Cloud Computing and Big Data

Josh Felmeden

October 27, 2021

Contents

1 Overview

1.1 Economic Driving Factors

Cloud computing works by charging someone to use a service for a certain amount of time. If, for example, you let someone use your computer, you might charge them for their usage (if you were a real meanie). How that would be worked out is the *operation expenditure* (Opex) and the *capital expenditure* (Capex, cost of the computer). Combining these, we calculate the *total cost of operation* and calculate the cost per day of the lifespan of the computer (in this example). So, if you used the computer for an hour, you would owe the person a 24th of this daily cost.

Now, this might come down to half a penny, but of course, this doesn't exist any more, so we might charge a whole penny instead. This seems like a marginal profit, but in terms of percentages, this is a 100% increase. The fundamentals of this are how cloud computing generates so much income.

The attraction of this is that the users of the services do not have to pay the Capex, and simply pay opex for the rental of the service.

1.2 Normal Failure

Failures in these systems are to be expected. Say the servers you are buying are guaranteed to have a 99.999% 3-year survival rate ('five-nines reliability'). This is good because there is a very high chance that this remains. Now, if you buy 10 of these servers, the probability that you have all of the servers still working is only 99.99%. Taking this to the extremes, if you buy 500,000 servers (this is standard practise for a lot of the big servers these days), the probability that all of them are still working is a measly 1%. Essentially, failure is something that should be normalised.

Modular data centres are used a lot in big data centres. A unit may be left in the shipping container, and this container is removed or added as a container, meaning that if one module fails, another can just be replaced, meaning that the whole centre doesn't collapse.

In the current climate, the modular centres are considered a whole working unit, and this way of thinking was mobilised by a group of Google engineers.

1.3 Blank as a service

- **SaaS: Software as a service**
  - End user application software that is remotely delivered over the internet.
  - Adobe is an example of this (Adobe Creative Cloud)
  - Used to be bought off the shelf as a CD
- **PaaS: Platform as a Service**
  - Developer application software (middleware) functionality is remotely accessible
  - Might provide a particular combination of OS, web-server, data-base and scripting
  - Popular instance is the free 'LAMP stack' (Linux, Apache, MYSQL and PHP)
  - Used to be dominated by *Google*

- **IaaS: Infrastructure as a Service**
  - IT infrastructure almost always virtualised and remotely accessible.
  - Virtualisation software allows one physical server to be used by multiple users, each on a virtual machine. If one crashes, the others keep running.
  - Used to be dominated by *AWS* (although this is now a much bigger thing).

While Google and Amazon dominated their respective fields, both of these companies have expanded into the other fields. This is very complicated, but the key thing is that both Google and Amazon offer both PaaS and IaaS.

In around 2015, Amazon created **FaaS: function as a service (aka 'serverless')**:

- No server processes visibly running. Pay only for the time spent executing a function
- UNlike PaaS, scale out without increasing number of servers.
- Amazon Lambda is the best known example, although both Google and Microsoft have answers to this.

1.4 Impacts of Cloud Services

Cloud services have revolutionised business in many ways. It is now possible to do tasks that would previously require access to high performance machines. Instead, it can be sent to a big data centre, and this usage is just charged as rental.

Interoperability is a big issue. **Vendor lock-in** is a concern for a lot of clients, and this simply means that once a client is locked into a vendor, it becomes financially or practically unviable to switch to another supplier. This led to an attempt to develop a sense of unity between cloud companies, where companies created the *Open Cloud Manifesto*. A lot of big companies signed up to this, but a lot of the top dogs didn't sign up for this (unsurprisingly, Google and Amazon). This manifesto seemingly no longer exists as an original. As a consumer, this is bad and worrying because vendor lock in is very possible.

2 IaaS and AWS

2.1 Amazon Simple Storage Service

Amazon S3 is a cloud-based persistent storage. It operates independently from other Amazon services. The simple refers to the features, not that it's simple to use. You can store data in the cloud. You also don't store files, you store *objects*, and these are kept in buckets. Objects have a size limit (5Tb) and a max size on a single upload is 5Gb. All buckets share the same namespace, so no sub-buckets.

It's very easy to use; just use a web GUI that is similar to AWS. It also has a command line interface and has scripting. Default storage can be selected (geographically).

S3 is accessed via API, either by SOAP (xml) or REST (http). Wrappers are available to abstract the

API for programmers.

