Tips and Tricks.

Creational Pattern – Object creation mechanism.
Structural Pattern – Structure of class or object.
Behavoiral Pattern – Communication & responsibility distribution.

Creational Pattern.

Factory, Abstract Factory, Builder, Prototype & Singleton FAB PS

FAB PS Fabulous PS5

Structural

Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy. ABCD FFP.

Behavoiral.

Strategy, Observer, Command. State, Template, Iterator, Mediator. Chain of Responsibility, Visitor, Interpretor.

SOC STIM CVI.

Social Stimulation with CVI.

Thumb Rule → Donot fail to deliver working system they will mention as time issue.
Make a linear way to build system.

System Design Number.

System up time - 99.99% (52 mins down in a year).

[1 Year = 360 days × 24 hours × 60 mins = 525,600 mins.
99.99% up time = 0.01% downtime.
0.01% of 520,000 = 52 mins.
99.999% uptime = 0.001% downtime = 5 mins.]

Load Balancer can be set anywhere between user & web server.
web server & application server.
application server and database.

There are many algorithm → LSR, Hash, Round Robin.
It can be a single point of failure as it will redirect all request. There can be a standy LB and active.

Cache and static can be done by CDN.
Cache data might be state.

Write Through
Data written to cache and db at same time. (Write-latency impacted).

Write Around.
Data bypass the cache and goes to db. (Read latency impacted).

Write Back.
Data written to cache and then in db. (Low latency but data loss).

Cache hit the size we need eviction policy.
LSR (least Recently Used). FIFO, Least Frequently Used (LFU).

No SQL
- key-value → Redis.
- document db → Mongo.
- wide-column store → Cassandra.
- graph db → Neo4j.

SQL (scale vertically mainly horizontally with Sharding).
No-sql (scaling horizontally).

SQL scale by partitioning.

Horizontal Partitioning (Sharding) → Divides rows of table in many db.
Vertical Partitioning → Separate feature or column into different db.
Directory Partitioning → Lookup service to abstract the partitioning schema.

Partition can be done using Consistent Hashing, Round Robin, List Partitioning, Composite.

System design is mainly about → latency, throughput and costs.

Example — A system with high-throughput ingestion service (best practices using cache) having latency spiking 500 ms.

Naive solution add more nodes.
Shard db.
Scale horizontally.

Telemetry revealed —
Distributed cache was the bottleneck.
Approx 50 network calls/requests.
Cache RTT dominates latency.

There is always network round trip calls. (Network means slow).

Cache is fast when the distance is 0.

1 CPU cycle = 0.3 nanosecond.

Easy example = 1 sec.
Latency heirarchy.
CPU register 1 sec.
L1 cache → 4 sec.
RAM 100 sec.
SSD 2-6 days.
Network RTT → 15 yrs.

Desk notebook = Cache.
Bookshelf → RAM.
Warehouse → Disk.
Another continent → Network.

await cache.get (key)
↳ 100 key then it will take long.

Data has distance. You cannot treat local variable and a db network call same way.

Mechanical Sympathy → Understand the physical constraints of the machine and design software that cooperates with them.

Hardware Reality

- Data is stored in blocks.
- Access has set up costs.
- Sequential access is predictable.
- Random access is expensive.

Random I/O.

- Seeks overhead.
- Cache misses.
- Latency spikes.
- Throughput collapse.

Sequential Vs Random I/O.

Hard Disk Drive
Spinning platters.
Moving read head.
Random I/O = physical movement.

SSD.
Pages.
Erase block.
Write amplification.
Still prefer sequential access.

DB are shaped in such way using disks. LSM tree used by NoSQL like Casandra and RocksDB. (They are fast as they put random writes into sequential writes.) They append the data at the end of the file.

(NoSQL is fast as it perform sequential write and its okay to tradeoff read complexity to write throughput).

Latency vs Bandwidth.

- Latency (Time to travel one request).
- Distance dominated.
- Hard to improve.
- Data will travel via cable or optic fibre Speed of light limited. We cannot improve without putting the server closer.

- Bandwidth (Number of requests per second).
- Pipe width.
- Parallelism.
- Easy to buy more.

Water pipeline → Latency is how long it took a drop of water from one point to another.

Throughput → diameter of the pipe.

How requests flow through the hardware.

Little's law. $L = \lambda W$. (L inflight requests, no. of requests in the system.
λ arrival rate, how many users are hitting the site per second.
w latency, average time it takes to process one requests).

If a service takes 1 sec to process a request and we get 100 requests per second then there will be 100 request sitting in your system all the time.

