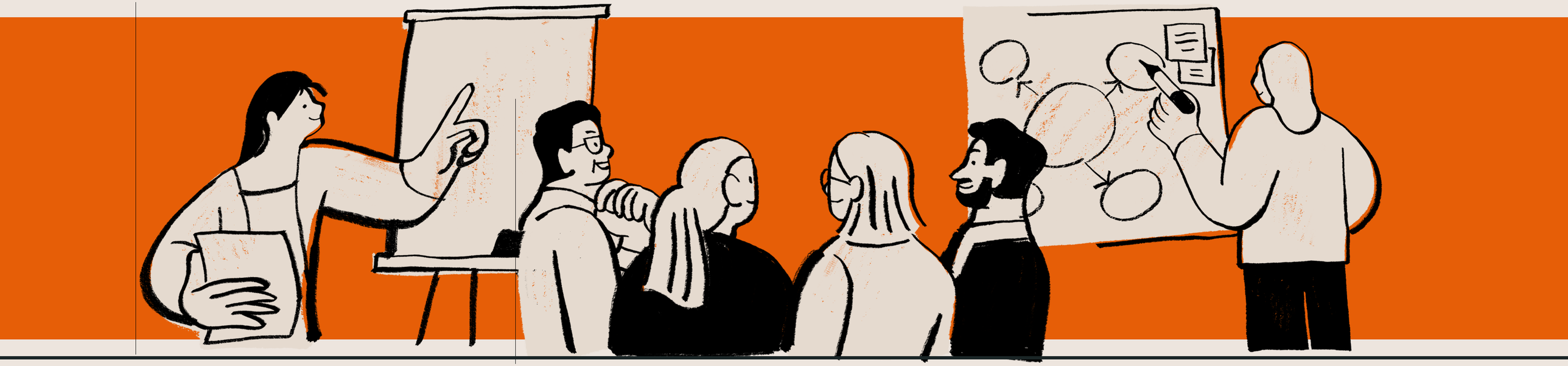
DESIGN FOR MILLIONS

Creating services that serve millions of concurrent users without breaking a sweat

Mebin Thattil · mail@mebin.in



CONTENTS

- Objectives to achieve
- A simple story to grasp the concept
- A case study on Hotstar
- How generic video streaming is done

OBJECTIVE

The primary objectives while building such services are to ensure:

- **Scalability**
- **Compatability**
- **Reliability**
- **Observability**
- **Performant**
- **Reduced Costs**
- **Failover mechanisms**

SCALABILITY : WHAT?

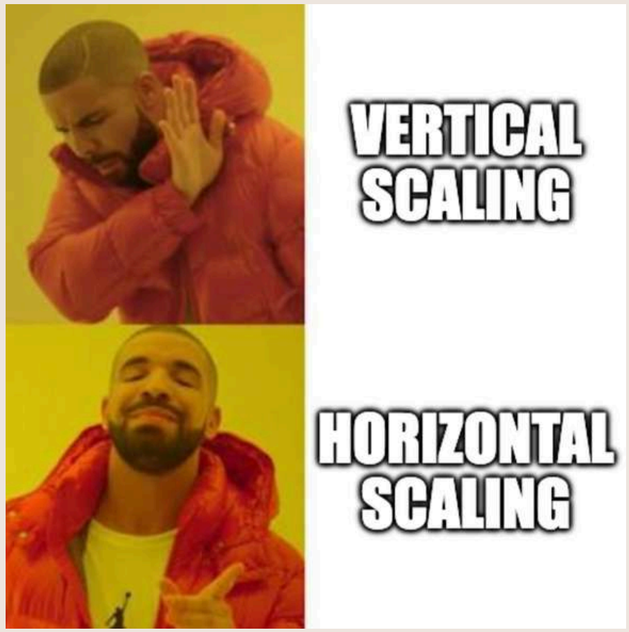
Simply put, how well your system can handle *growth*

Growth could mean:

- more users
- more requests per second (RPS)
- more data
- more random traffic spikes
- more features / services

SCALABILITY : TYPES

- Number of users suddenly grow, how do I “scale”
  - Vertical Scaling - buff up the same machine
  - Horizontal Scaling - add more machines



COMPATABILITY: WHAT

- Internet runs on protocols or mutual agreements
- Protocols evolve - some are better than others in specific cases
- Older devices might not support newer protocols
- Dropping support for those protocols means **service disruption not degradation**
- HLS, AV1 codec support etc

COMPATABILITY: WHAT

Why not stick to older protocols then?

Because using newer protocols / codecs help with:

- minimize cost - lower bandwidth used (higher compression)
- higher performance
- more throughput
- more secure
- less efficient - harder to scale

RELIABILITY: WHAT?