This is just the storage, so now we will look at the computing of data in the cloud.

2.2 Amazon Elastic Compute Cloud (EC2)

This is a remotely accessible virtual network of virtual servers. Usually, EC2 is run with S3 providing the storage.

A single EC2 virtual server, with the chosen OS etc, is an instance. An instance is instantiatied from an Amazon Machine Image (AMI)

- One AMI can be cloned n times to create n instances
- You can build your own by cloning an AMI from your local server
- Or, Amazon have a bunch of prebuilt AMIs that you can choose from

EC2 dynamically assigns a unique IP address to each instance, and this IP can be reassigned, perhaps to someone else. The IP can also be static (also known as an Elastic IP address) at a cost.

EC2 instances run in availability zones (AZs), grouped into regions. AZ is similar to a single data centre, guaranteeing an area has 99.95% uptime.

2.2.1 Usage

Some basic API routes as S3, command-line or a bit of GUI too:

- Amazon's own AWS web console
- Various EC2 plug-ins for browsers
- Third-party cloud management tools

Some AMIs are junk or malware, however, so be careful when selecting this.

There are three types of storage:

- Ephemeral local storage in the instance (dies with instance)
- Persistent cloud (S3)
- SAN-style Elastic Block Storage (EBS)
  - Allows user to create volumes from 1Gb to 1Tb
  - Any number of volumes may be mounted from a single instance

S3 is slow, medium-reliable, but super-durable. Never loses data, so is good for DR backups. Instance storage is simple and cheap, but speed can be really poor. EBS is high on everything, but is complex and costly.

There is some autoscaling based on matrics:

- **Cloudwatch**: automated monitoring of EC2 instances. Reveals many statistics such as CPU utilisation, disk reads/writes and network traffic. Aggregates and stores monitoring data that can be accessed

- **Auto Scaling**: dynamically adds or removes EC2 instances based on CloudWatch metrics. You define conditions upon which you want to scale up or down your EC2 instances. Auto scaling automatically adds or removes the specified amount of Amazon EC2 instances when it detects that the conditions have been met.
- **Elastic Load Balancing**: automatically distributes incoming application traffic across multiple EC2 instances. Better fault tolerance. Elastic Load Balancing detects unhealthy instances within a pool and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. Customers can enable ELB within a single AZ or across multiple for consistent application performance.

2.3 AWS Simple Queue Service SQS and Architecting for Scale-out

SQS is reliable, loosely-coupled fault-tolerant storage and delivery of messages. It can be between any clients or computers connected to the internet, and senders and recipients do not have to communicate directly. No requirement that either side be always-available or connected to the internet.

A **message** is up to 256Kb of text-data, sent to SQS and stored until it is delivered. A **queue** serves to group related messages together.

SQS is accessible to clients on any HTTP-enabled platform. Messages are stored redundantly over multiple data-centres truly distributed. Unfortunately, this brings about a few down-sides:

- Message retrievals may be incomplete
- Messages may not be delivered quickly (2-10 seconds)
- Messages may be delivered out of order
- Messages may be redelivered

2.3.1 Mitch Garnaat's Monster Muck Mashup

This is a service that converts AVI videos to mp4 using:

- 'Boto' Python interface to AWS
- S3 to store the video files
- EC2 to do the conversion processing
- SQS for inter-process communication

Uses AWS for *scalability* (scale-out not up, not just buying new resources, adding cheap machines to improve ability.)

The basic steps are:

1. Upload a bunch of video files to a S3 bucket
2. For each file, add a msg to SQS input queue
3. On the EC2 instance, repeat this until input queue is empty:

a) Read message MI from input queue
b) Retrieve from S3 the video VI specified in MI
c) Do the conversion creating VO
d) Store VO in S3
e) Write message MO to SQS output queue
f) Delete MI from input queue

This really illustrates how good AWS is at scaling software. It is good because any number of clients can connect to the bucket. If this bucket is 100% full, it doesn't matter, because additional instances can all talk to the same buckets and queues, so the workload is met.