In case latency increase (db slow) then L (no. of request) must increase. The RAM has a limit and it can take fixed amount of requests. When request increase it will be in queue and wait time increase. (Positive feedback loop of death).

We dont only monitor CPU usage. We monitor λ and w. When λ increases and request stays same then system will fall.

Costs.

RAM = 100 × SSD.
SSD = 10 × Cold Storage.
Latency decreases as cost increases.

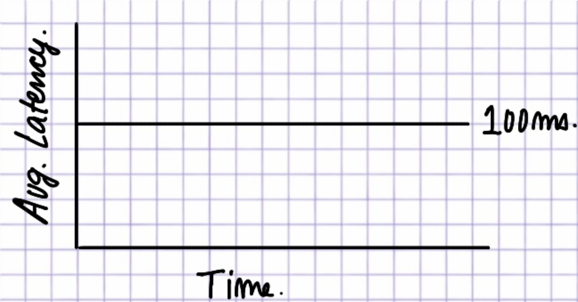
Mistake → Keeping everything hot is not safety. Its waste.

The main design is to align with the value of the data to the cost required to store it.

Data is hot meaning accessed 1000 times a second and belong to L1/L2 layers.

In design ask the question.
"What is the price/bit?"

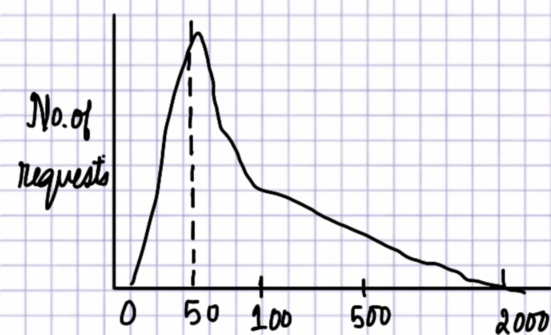
Points → Dont start with microservice. Start with latency. How far the data has to travel? Throughput. The state sequential or random. Cost?



Dashboard showing beautiful line of 100 ms latency.
It's telling story we are not listening.
The average is a great experience. ✓
The problem is 1 in 100 users were facing 15 sec delay

Total 10K users are facing issue.
1% of the audience are facing it.

The average is not correct measure.
It is percentile.



P(50) → median. The typical user experience.
It doesnot tell about the worst case.

P(99) ⟶ 99 th percentile represents the slowest 1% of the requests.

P(99) = 2 sec it means 1 out of 100 users waits atleast 2 sec.

(1 Billion user then 1 Million will face it).

Tail Latency Amplification.

When we are designing for P(99) then we are not fixing bug, we are engineering for the reality of high Concurrent system.

An application calls 100 microservices to render the home page (say product detail, user detail) and in case all microservice P(99) is 100 ms meaning that 1% of the request is slow.

Every single microservice has 99% chance of being fast then the probability of the home page request fast ⟶ Not at all 99%.

Probability of one service being fast = 0.99.
Probability of 100 service being fast = 0.99 × 0.99 × ...... 0.99 (100 times)

$= (0.99)^{100}$ (The exponent is the number of dependencies. The more services we add the worst value it will get).

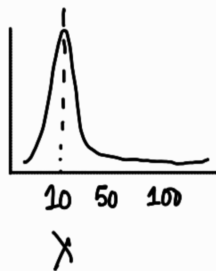
$= 0.36$

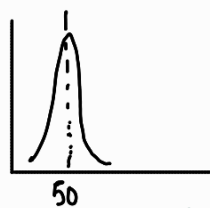
There is 36% chance user has fast experience.

At a big level focus on a system that is always 50 ms than a system usually 10 ms.

In a distributed system with 100s of call you are as fast as your slowest component.

Its called Tail Exploding.



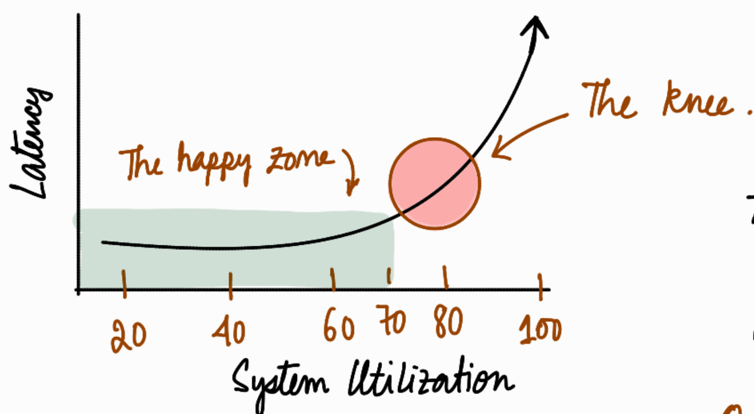


Always 50 ms better than usually 10 ms.

