs)

High-performance compute

# API Styles

## Procedure/Action Oriented

- Data sits behind functions
- Similar to writing code in general
- Emphasizes the **behavior** of the system
- Usually results in **large** set of procedures (and is ever expanding)
- Each system has its own set of procedures

## Entity/Resource Oriented

- Data is the main actor
- The behavior of the system is based on the data
- Similar to working with databases
  - System-specific data model
  - Common well-known features/capabilities
- Usually results in concise set of operations that can be used across multiple use cases

# Data Formats

JSON

XML

YAML

Protocol Buffers (Protobuf)

Avro

Parquet

...and more - check out [the comparison on wikipedia](#)

# Overview



API styles over time, Source: [Rob Crowley](#)

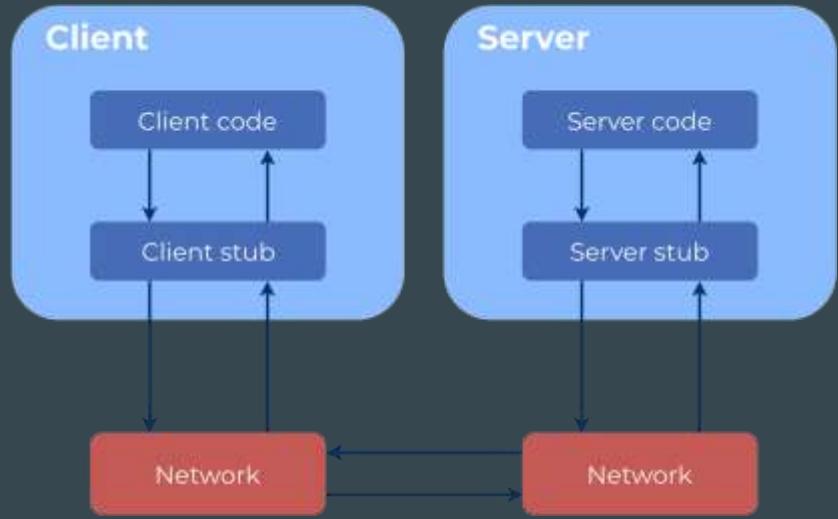
# RPC

## Remote Procedure Call

Instead of calling a function/method locally, we call over the network in a different system.

Implemented by various protocols

- XML-RPC, JSON-RPC
- gRPC (used in many Google products, e.g. firebase; also heavily used in microservices architectures)



```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

Example from the JSON-RPC Spec

Tends to achieve high performance due to low overhead.

# SOAP

Stands for **S**imple **O**bject **A**ccess **P**rotocol

One of the first protocols to standardize how information is exchanged over the network

Based on RPC

Only works with XML

WSDL (Web Service Description Language)

- Used to describe the capabilities of a web server
- Tools/frameworks can generate stubs/code based on it

# SOAP

## Pros

- Widely used and adopted
- Establishes an interface contract
  - Important for businesses
- Large number of extensions
  - WS-Addressing, WS-Security, etc.
  - See more [here](#)

## Cons

- XML only
- Not suitable for high performance/throughput requirements
  - XML tends to be verbose and slow to parse
- Tight coupling (since it's RPC)

# REST

## REpresentational State Transfer

Introduced by Roy Fielding in his doctoral thesis in 2000 - [link](#)

Most commonly performed over HTTP (Roy also contributed heavily to HTTP/1.1)

- Resources (e.g. cars, comments, movies) are represented by URLs (e.g. <https://icdb/cars/1>)
- Aspects such as cacheability and data format can be controlled through headers
- Each resource might support multiple operations, each corresponding to an HTTP verb
  - List: GET /movies
  - Read: GET /movies/1
  - Create: POST /movies
  - Update: PUT /movies/1 (expects the entire representation of the movie), PATCH /movies/1 (expects only the fields that must be updated)
  - Delete: DELETE /movies/1
- HTTP status codes indicate the result of the operations (e.g. 201 = resource created)

# REST

It introduces a set of constraints such as **statelessness**, **cacheability**, and the most distinctive one, **uniform interface** which requires:

- **Resources are identified by URIs**
  - E.g. when reading the details about a car, besides the expected information such as model and engine, we also get a URL pointing to the details about the manufacturer and the URL of the car itself
- **Representations of the underlying resources provide enough information to enable us to change the resource's state.**
- **Self-descriptive messages (requests/responses)**
  - Each message contains all the information needed to be able to consume it (e.g. contains the format of the data)
- **HATEOAS**