- Keep systems running at all times
- Prevent failures
- Have redundancy
- Heath checks

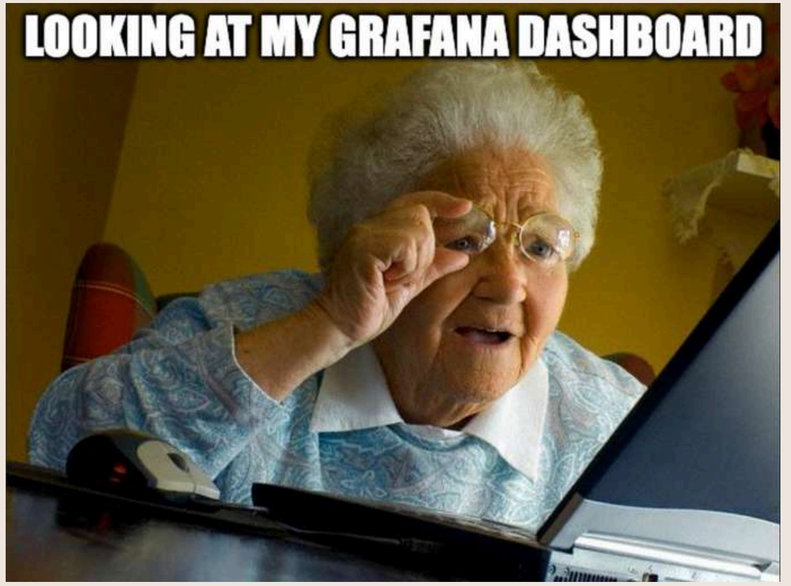
OBSERVABILITY: WHAT

- How do I know what's happening to my system at any given point in time?
- Check external outputs - logs, metrics, traces

- Helpful to find bottle necks while stress testing.
- "If you can't measure it, you can't improve it."

OBSERVABILITY: HOW

- Lot of cool tools available, including open source ones:
- LGTM Stack:
  - Loki: log aggregator
  - Grafana: dashboard
  - Tempo: tracing especially for microservices
  - Mimir: storage



I have monitoring set up on my RPI : *grafana.mebin.in* and *prometheus.mebin.in*

Grafana playgrounds: https://play.grafana.org

FAILOVER MECHANISM

- When your system fails, how do you ensure:
  - core services are still up
  - end user experience is degraded rather than disrupted
  - system can auto-recover

NOTE

Handling more load can be done in two ways:

- make existing system more efficient - better algorithms
- add more systems and distribute load

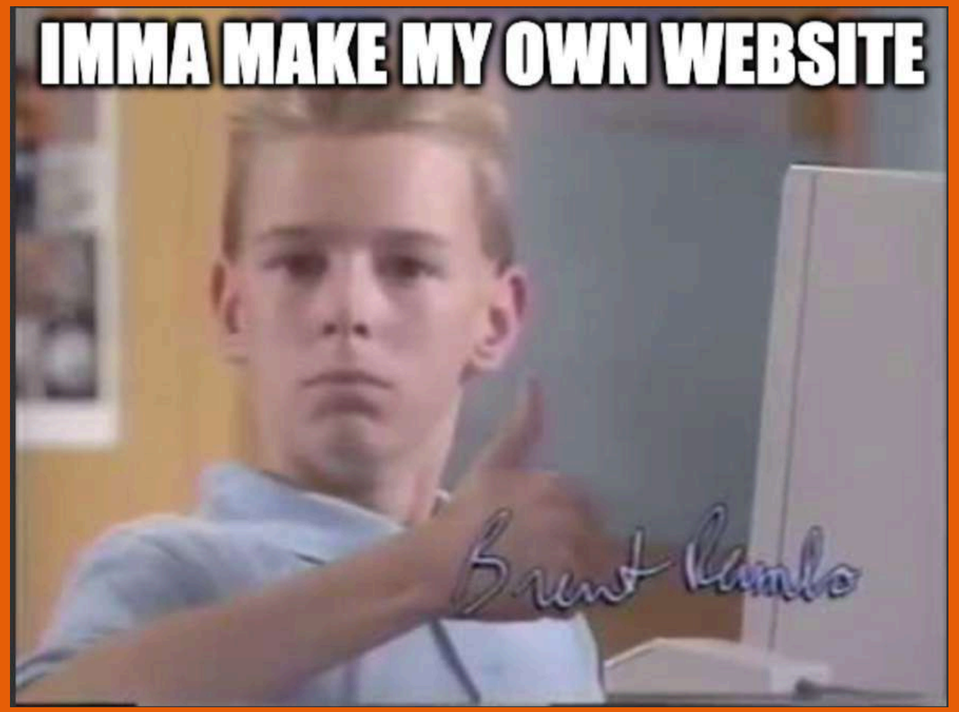
The first part can't take you very far - you hit limits [*eg. you can't sort a list in O(1)*].
The latter part can help scale if done right [*you could add more nodes that serve traffic for your amazing list sorting app*]

For this talk we are going to be focusing on the latter.