The target is to make always 10 moving towards left with no tail.

Latency → Time taken to move from Point A to Point B. (Ferrari single road, individual request).
Throughput → Number of requests send at a given time. (Bus aggregate requests moving as much data as possible).

They both become bitter when the load increases → The knee of the Curve.



Happy Zone (0-70%)

Adding more car in the road doesnot slow down.
Throughput increases and latency stays slow.

80% utilization.
A car tapping brakes causing jam.

Queuing delay → In case CPU hits 95% new requests needs to wait in the queue.

Little Law's show this wait time grows exponentially as you approach 100%.

The senior engineer secret.

Run the system 60-70% and 30% is the buffer that keeps P(99) from skyrocketing during load.

The Physics of Slowness — Amdahls' law, the hard limit.

Amdahls law → The speedup of a task is limited by its serial fraction.

Example 1000 workers to make the house they cant start the roof until the walls are up and the foundation is a serial bottleneck.

$$\text{Max speed up} = \frac{1}{S + \frac{(1-S)}{n}}$$

S = the fraction of the task that must be done serially.
n = the no. of processors you have.

Code is 95% parallelizable but 5% must be serial (eg a single db lock) even an infinite processors ($n = \infty$) the max possible speed is 20x.

$$\frac{1}{0.05} = 20.$$

Thumb Rule → No point of making faster System faster find the single choke point.

API slow → Optimise the code → CPU become fast. → The code hitting disk faster.
↓
The cache added (to limit disk hit).
← Disk is fast. API still slow the cache is so fast that it is saturating the network bandwidth of the network card.
← Upgrade to a 10 GBPS network card.
← API slow as it reached the limit of lock contention in kernel.

The fix is moving the bottleneck to somewhere else where it is cheaper and easier to solve.

The 4 Golden Signals.

Latency.

The time to service a request.
P(50), P(95), P(99) look separately.

Errors.

The rate of requests that fails.
Always measure your error rate relative to the traffic.

Error stays flat and traffic drops – the error percentage is skyrocketing. It means the system is so broken that user could not able to reach it to trigger error.

Traffic

How much demand is on the system.
(λ) lambda in Little Law.

A measure of how much work your system is being asked to.

$L = \lambda W$

Average no. of items in systems = arrival rate * average time in system.

Saturation.

How "full" the most constrained resource — CPU, RAM, disk, DB.
The most imp leading indicator of future problem.

* Saturation is not only CPU and RAM. It can be no. of available thread in thread pool, no. of open file descriptors, depth of message queue.

CPU ulitisation increasing from 50 to 80% latency will stay good. By the time latency will give any alert user are already in pain.
Latency is a lagging indicator.
Saturation is a leading indicator.

The fuel gauge analogy. → We dont wait for the plane engine (latency high) to stop to realise we are out of gas. We look at fuel gauge (saturation).

We are 85% saturation on our database connection pool. Based on the current growth we will hit 100% in 2 weeks. In case we do then P(99) latency will increase by 500% and cause a cascading failure. We need to increase the pool size or add read replica.

The Hedge Requests.

A request sends and say slow for P(99) then it will wait for 10 ms and no response then send the same identical request to replica.

The probability of all of the request slow = P(A slow) * P(B slow)
= 0·01 * 0·01 = 0·0001.

1 in 10,000 requests. 100x improvement.

Average lie →
Distributions. →
Tail Amplification →
System Queues (Bottleneck) →
Golden Signal →
Hedged Requests.

Go to projects monitoring dashboard.
Find mean latency and P(99) latency.

Calculate P99 to mean ratio. More than 10x?
If so you have a significant variance problem that is likely causing hidden user pain.

In case the traffic doubles next day what is the physical resource that will cause P(99) to explode.

The cost of communication.

```
public UserProfile getUserProfile (String userId)
{
    User user = db.fetchUser ( userId);
    Preferences pref = cache.get ("pref" + userId);
    return new UserProfile (user, pref);
}
```

→ Launching a signal, packaging of electron and photons sending via copper wire or fibre optics cable, hitting switch, router, going through firewall, entering another machines kernel and finally reaching another application.