2.4 AWS simpleDB and AWS Relational Databases (RDB)

2.4.1 Amazon SimpleDB

This software provides:

- Reliable storage of structured textual data
- Languages that allows you to store, modify, query, and retrieve data sets
- Automatic indexing of all stored data

SimpleDB provides 3 main resources:

- **Domains**: Highest-level container for related data *items*: queries only search within one domain
- **Items**: a named collection of *attributes* that represent a data object. Each item has a unique name within the domain; items can be created, modified, or deleted; individual attributes within an an item can be manipulated
- **Attributes**: an individual category of information within the item, with a name unique for that item. Item has one or more text string values associated with the name

The downside to this type of storage is that it really does only do one data type (textual). If, for example, you wanted to store pi, you would need to store it as a character string ('3', '.', '1').

There is no Database **schema**, meaning if you mistype something, the database will just accept it as a definition, leading to some unfortunate results. It is not a traditional relational database management system:

- SimpleDB items are stored in a hierarchical structure, not a table
- SimpleDB attribute value max size is 1Kb
- SimpleDB data is all stored as text
- The query language is really basic
- SimpleDB is distributed, so data consistency may suffer due to propagation delays

2.4.2 Amazon Relation Database Service

There is also a relational database system, possibly as a response to Azure. You can set up, operate and scale a full MySQL RDBMS without having to worry about infrastructure provisioning, software maintenance, or common DB management tasks, like backups.

The processing power and storage space can be scaled as needed with a single API call and you can fully initiate fully consistent database snapshots at any time. Can import a dump file to get started. Each DB instance exports a number of metrics to CloudWatch including CPU utilisation, etc.

2.5 Availability Zones

Availability zones are clusters of independent data centres that are up to 20 miles apart. They are interconnected using low latency and enable fault isolation and HA.

Choosing which region to use comes down to a few reasons:

- Data sovereignty and compliance
  - Where are you storing user data?
- Proximity of users to data
- Services and feature availability
- Cost effectiveness
  - Each region has differing costs

High Availability (HA) is the ability to minimise service downtime by using redundant components. It also requires service components in at least two AZs.

Fault tolerance is the same as HA, but also the ability to ensure that no service disruption by using active-active architecture meaning that all components are active all the time. This is of course a lot more costly that just having HA.

IaaS may have HA, but FT unlikely. PaaS will usually have HA, but some services offer FT.

3 Virtualisation

The most fundamental type of cloud computing is IaaS compute, and the most fundamental type of this is the virtual machine. It has unmanaged services, which means you control what they do. You create, save or reuse them. They are networked and connected to storage and have certain security systems.

EC2 instances are a form of virtualisation.

- They mostly run on Xeon processors, but also have other processors available.
- There are lots of different tiers available on AWS that use different purposes
- They run in AMIs.

- After creating a virtual machine, you can start, stop, and terminate the machines. Once the machine is terminated, it will not come back.

In virtualisation, there is some virtual memory that points to addresses in physical memory. It is an abstraction of the storage resources, because the virtual memory seems contiguous, but in physical memory it could be in various locations. The operating system manages this and also has hardware support.

Virtual Memory	**Virtual Machine**
Abstraction of the RAM memory resources	Abstraction of the storage/process/IO resources.
Mapping of program (virtual) memory addresses to physical addresses	Time slicing VM use of virtual memory addresses/CPU/IO registers in physical addresses.
Operating system manages	Operating system/Hypervisor manages
Hardware support (memory management unit)	Hardware support (e.g. Intel VT)

3.1 The hypervisor

This virtualisation is not a new idea. It has been in use since around 1960, but it has been formalised in 1974, where they defined three important properties:

- *Fidelity*: Program gets the same output whether on VM or hardware
- *Performance*: Performs close to physical computer
- *Safety*: Cannot change or corrupt data on physical computer

This was ensured by analysing the instruction sets and identified two instructions that are the most important: **sensitive** instructions (that can change configurations of resources) and **privileged** instructions. Sensitive instructions need to be caught by the operating system, jumping from user mode to kernel mode. Therefore, all sensitive instructions need to be a subset of privileged instructions. This is all handled by the hypervisor.

Therefore, we have the hypervisor in contact with the server hardware, and then on top of this, we have the VM (or multiple machines) that consist of a guest OS, middleware, and apps. This type of VM is called **bare metal**.

Another type, called **hosted**, has a host OS running on top of the server hardware, and then the hypervisor.

The difference between the two is clear; in type two, the hypervisor can write to the host OS meaning that it can be more lightweight, where as type one can be much faster, at the cost of having to contain stuff that is normally handled by the OS.

3.1.1 Simulated Hardware

Full virtualisation is a complete or almost complete simulation of the underlying guest-machine hardware: virtualised guest OS runs as if it were on a bare machine.