BOB'S STORY

Bob wants to build a website

Bob wants to publish blogs



Disclaimer: Next few slides have certain topics that are over-simplified to simplify this talk and may not be representative of the real world

BOB'S STORY

Bob rents out an EC2 instance - a T2 micro

Works well, but then his blogs went viral.
His instance could not handle the load!

Let's say the system's memory is full

BOB'S STORY

Bob then decides to upgrade his instance.

Let's say the upgrade pricing looks like this:

Ram Upgrade:

2GB → 4 GB : $1

4GB → 8GB : $5

Bob chooses option 1 [vertical scaling]

BOB'S STORY

Bob's blogs become EVEN more viral!

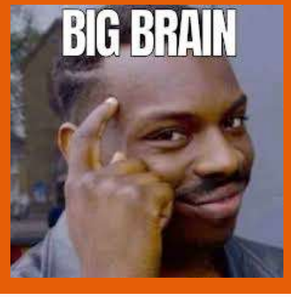
Pricing for reference:

Ram Upgrade:

2GB → 4 GB : $1

4GB → 8GB : $5

Bob is now about to choose option 2, then he has a realization: It's cheaper if he gets 2 machines with 4GB [horizontal scaling]

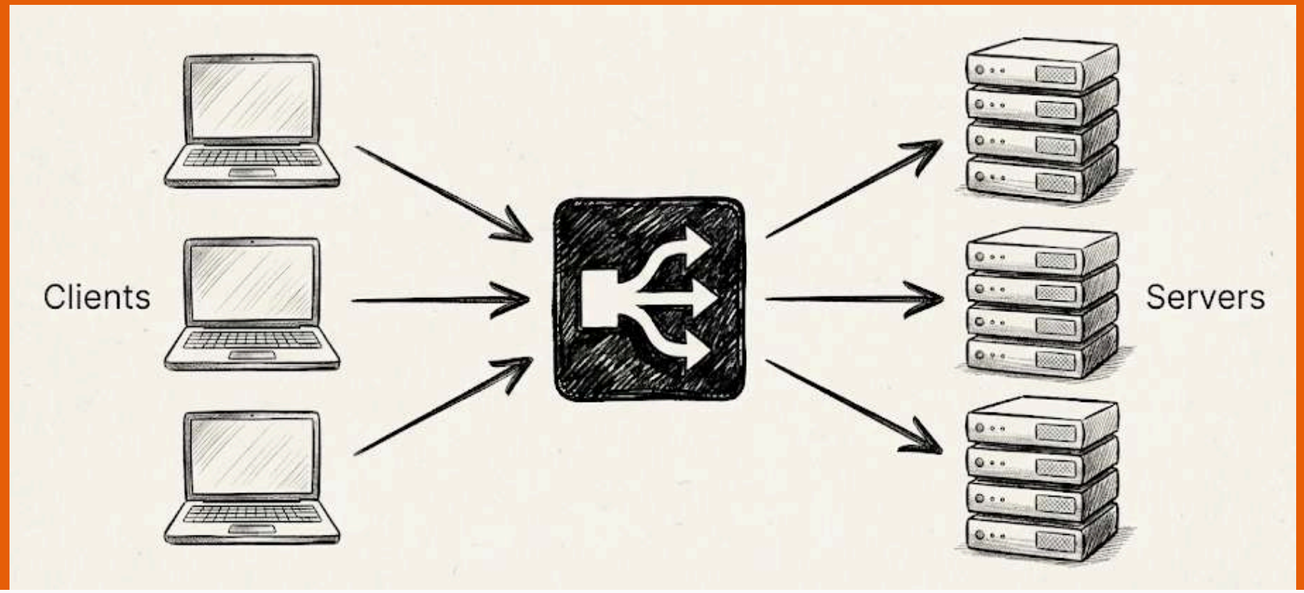


BOB'S STORY

Now Bob has two systems, how does he split the load?

Answer: Load balancer.

Bob uses a load balancer that has health checks and distributes traffic accordingly.