L1 cache hit — 0.5 ns (1 min)
RAM access — 100 ns (3 min).
Network call — 0.5 ms (11 days).

↳ The local call abstraction hide the time complexity.
Anything that human takes 11 days then we will not do that 50 times/day.
Batch and try to reduce work.

What makes the network call expensive?

↳ Speed of the light (fixed)
All OS and protocol demand time each step.

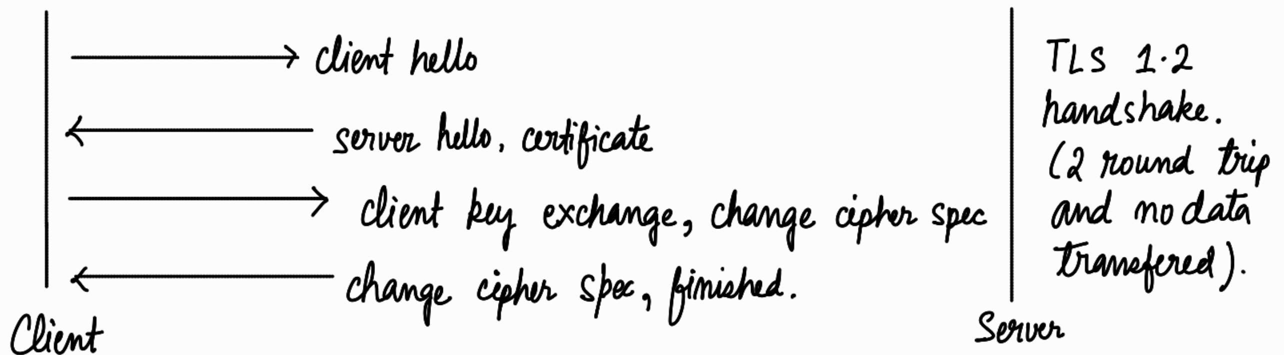
Sending a packet we need IP address.

computer doesnot know the IP address.
see the local cache ✓ good
✗ send UDP packets to the DNS server ✗ if not then another server
DNS resolution take 1 ms - 10ms pre work.

Connection will use TCP and TLS.

TCP is stateful protocol but the wire it is connected is stateless.
The user send SYN and server send SYN-ACK and you send SYN.
RTT to the server and no data send it is connection established.
(50 ms)

After the TCP will do TLS (HTTP) connection handshake.
It will ask for certificate, cipher spec.



CPU performed the RSA and verify the certificate. (Compute time tax on top of network call).
It took 3x of the speed of light to say hello.

The kernel tax context switching.

Application code lives in user space and NIC (Network Interface Card) is managed by kernels. → Data coming in wire kernel receives it.
It copy the data from kernel memory space to applications memory space.

SYSCALL → CPU has to stop running the code, switch to kernel mode move the bytes and switch back.

Reliability (Retry and congestion).
↳ any time line fails then retry and the request max 200 ms because of timeout.

The Serialization Crisis →

JSON and all send object and network understand bytes. (Not good for fast process).
It has been JSON for long. The CPU has to read the character and text parsing to get in binary.
New string for the key, new object for value.

Protobuf / Binary → {"id" : 120}
In Protobuf we dont send id and string.
Will set id to a assigned field say 1 and send the data.
CPU will only copy the message.
gRPC build on Protobuf.

In JSON, 30% CPU budget is used in Parsing string.

The global speed.

Latency has huge portion on the speed of light. (light 300K km/sec
Optic fibre 200k km/sec).

Server in London and user in San Francisco the distance 8.5k km. Round trip 17k km.
Light speed 85 ms time. (cant be changed).

The TCP and TLS will take more time.

TCP trip (85 ms) — TLS Trip 170 ms — HTTP Trip 85 ms (Total 340 ms)
(3 trip) HTTP 1.1 and TLS 1.2 (2 trip) (1 trip)

Solution → QUIC HTTP 3.

TCP where connection is very less and long lived. (TCP and TLS together).
QUIC combined the connection and handshake into one.
0-RTT for repeat connection.

Combined handshake (85 ms) → HTTP Trip (85 ms)
2 trip Total 170 ms.

CDN moves handshake closer to the user.
Putting server closer so the 85 ms time will be 5 ms.
The edge server stays warm and keep a long lived connections to main server.

Discord maintains millions of concurrent users. The "gateway" service is the entry point for the phone and desktop.

Service A calling B calling C → the time will be high.
gRPC (Protobuf and HTTP2/Multiplexing)

Previously to send 50 update to another service then 50 connection (kernel task) Or send one by one.