The alternative to this is *paravirtualisation* and is only possible when the source code of the OS is available. The guest OS is edited and recompiled to make system calls into the hypervisor API to execute safe rewrites of sensitive instructions. In this example, the hypervisor doesn't simulate hardware.

EC2 offers both of these virtual machines:

- HVM — hardware virtual machine
  - Virtualised set of hardware
  - Can use OS without modification
  - Intel virtualisation technology
- PV — Paravirtualisation
  - Requires OS to be prepared
  - Doesn't support GPU instances

3.1.2 Xen

Xen is a free, open-source hypervisor developed at University of Cambridge. It has multiple modes:

- Paravirtualisation: guest OS recompiled with modifications
- Hardware assisted virtualisation: Intel x86 and ARM extensions
- Widespread
- Not easily virtualisable (17 instructions that violate the sensitive rules above)

3.1.3 Areas of Virtualisation

There are three main areas of responsibility for virtualisation managers:

- CPU virtualisation
  - Guest has exclusive use of a CPU for a period of time
  - CPU state of the first guest is saved, and the state of the next guest is loaded before control is passed to it
- Memory virtualisation
  - Additional layer of indirection to virtual memory
- I/O virtualisation
  - Hypervisor implements a device model to provide abstractions of the hardware

3.1.4 KVM

Converts Linux into a type-1 hypervisor. Memory manager, process scheduler, I/O stack from Linux. The VM is implemented as a regular Linux process. KVM requires CPU virtualisation extensions to handle instructions.

3.1.5 Nitro Hypervisor

- Based on KVM
- Offloads virtualisation functions to dedicated hardware and software
- Hypervisor mainly provides CPU and memory isolation to EC2 instances
- Nitro cards for VPC networking and EBS storage
  - Can handle NVMe SSD for instance and net storage, transparent encryption
  - Nitro hypervisor not involved in tasks for networking and storage
  - In OS Elastic Network Adapter driver
  - Security groups implemented in the NIC
- Can run bare metal

4 Containerisation

A **docker** is essentially a shipping container system for code. They eliminate the problem of running code on loads of different platforms by simply shipping this container to a computer, and then the application can be run locally.

Containers also bring:

- Reproducability
- Portability
- Flexibility
- Isolation

It makes things like experiments really easy, because the runtime environment is always the same.

Container systems use a number of components:

- Linux containers (LXC)
  - Cgroups and namespaces
- Container runtimes
  - Executables that read the container runtime specification, configure the kernel
- Container images
  - Applications
- Container storage
  - Linux storage systems used to store containter images on copyon-write (COW) file systems
- Container registries
  - Web servers used to store container images
- Container engines
  - Container tools used to pull images from container registries and assemble them on the host before creating the runtime specification and running the container runtime
- Container image builders
  - Tools used to create container images

Docker is the most popular container system:

- A container image format spec
- Tools for building container images
- Tools to manage container images
- Tools to manage instances of containers
- A system to share container images
- Tools to run containers

The **Open Container Initiative** (OCI) created an open governance structure creating industry standards around container formats and runtime. The runtime specification and the image specification were also created as part of this.

On the client computer, you interact with the docker via a CLI. An image is built (or pulled or runned) on the docker daemon. These images are a read only template with instructions for creating a Docker container. You can provide a Dockerfile that will determine how the image is run. The **container** itself is a runnable instance of this image. They are *ephemeral*, meaning that unless specific action is taken, any changes that are not stored to a mounted persistent storage volume are lost.

4.1 Namespaces

Namespaces let you virtualise system resources like the file system or netowrking for each container. Each kind of namespace applies to a specific resource and each namespaec creates barriers between processes:

- **pid namespace**: Responsible for isolating hte process (Process ID)
- **net namespace**: Manages network interfaces
- **ipc namespace**: Manages IPC resources (IPC: InterProcess Communiccation)
- **mnt namespace**: Manages the filesystem mount points (MNT: mount)
- **uts namespace**: Isolates kernel and version identifiers and hostname (UTS: Unix Timesharing System).

4.2 Control groups

Cgroups provide a way to limit access to resources such as CPU and memory that each container can use. Each time a new container is named, a cgroup of the same name appears.