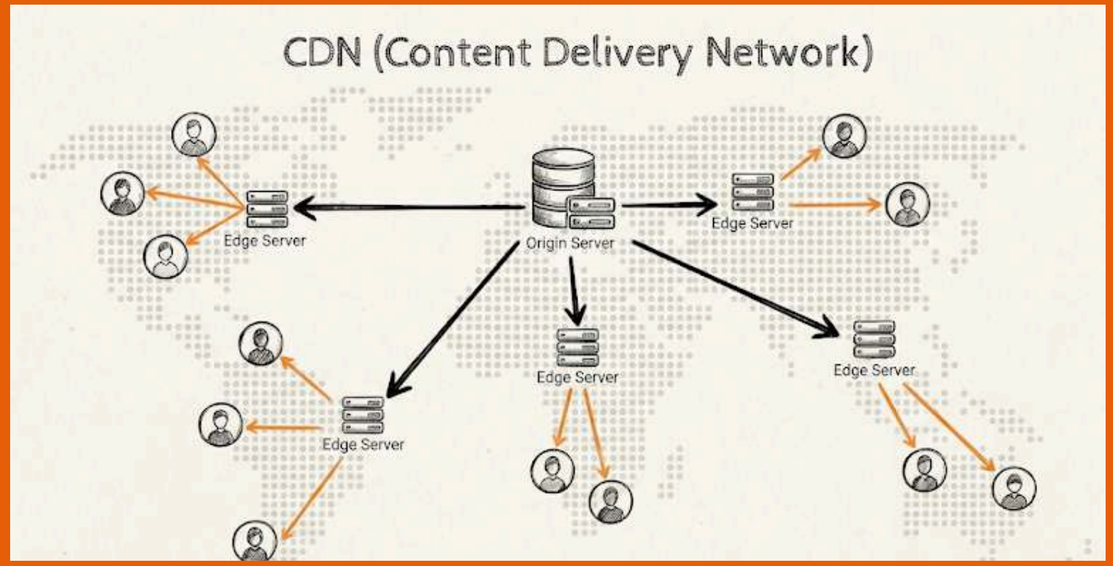


BOB'S STORY

Next hurdle: Bob has a friend in the US who complains his site is very slow

Reason: Request travels a long way to get served

Solution: CDN - Content Delivery Network



BOB'S STORY

Bob now want's to live stream his gaming sessions on his website
He quickly faces a lot of issues:

- **His servers can't take the load**
- **Video does not work on a friends who has a 12 year old phone**
- **The live stream crashes when more than 10 people try to view it**
- **He has no option to view what went wrong**
- **His system crashes and even the critical parts [blogs] are now down**

BOB'S STORY

Bob now want's to live stream his gaming sessions on his website
He quickly faces a lot of issues:

- **His servers can't take the load → SCALABILITY**
- **Video does not work on a friends who has a 12 year old phone → COMPATABILITY**
- **The live stream crashes when more than 10 people try to view it → RELIABILITY**
- **He has no option to view what went wrong → OBSERVABILITY**
- **His system crashes and even the critical parts [blogs] are now down → FAILOVER MECHANISMS**

Bob is clueless so looks at how Hotstar does it

RECAP

Recap, from Bob's story we learnt about:

Scaling services

Load Balancing

Content Delivery Networks

Now let's move on to the case study...