HTTP2 allows to send all request with one connections.
The connection pooling.

Apache Arrow.
Massive dataset in the db.
Pulling the data into the Python based model.

- Serialise the data into the format like CSV or JSON.
- Send it over the wire.
- The Python side has to parse it and turn it to a numpy array.

Apache Arrow changed the game by defining a standard memory layout.
The data on disk, the data on the wire and the data on the RAM are identical.

Zero Copy deserialization. → When bytes arrive in the NIC it directly copies it in the applications memory.

The Rule ⟶ Batch (Dont make 1000s insert call. Instead make one connection and update).

Data locality (2 microservice network call multiple time then put together).

Coarse grained API (Not chunky API make chatty).
Assemble everything to show in the home page directly and dont load separately).

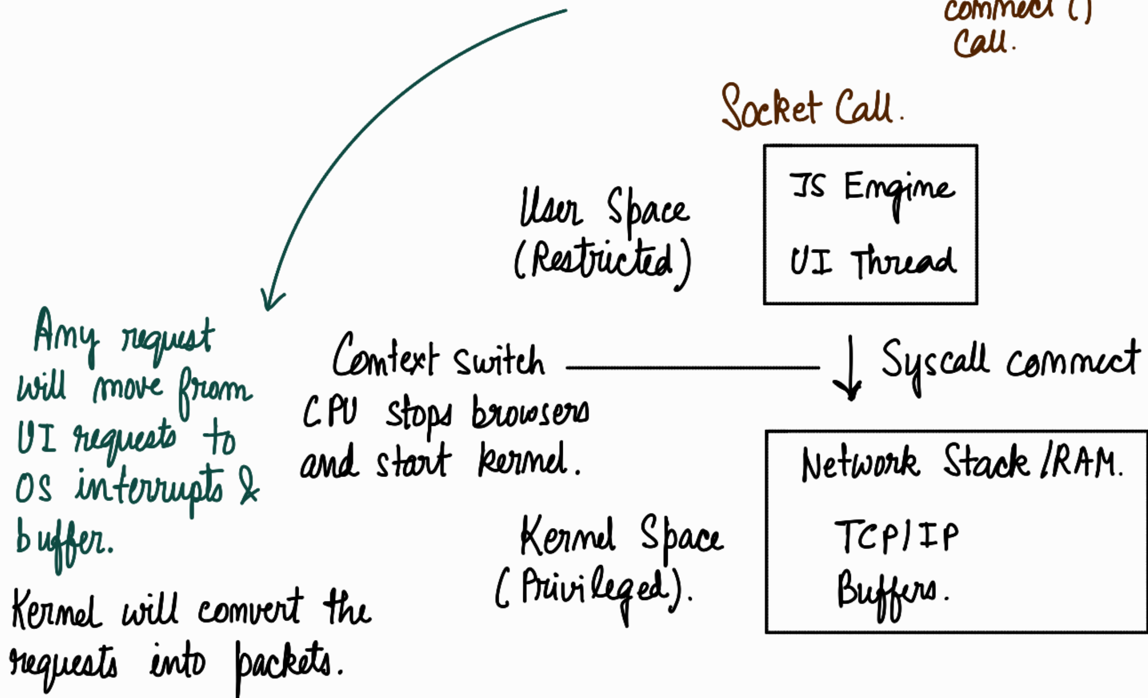
The Anatomy of a Request

https://api.netflix.com/v1/movies. ⟶ OS intervention

string stored in memory of browser.
UTF-8 bytes

Browser cant talk
to wifi or fibre cable.

It happens via Syscall. Socket Call. then connect() Call.



Network has MTU (Maximum Transmission Unit).
1500 bytes MTU limit.
(Historical artifact in Ethernet).

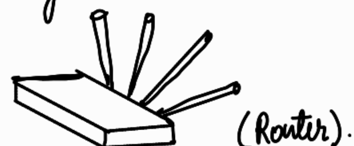
To transmit 10,000 bytes of message kernel will make 7 packets. The kernel will add wrapper on 1500 bytes.