4.3 Container images

They are standard TAR files with a base image and layered with differences. The base image contains:

- Rootfs (container root filesystem)
- JSON file (container configuration)

4.4 COW systems

Instead of overwriting data, data is written somewhere else meaning better recovery. There is built in transactions also. There are snapshots, since we just take the current number of written layers and store a reference to that. If you make a modification to this, we can reconfigure any system with this reference point. Using this means that multiple containers can share the same base images, and just use the reference to the configuration they use.

4.5 Performance

Containers are almost always better than virtual machines. While they both emulate infrastructure and encapsulate the tenant, they differ in key ways:

- VMs provide hardware level virtualisation, while containers provide OS level
- VMs need tens of seconds of provisioning, while containers only require milliseconds
- Virtualisation performance is slower than containers in most dimensions except networking
- VM tenants are fully isolated while containers provide process level isolation to tenants.

5 Application Orchestration and Kubernetes

Orchestration is a number of things:

- Running containers at scale requires management tools
- Manage networking volumes, infrastructure
- Automate
  - Fault tolerance, self-healing
  - Auto scaling on demand
  - DevOps
  - Update/rollback without downtime
- The tools available for us to do this are Mesos, Docker swarm, and Kubernetes

5.1 Kubernetes

The features of Kubernetes include:

- Automatic scheduling of work based on resource usage and constraints
- Self-healing: automatic replacement and rescheduling of failed containers
- Service discovery and load balancing
- Automated rollouts and rollbacks

The most important things to get to grips with are the runtime objects: pods, deployments, and services.

5.1.1 Pod

A pod is a set of one or more containers that act as a unit and are scheduled onto a node together. They share a local network and can share file-system volumes.

5.1.2 Deployments

These describe the *desired state* through declarative updates for pods and ReplicaSets. Essentially, you describe what you want, submit this to Kubernetes, and then it is deployed how you want (without actually telling it exactly what you want). The ReplicaSets balance the number of scheduled and running pods (kills or creates).

The deployment is managed via the spec (what we want), the monitors status (current state), and the template (how).

You are able to create a deployment to rollout a ReplicaSet, declare the new state of the pods, rollback to a previous version, or scale up the deployment to cope with more load.

Deployments use **Yaml files** to describe them (similar to Python, in that it uses indentation to infer scope).

5.1.3 Services

Services define networking to access pods consistently. They also expose pods to the external world. They create groupings of pods that can be referred to by name and have a unique IP address and DNS hostname (these by default cluster scope only). The services ensure the pods that are in use are load balanced, and environment variables containing the IP address of each service in the cluster are injected into all the containers.

Services are defined by a *service config file* (also in Yaml).

There are a few service types:

- Influences networking configuration
- Cluster IP
  - Default
  - Service is discoverable/routable only within the cluster
  - kube-proxy watches API service and updates pod IPTables on change
- NodePort
  - Exposes the service on each Node’s IP at a static port (the NodePort)
  - Access the service via <NodeIP>:<NodePort>
  - The simplest way to make your service externally accessible

6 Cloud Native Applications

Cloud Native is container based and elastic. AWS's biggest customer is **Netflix**. They roll out so many instances and microservices, and there is roughly 1 application change per second, meaning new features are coming out very frequently.

Microservices are the opposite of 'monolithic applications' and support organisational agility, such as rolling releases and hot swapping, without any down time. This is achieved through horizontal auto-scaling, design for failure, modular design (containerised etc.). APIs are also used, as well as automating things.

A *monolithic* application is built for one big block. Microservices aim to break this apart, where we have modules separated by API gateways.

The advantages of using this is that the services can be deployed to a subset of your users (called *canary* deployment). This allows testing for small groups of people before rolling them out to everyone. These changes can be tiny; changing the size or colour of a button to see if it changes the click-through rate. Another deployment method is called *blue/green*, where you have an example of your user base, and then you flip a switch and everyone has the new features instantly.

Distributed systems are harder to program, since remote calls are slow and always at risk of failure. There is also the issue of **eventual consistency**. Maintaining strong consistency is really difficult for a distributed system, meaning everyone has to manage this.

6.1 Observability

Once you have a massive cluster, how do you keep track of what's going on inside? Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

Logging is a very useful method for observation. Local log files are useful at first, but as containers are ephemeral (deleted upon termination), if an instance crashes, you won't be able to retrieve the log data. Additionally, the log could grow potentially unlimited. *Aggregated logs* are useful, and is what is used more commonly. It is a central server that records the log requests. Currently, a *side car* approach is used for logstashes, meaning they have access to the same mounted volumes etc, and are an aid to the program.