CASE STUDY

THE IMPRESSIVE ENGINEERING BEHIND JIOHOTSTAR

Mebin Thattil · mebin.in

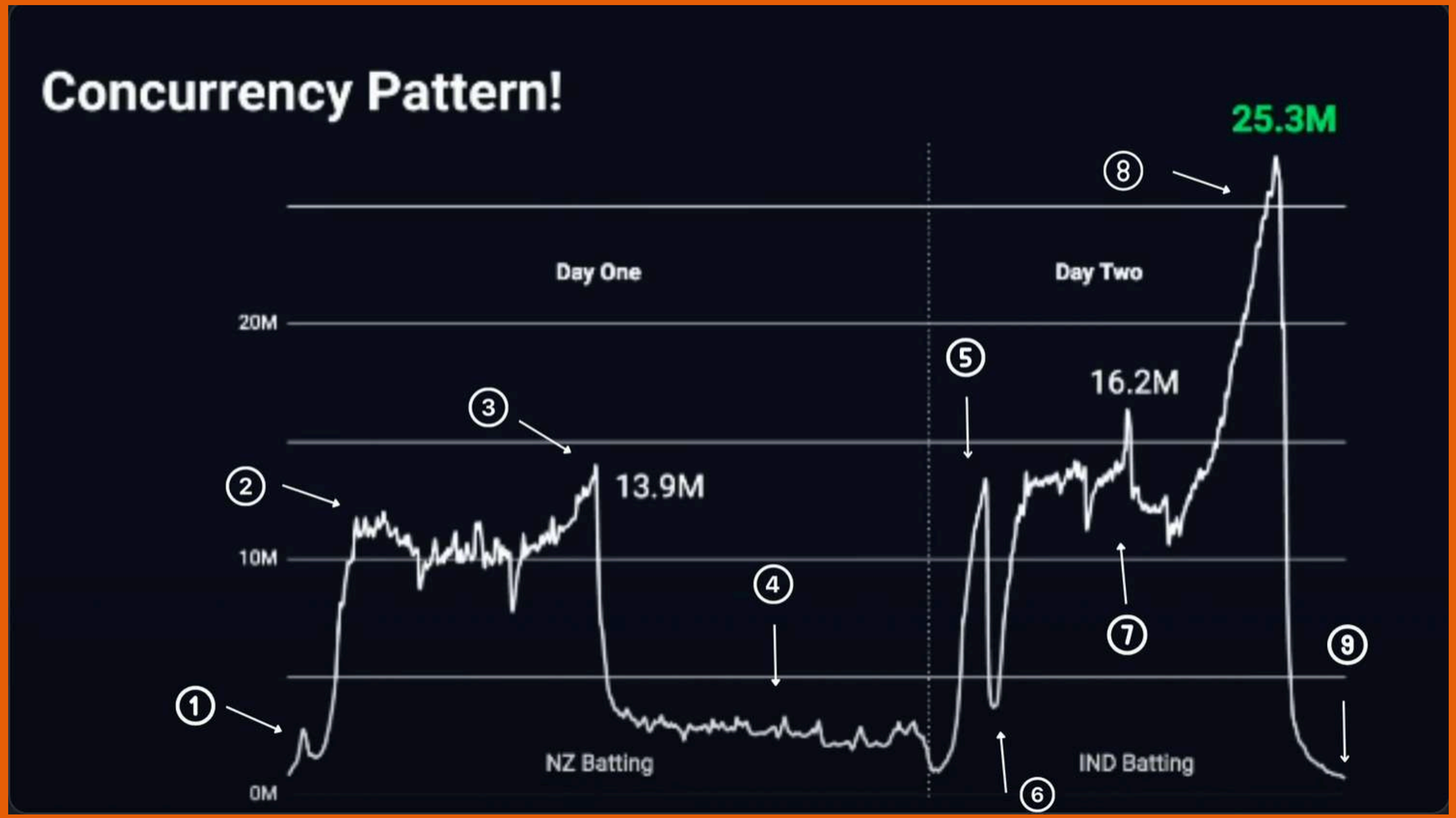
ENGINEERING MARVEL

Talk by Hotstar engineer

ENGINEERING MARVEL

No one had come *close*

CONCURRENCY PATTERN



1 → Toss
2 → Match Starts
3 → Rain
4→3M people watching the rain
5 → Match continue
6 → Innings over
7 → India is batting
8 → Guess
9 → Match ends

THE DROPS

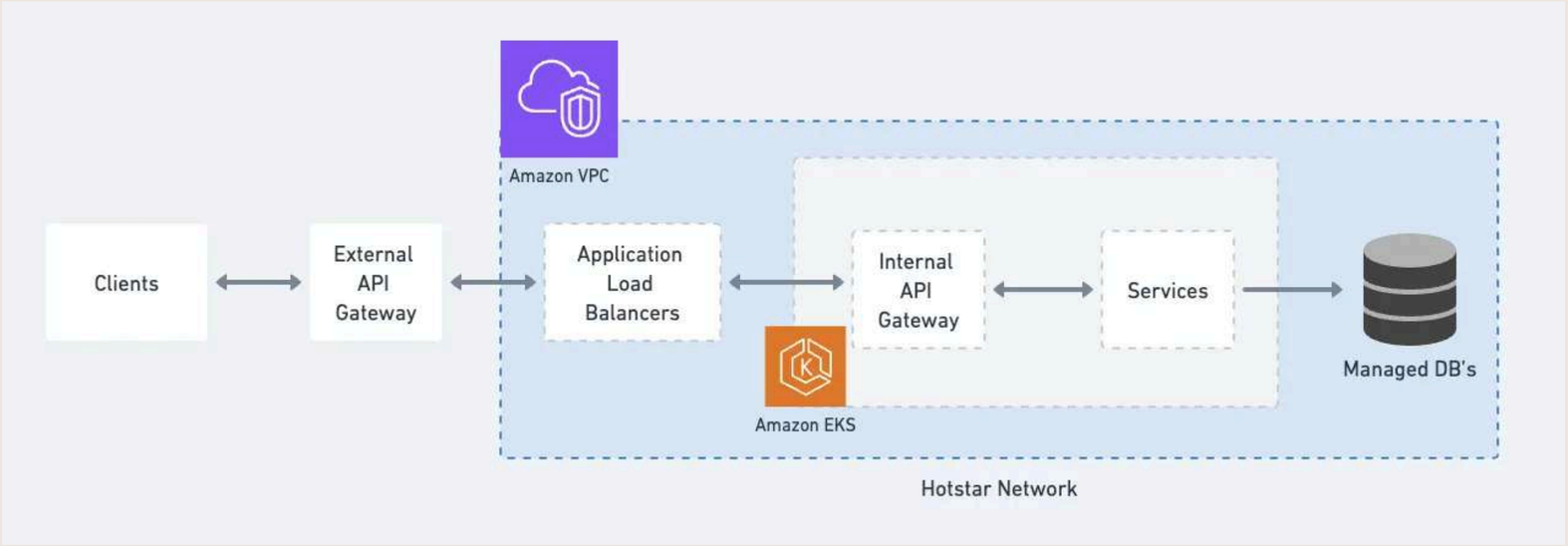
When match resumed after rain viewership went from 3M → 15M

When Dhoni was out viewership fell from 25M → 1M

Handling these spikes is the hardest part

BACKGROUND

- Multiple clients (Android, iOS, Web, Smart TVs, etc)
- Client apps make calls to external API gateway (CDN - Akamai)
- This runs security checks and executes routing rules to fwd req to internal API gateway
- Internal API gateway sits behind a fleet of application Load Balancers (LB)



BACKGROUND

- Internal API gateway fwd req to relevant backend
- Backend - managed self-hosted DB solutions. (managed and *self hosted possibly for performance tuning*)

HITTING LIMITS

- External API Gateway scales by deploying cluster of edge and mid-gress nodes
- Each of these nodes act as gateway proxies and perform:
  - security checks
  - rate controls
  - request unpacking, etc
- This puts stress on compute capacity and limits the overall throughput you can push through the external gateway.