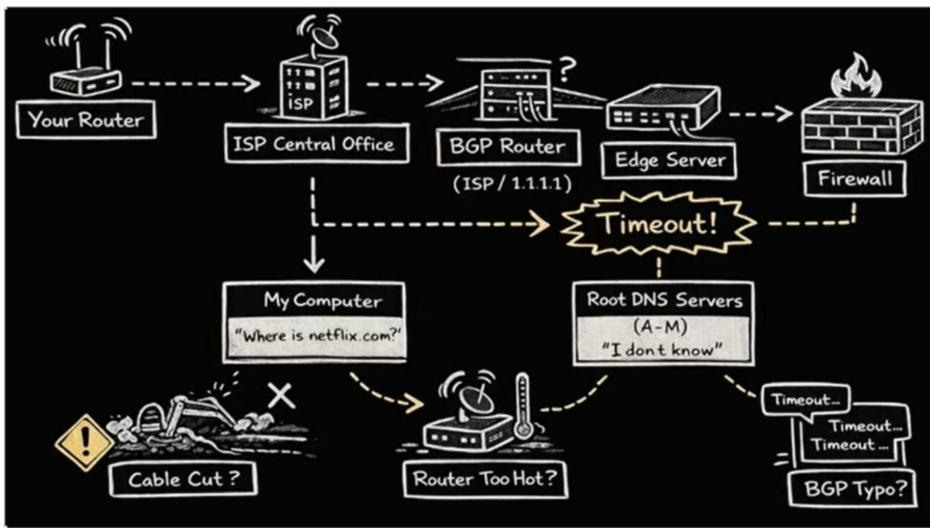
- Kernel will add TCP header in the requests (20 Bytes).
- IP header (20 Bytes source (laptop) & destination).
- Ethernet Header (MAC address of Wifi router). (14 bytes).

Total of 50 bytes on header.

Now comes the physical layer.

- Kernel made the packets. It sends a signal to NIC (Network Interface Card).
- In wifi the NIC takes in binary and converts in vibration of 2.5 GHz or 5GHz
- The request is moving electromagnetically bouncing in walls and objects
- The wifi router is a gateway and listening calculates the waves and convert back to electrical pulses.





- Packets – The string url to 1500 bytes binary message vibrating in the air.
- String name of website and the DNS will give destination.
- Hit the name it will go to ISP or cloudflare recursive resolver to get IP. Then to root server DNS.
- It will tell the server like TLD Server manages .com url.
- .com TLD server will ask Netflix (authority to get IP).

Protocol UDP Port 53.

DNS uses UDP Protocol.
- TCP needs 3 way handshake.
- UDP fast but doesnot gaurantee any return.
  There is no return then it waits, issue in P99.
  (DNS latency causes issue in P99).

- TTL set across all IP address. The OS, Network, DNS all having same IP address. DNS will directly give IP and not search in record.
- TTL set to 24 hours means any change in the IP address of Netflix will not reflect to the user and when set to 6sec then lot of latency.

TTL time to live → the memory of internet. (It cached the old IP).
Say netflix change the URL. It will be visible when the TTL time get done.

Consistency vs Availability tradeoffs.

In global setup we use Geo-DNS.

-The authoritative server looks the users IP address (Say Japan IP)

- it will redirect to Tokyo.

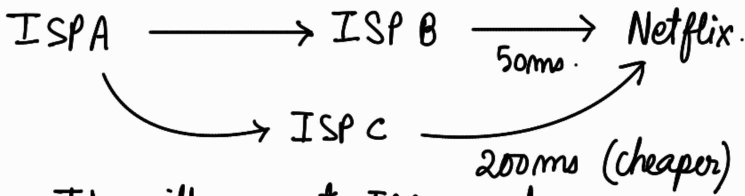
-DNS becomes the first load balancer of the system.

The packet now has the IP address.

↳ Hits the router → enters the internet.
  How the packet knows which physical cable to hit to get the data center in Virginia?

↳ There is no Google maps in cable.
  The internet is a collection of 70k independent island called autonomous system.
  Comcast is an AS.
  To move the packet from one island to another use BGP (Border Gateway Protocol).

ISP A connects to Netflix by ISP B in 50ms.



It will connect ISPC as cheaper.
There is no control in public internet.

Any ISP provider advertises that they have the best route to google and global connection tries to connect then it fails.

↳ Google, Cloudflare solves the issue using Anycast.

- In Anycast they gave the same IP address to 100 different data center.
- BGP look and see the IP address and see the loc say London then it will redirect to London.

1 2 3 4 5 6
✓
3 4 5 6 1 (2)
l ↑ r

5 > 3 (l → mid + 1)

0 1 2 3 4 5
3 4 5 6 1 2
↑ ↑ ↑ ↑
l l r

mid ! > l
r = mid - 1.

airbnb mcp server
↳ can use the server written
Mcp server Rest API.
Chat → Text (It will connect to LLM)
↓ (LLM)
Restful API
Claud desktop → Chat connection → airbnb
Mcp Server
Facebook API
Google api
PRACTICE

Hello
Sure
Hello
Hello
SUCHISMITA
DEB
MATU.
DEB.
BANKOK.