6.2 Metrics

Metrics are more than logs. This checks the CPU, load average, interrupts, etc. and is able to identify bottlenecks. This is *time sensitive*, so is stored in a time sensitive database. There is a plugin for Kubernetes that runs as a pod.

7 Serverless Systems

Firstly, it's important to know that serverless systems **still use servers!!** The essence of the serverless trend is the *absence* of the server concept during software development. This allows developers to focus more time on things that will *differentiate* their business from other businesses, rather than worry about the inner workings of a server. AWS defines serverless application as one that doesn't require you to provision or manage any servers. Unfortunately, this does lend itself to vendor lock-in.

Serverless systems have lots of possible usages, such as computation, data, messaging, user management and logging.

7.1 The Four Pillars of Serverless

1. No server management
2. Flexible scaling
3. No idle costs
4. High availability

When you think of serverless, the most common flavour is AWS Lambda. The Lambda flavour operates with **two sets of permissions**, one for the invocation, and one for the service (or resource).

7.2 Invocation

A Lambda function is invoked via event or schedule. Minimal Amazon Linux data is run containing the required runtime and function. Function provided with JSON invocation context and parameters. The function executes or times out. Result returned if defined. It is typically invoked in a few ms (also called a warm start) but can take around 1000ms sometimes (cold start). The container is retained for a short time to mitigate this latency.

7.3 Billing

Billing for Lambda systems depends on the amount of memory allocated and duration (for invocation). Therefore, the language used for scripting has a big impact on this. For example, Java would cost a lot due to its heavyweight libraries, while Python or Ruby would cost much less, being very lightweight.

7.4 The Four Stumbling Blocks of Serverless

1. Performance Limitations (cold start etc)
2. Monitoring and Logging (closed source, so difficult to expand existing tools)
3. Vendor lock-in — One of the worst forms of vendor lock in
4. Security and Privacy — All data written is encrypted by Lambda, but in multi-tenant cloud systems, there is a lot of suspicion.

8 Scalable Cloud Architectures

Software architecture is a set of structures needed to reason about a system. It includes software components, the relations between them and properties of both. The architecture may be implicit, meaning that it might not be written down, but every software has an architecture. If you build a significant system, you cannot not have an architecture.

Looking at scaling, we can either scale with pure cores, or use multiple servers (called vertical or horizontal scaling respectively). With **vertical scaling**, there will be no need to change your architecture, while **horizontal scaling** will be cheaper.

8.1 Scaling cube

The scale cube only looks at horizontal scaling. There are three dimensions to this cube:

- X axis: Horizontal duplication — Scale by cloning services and data
  - Each clone can do the work of all the other clones. Work is distributed among clones without bias
  - Inefficient compared to alternatives
  - But, very easy to do
- Y axis: Functional decomposition — Scale by splitting different things. Isolating and making scalable individual responsibility of components
  - Needs to be split in the code base
  - More costly than the x axis.
- Z axis: Data partitioning — Scale by splitting similar things. We partition the domains of incoming requests
  - Data partitioning split relative to the client
  - Improves fault tolerance and cache performance
  - More costly than the other two

8.2 Load balancers

Load balancers are used for:

- Distributing requests
- Managing availability
- Performing health checks
- Session affinity (sticky sessions)
- List of backend servers
- Load balancing policy
- SSL termination
- Alternative: DNS or router based

Load balancers can detect denial of service attacks and shut servers down rather than scaling the service up to keep up with the increased traffic (that could end up costing a hell of a lot of money).

Sticky sessions are really useful. If we use a sticky load balancer, we can always route a client's request to the same server instance, perhaps with an IP hash. This sticky session is really good because say you order things on amazon and have a shopping cart full, we want the next request to go to the server that has just been handling this shopping cart. Cookies are managed by load balancers or the application cookie. It enables session replication via shared session DBs or caches. Sometimes, the session is stored with the client, though this can be risky as the internals of the application can become exposed to the client.

There are a few load balancing algorithms, such as:

- Round robin — Simple even task distribution (ignores the difference in work)
- Weighted round robin
- Least connections

Load balancers allow high availability. If one load balancer has a high load, it can switch to a secondary load balancer. Kubernetes provides stateless LB via a service (round-robin by default). Additionally, they have sticky sessions.