PROFILING

- Now try to calculate if this throughput is enough for the target volume we are trying to achieve

- With current setup , scaling CDNs to that volume was neither cost not time-effective

Then came the breakthrough question - Must we treat all requests the same way?

SEGREGATE

- Segregated APIs:
  - cacheable (scorecard, concurrency metric, key moments, etc)
  - non-cacheable
- Have leaner but optimal security and rate controls for cacheable content
- Helps reduce stress on compute capacity

- Created new CDN domain
  - configuration rules for cachable paths
  - better isolation from core services
  - prevented side effects from any misconfigs if any

SPREAD AND SCALE

That's what she said

- Focusing on two parts of a typical production cloud environment infrastructure involves:
    a. NAT Gateways → networking
    b. Kubernetes clusters → container orchestration
- All work together to make the service work
- To effectively scale → need to understand limitations of each component.

A. NAT GATEWAYS

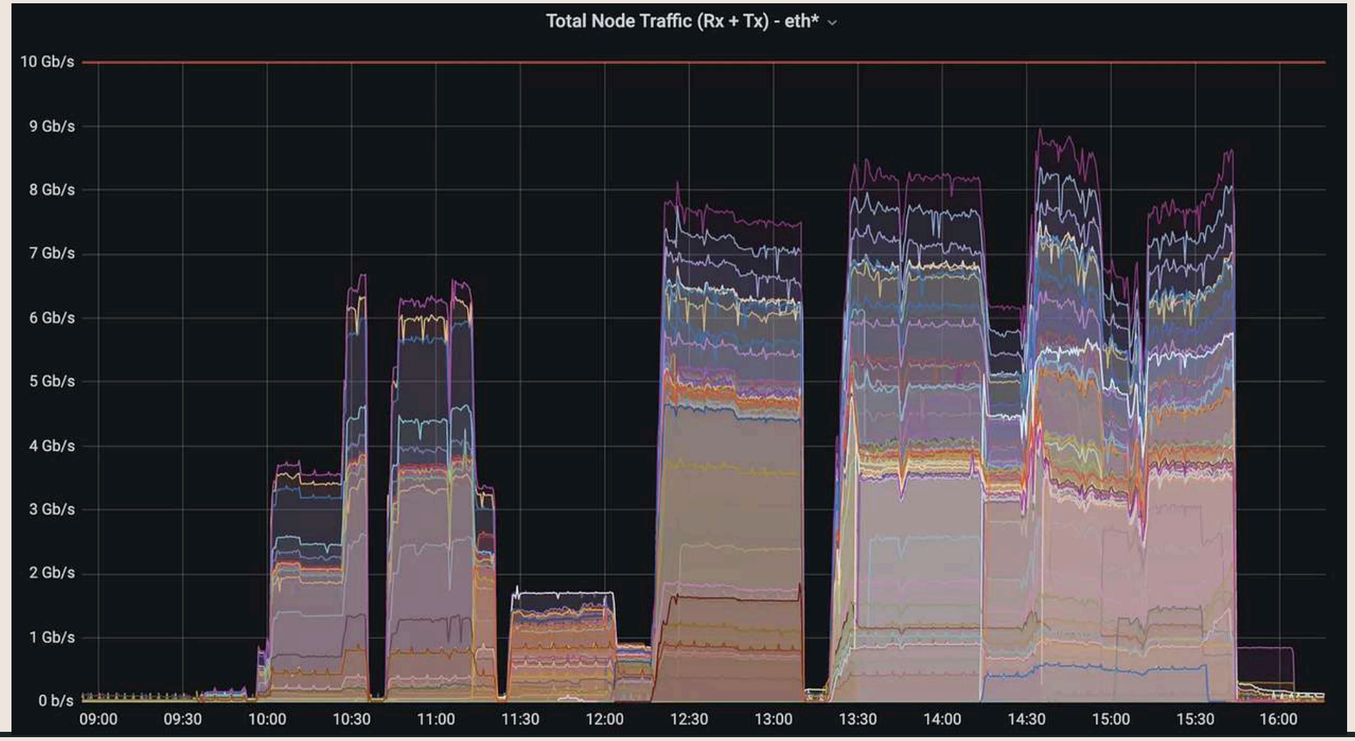
- After gathering data they realized just 1 K8s at 1/10th peak traffic load was using 50% of network throughput capacity
- Need to scale out