MCP → FastMCP. → Setting: dev setting → config file.

Claud Desktop. (MCP Host)
↓ ↓ internal
one config file MCP Client
in json can be added as MCP server.
add as many

MCP Host → MCP Client (LLM agent) (routing). → MCP Server
→ MCP Server.
↓
Eat

Centralization (Control user for unauthorised user) (gateway)

RBAC
Role Based Access Control.

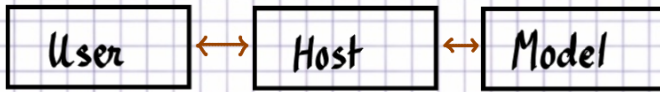
MCP Server
MCP server via LLM model.

1 2 3
2 5 1 2 (3)
↑

MCP → Protocol.

User Request
MCP Host
(chat engine).

The Physics of Persistence

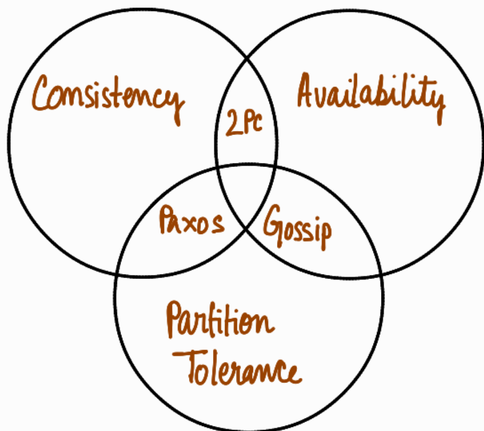


User interact with host (claud desktop any chat engine).

The host then talk to underlying LLM model

Example Host → ChatGPT
Model → GPT4.0, GPT 5.

CAP Theorem and PACELC.



CAP → These are not features. These are promises system makes under pressure.

Consistency	Availability	Partition Tolerance.
Every read sees the most recent writes.	Every requests gets a response.	System works even if msgs are delayed.
linearizability. one brain one truth.	No timeouts. The system never says no.	The network will fail.

CP system.
- Locking in distributed system
- Consensus algorithm like Raft or Paxos.

CP system.

- In the CP system a write is not successful until majority nodes agreed. Its called Quorum.

Say total 5 nodes then 3 needs to be success before you say success.

AP side

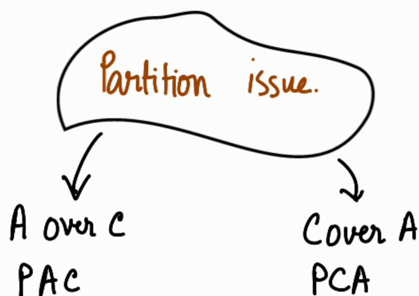
Mostly there will be no network issue.
In case of network you really dont have much to do.
The point is pick AP or CP.

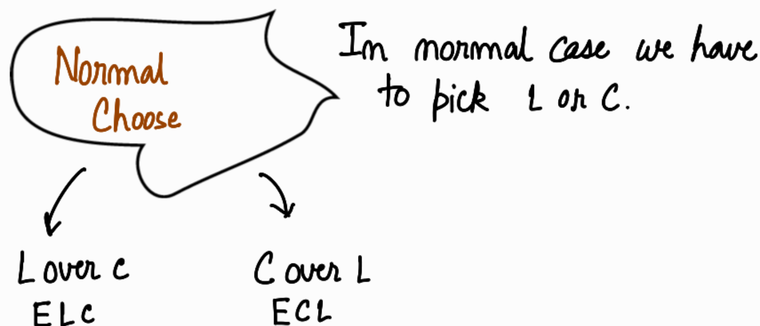
PACELC (pass-elk).

Partition → Availability vs Consistency.
Else → Latency vs Consistency.

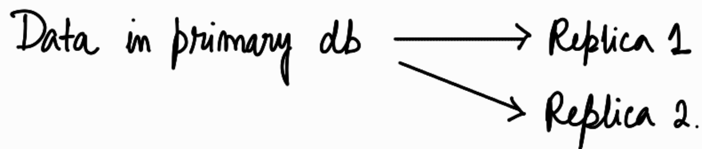
→ Else meaning when Partition good then how do you trade off Latency and Consistency.