```
        "account": {  
            "account_number": 12345,  
            "balance": {  
                "currency": "USD",  
                "value": 100.00  
            },  
            "links": {  
                "deposit": "/accounts/12345/deposit",  
                "withdraw": "/accounts/12345/withdraw",  
                "transfer": "/accounts/12345/transfer",  
                "close": "/accounts/12345/close"  
            }  
        }  
    }  
}
```

# REST

In practice, only a small percentage of APIs can truly be considered **RESTful** (mostly because of missing HATEOAS and URL identifiers).

For simplicity (and in lack of a better term), the term REST is commonly used to identify **resource-based** APIs that follow the **HTTP conventions** (verbs, status codes, headers).

While there are many good examples of REST APIs, [SWAPI \(Star Wars API\)](#) is among the first examples people give.

Learn more about REST [here](#)

# REST Standardization

## OData (Open Data Protocol)

- Rarely used by new projects, but still heavily used by Microsoft (e.g. the [Azure REST API](#)) and SAP among others.
- Standardizes aspects such as:
  - Filtering: e.g. `$filter=createdAt ge datetime'2017-06-01T00:00:00'` (entities created after 01.06.2017)
  - The service document (the root of the API): provides links to the resources
  - The metadata document (`/$metadata`): provides a representation of the data model and the supported operations.

## Message improvement standards (format and/or description - hypermedia)

- [JSON:API](#), [HAL](#), [JSON-LD](#), [Collection+JSON](#), [Siren](#)
- See [this article](#) for more info and how to choose between them

# OpenAPI/Swagger

A standardized way to describe RESTful services (somewhat similar to WSDL)

Useful for:

- Generating documentation (tools such as [Swagger UI](#) and [ReDoc](#))
- Helping developers understand the API
- Generating clients/SDKs and server stubs (tools such as [OpenAPI Generator](#))
- Exposing APIs through certain API Gateways (e.g. AWS API Gateway, Azure API Management, Google Apigee).
- Testing the API (e.g. Postman can import OpenAPI specs)

There are other alternatives such as [RAML](#), [API Blueprint](#), [WADL](#)

It's recommended to maintain a specification when building a RESTful API.

# REST

Pros	Cons
<ul style="list-style-type: none"><li>• <b>Widely used and adopted</b></li><li>• (Somewhat) Loose coupling</li><li>• Well suited for public APIs</li><li>• Low overhead</li></ul>	<ul style="list-style-type: none"><li>• Multiple requests might be needed to retrieve all the data for a given page/screen</li><li>• Always returns all the fields of the resource (overfetching) (relevant especially for mobile devices and/or very large data sets)</li><li>• No shared definition/specification of how to build REST APIs</li></ul>

# GraphQL

Open sourced by Facebook in 2015

- Internally used since 2012 for the mobile apps
- Designed specifically for optimizing APIs at their scale (and traffic)

Has 3 main operations:

- Queries - for reading data
- Mutations - for changing data
- Subscriptions - for real-time data reading

Examples: [Star Wars GraphQL API](#), [SpaceX API](#), [Github](#)

Do not confuse with Microsoft and Facebook Graph APIs (which are RESTful)

# GraphQL

## Pros

- Shared definition (everyone implements GraphQL APIs in the same way)
- Introspection (also provides strong data typing) (enables service discovery - this is why GraphiQL works) (think of OpenAPI for REST)
- **Specify what to fetch**
- Versioning (likely) not needed (Facebook didn't have to version their GraphQL API since 2014)

## Cons

- Hard to reference entities from other systems
- Hard to cache (client libraries such as [Apollo](#) usually handle caching though)
- Adds complexity (on both the server and the client)
- Hard to scale (mostly due to real-time functionality)
- Doesn't support proxies

# Related Cloud Services

## API Gateways/Management Services

- [AWS API Gateway](#)
- [Azure API Management](#)
- [Google Apigee](#)
- [Kong](#)

## Managed Services

- AWS AppSync
- Hasura

## Callable Functions

- AWS and Google functions can be called as RPC (through their SDKs)

# Resources

[RPC vs REST vs GraphQL](#)

[Altexsoft comparison of API Styles](#)

[Designing Quality APIs \(Cloud Next '18\)](#)

[GraphQL: The Documentary](#)

[Building Modern APIs with GraphQL](#)