- Usually we have 1 NAT Gateway per AWS Availability Zone (AZ)
- All subnets usually route external traffic through that gateway
- This setup was becoming a bottleneck → huge external traffic

Solution: provision NAT Gateways at the subnet level instead of AZ level

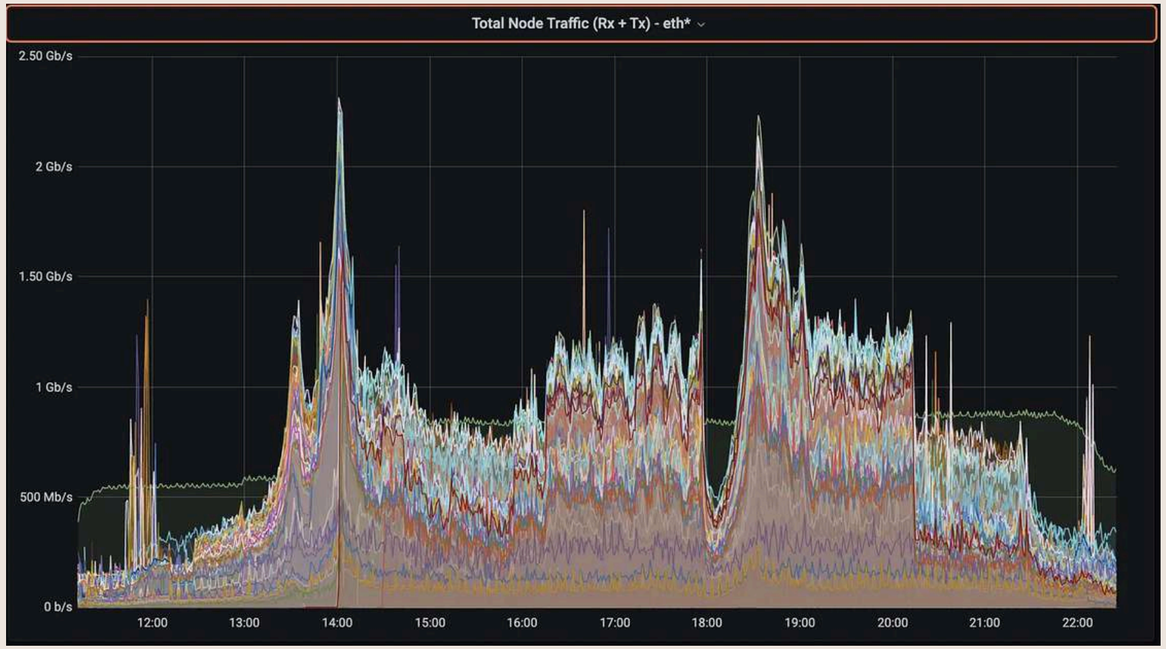
B. KUBERNETES

- Now analysis of network throughput at the Kubernetes worker node level
- Some services consumed very high bandwidth (8-9 Gbps)
- Nodes running multiple internal API Gateway pods simultaneously could be optimized
- Need to flatten the spikes



SOLUTION

- Deployed high-throughput nodes (min 10 Gbps)
- Topology spread constraints to API Gateway → Each node hosted only a single pod (spread out more evenly now)
- This kept throughput per node at 2-3 Gbps even during peak load!



MORE K8S

- Initially had 2 K8s clusters, but that was not enough to support 50M concurrent users
- Need some way to automatically scale this
- Tried Amazon Elastic Kubernetes Service (EKS)
- EKS takes care of the control plane, so Hotstar can focus on data plane

- *Control plane → controls data plane*
- *Data plane → where the processing happens*

OH NO, I CANT SCALE

- Ahead of the 2023 World Cup, the team faced a production incident
- Application needed 400 K8s nodes, but application could not scale beyond 350.
- The issue? They ran out of IP Addresses!!

OH NO, I CANT SCALE

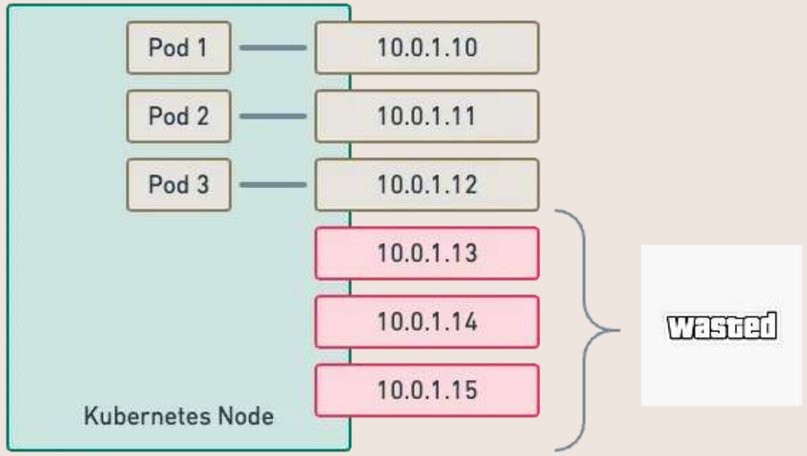
- Previous setup:
  - 3 subnets across multiple AZs
  - each with /20 CIDR blocks (~4090 IPs per subnet)
  - Total 3 X 4090 = 12.3K IP addresses available

Wait, so if we only need 400 nodes and we have 12.3K IPs how did we run out?