CAP says what happens during P. But Partition is not usual.





Normal Case - Trade offs.



Synchronous (C over L).

Primary write on both replica then get success.

Postgress (Sync commit).
PC → In Partition issue.
EC → Normal (slow but correct).

Asynchronous (L over C)

Primary saves data locally then eventually writes in replicas.

DynamoDB and Cassandra.
AP → In Partition issue.
EL → Normal case.

Consistency is a spectrum. (Strong Eventual Casual).

Strong Consistency (Linearizability).

- The target.
- Single Postgress node or distributed raft system.
- Expensive Slow Never wrong.

Eventual Consistency (Least consistency)

- When there is no new updates eventually

1 Million Requests per seconds.

AWS → IAM manages 400 million requests per seconds.

CPU Utilization.

$$\text{Core Utilization} = \frac{\text{Total time} - \text{Total idle time}}{\text{Total Time}} \times 100.$$

$$\text{CPU Utilization} = \frac{\text{Core Utilization } \#1 + \#2 + \#n}{\text{Total no. of cores.}}$$

Get the no. of core you got in CPU.

wmic cpu get NumberOfCores, NumberOfLogicalProcessors.

NumberOfCores	NumberOfLogicalProcessors.
6	12.

Repo Name → agile 8118/ node-1m-rps.

Any code.

```
while (true) { }
```

Single Thread → Task Manager we can see System and User and CPU percentage.

↳ Each thread can utilise one CPU core at any given point of time.
In the code it will work on single thread and it will perform one task and any additional task will wait for it to execute.
To make it happen we need to spin up another task.

Multithreading Code.

```
public static void main (String args[])
{   for (int i=0; i<6 ; i++)  → # Spawn 6 system-level thread.
                                  CPU has 6 thread.
    {   Thread work = new Thread (() → {
                    while (true) { }   # Run in each thread.
                  });                     Busy waiting
        work.start();
    }
}
```

Repo Name → agile8118/node-1m-rps.

```
@GetMapping("/hello")
    public String hello()
    { return "hello"; }
```

In Postman time = 12 ms.

In node (npm i -g autocannon) here we can simulate sending 1000s of requests.

autocannon -m GET -c 20 -d 20 -p 2 "http:localhost:3001/simple".

- -c = connection.
- -d = duration. (20 sec)
- -p = pipeline. (URL).
- -m = method GET.

```
(joseph) → node-1m-rps autocannon -m GET -c 20 -d 20 -p 2 "http://localhost:3001/simple"
Running 20s test @ http://localhost:3001/simple
20 connections with 2 pipelining factor
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1 ms	2 ms	3 ms	3 ms	1.72 ms	0.65 ms	28 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	16,055	16,055	18,127	18,559	18,000.6	516.28	16,048
Bytes/Sec	4.03 MB	4.03 MB	4.55 MB	4.66 MB	4.52 MB	130 kB	4.03 MB

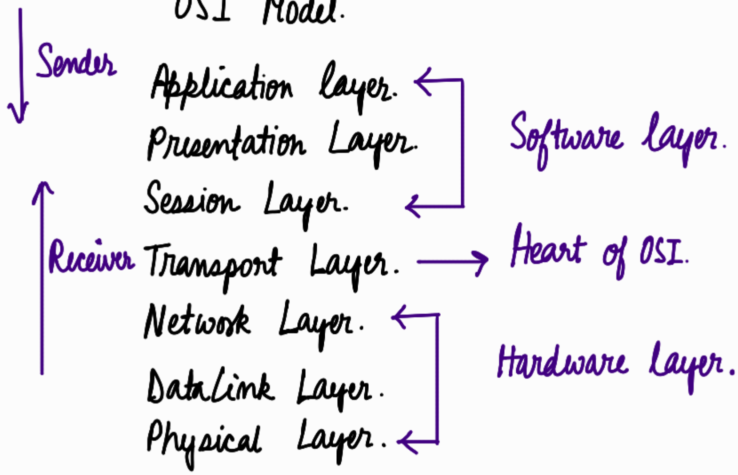
```
Req/Bytes counts sampled once per second.
# of samples: 20

360k requests in 20.01s, 90.4 MB read
```

A computer specification 12 CPU core 32GB RAM.
10GB/s Network Performance (1.25 GB/sec)

When sending more record then there will be network failure.

(34:25) ✓.

Network connects Compute, Storage, DB, load Balancer.



Networking Pillar.

Core Fundamental.
Network Infrastructure.
VPC
Containers and distributions.
Security and Performance.