8.3 Decoupling services

8.3.1 Message topics

Consumers can subscribe to different message topics, meaning that producers just send a message to a pub/sub (publish/subscribe), and then only the consumers that are subscribed to a topic would receive this message. This generates a disconnect between the producers and the subscribers but also means more scalability because the producers don't need to worry about each individual consumer.

Message queues are also used, and they are *asynchronous*, meaning it can be queued now and then run later, waiting until a consumer is ready to process. This decouples application logic and is also much more scalable, although it introduces latency into the system.

8.3.2 Service registries

Service registries resolve addresses for names and are based on HA, transactional data stores. For example, Apache.

These decoupling services allow for *event driven* algorithms.

8.4 Automation

You can't manually scale instances in your architecture, so you need to automate this. You'll need an elasticity controller that looks at metrics (CPU, mem, disk...). When exceeded, add more nodes, and when it isn't using much, can decommission some of these nodes.

Another way of judging whether or not we need to scale a system is to look at a job queue and scale depending on this.

8.5 Sharding

Sharding is the act of partitioning and storing a dataset in multiple databases. This might occur when a database gets really full, because replicating this data wouldn't solve the problem, or it might also hold some redundant data. This is particularly useful when having multiple caches. If we have multiple caches and each cache has the same data, we will have a lot of cache *misses*, so we are not getting the most bang for our buck.

8.6 Decentralised Hashing

Regular caches distribute evenly across buckets as a function of n. However, if n changes, all objects need redistribution. The solution to this is *consistent hashing*:

- Each peer received similar amounts of keys
- Little reshuffling on peers entering or leaving
- Map each object to a point on the edge of a circle
- Make a node hold keys for a range of consecutive keys
- If a node is removed, its interval is taken over by a neighbour node. All the remaining caches are unchanged.
- If a new node enters, keys are redistributed from a preceding node. All remaining nodes are unchanged.

8.7 Database scaling

Database scaling is one of the harder things to scale. One technique is **read-replicas**, but this is not good for writing. Another example is **sharding** (looked at above).

8.8 F1

F1 is Google's NoSQL database. It is built on Spanner and uses five replicas spread across the US. It is a rewrite of Google's advertising backend.

F1 chose to use Spanner over other solutions for several reasons:

- Spanner removes the need to manually reshard
- Spanner provides synchronous replication and automatic failover
- F1 requires strong transactional semantics and this is not practical in other NoSQL systems.

F1 has several design goals:

- **Scalability**: scale up, trivially and transparently, just by adding resources
- **Availability**: system must never go down
- **Consistency**: provide ACID transactions
- **Usability**: full SQL support, without using SQL

9 The Hadoop Ecosystem

Originally, there was Hadoop, which is Apache's answer to MapReduce. Of course, Hadoop needed something like GFS, which Apache dubbed HDFS. BigTable was needed, and this was called HBase. These three projects were known as 'The Hadoop Core'.

Writing MR jobs is pretty tricky, so some programmatic abstraction was required. This was called Apache Pig. Additionally, they needed an SQL-like query language, which they called Apache Hive.

Mahout is Apache's scalable machine learning platform, and runs on Hadoop/Spark. It provides tools for using Hadoop to create systems that perform clustering, classification, and collaborative filtering at high sped on large data sets.

Giraph is a graph database system used by Hadoop.

9.1 Hive

Hive is best suited to *data warehousing*. It operates on relatively static data where the response time is not critical. It has its own SQL-like query language called HiveQL/HQL, which is much more concise than lower level MR code.

Routes into Hive could either be a CLI, a web interface, or programmatically. Hive has a **driver** that compiles input, optimises computation and executes subjobs.

9.2 Pig

Pig is actually short for Pig Latin, and works on HDFS and NoSQL. Users describe how data is read/processed/stored in parallel. Dataflows could be linear or DAG (directed, acyclic graph). Interestingly, it has no if/for/while. Unlike SQL/HQL, Pig allows users to describe exactly how data will be processed.

9.3 Cascading

Similar to Hive and Pig, it is another layer on Hadoop, extending its use. It is still in development. Cascading lets you specify plans for queries. It provides a set of operators and tools for combining them into a DAG (as used in Pig). Each vertex in the DAG is an operator, and data flows from operator to operator. Therefore, it's very similar to Pig, except the graph can be displayed.

Once one job is finished, the next queued job is woken up, and then begins executing.