- Each node requires more than 1 IP address, since each node can have multiple pods.
- AWS needs to reserve these IPs ahead of time (*faster pod startup, propagation takes time*)

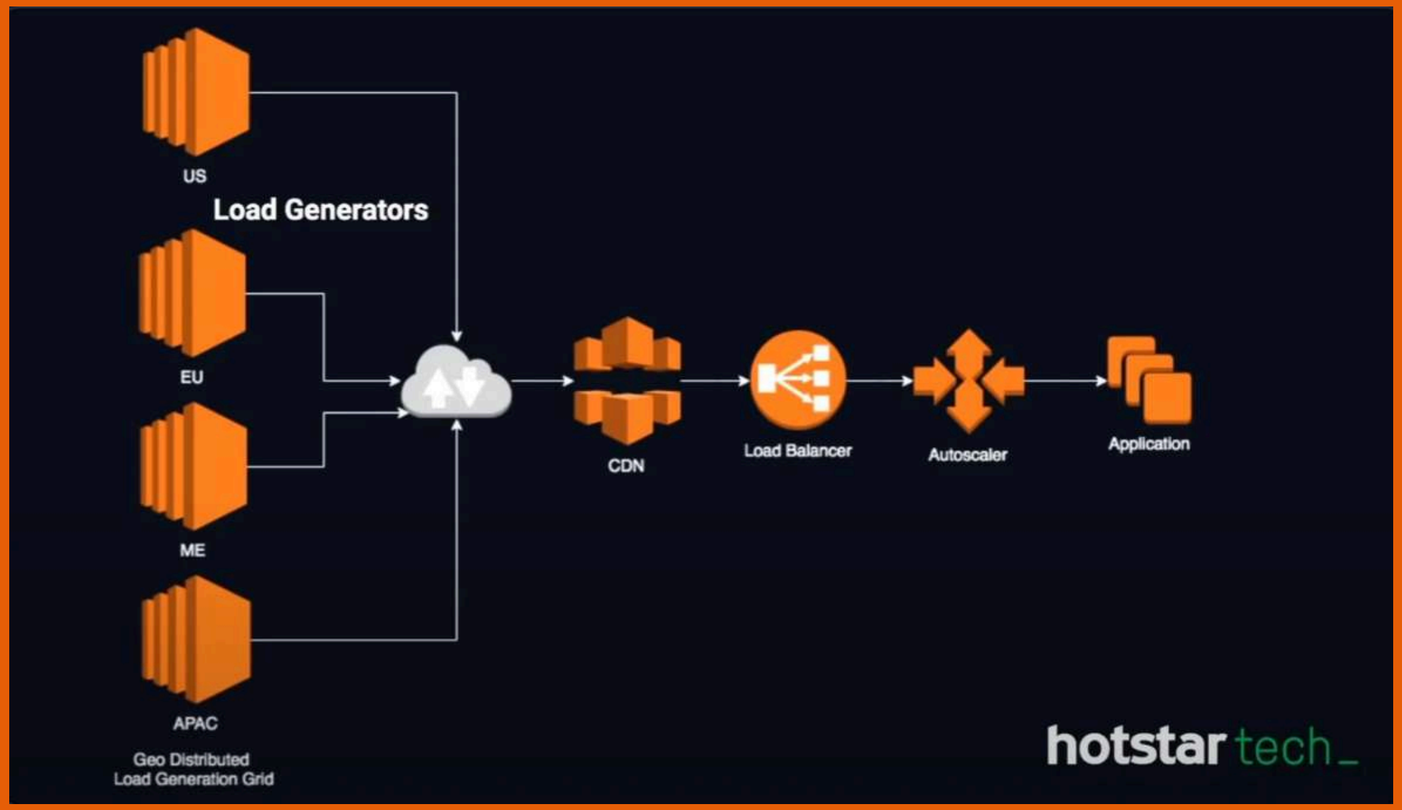
FIX

- Modify the VPC Container Network Interface (CNI) and modify:
  - MINIMUM_IP_TARGET: from default of 35 → 20
  - WARM_IP_TARGET: from default 10 → 5
- Added new subnets with larger /19 CIDR blocks (~48K IP addresses)



- MINIMUM_IP_TARGET: max no. IP add. allocated to each node
- WARM_IP_TARGET: how many additional IP add. pre-allocated for scaling

PROJECT HULK



- Large scale testing needs to be done prior to event.
- Project HULK is an in-house load testing service
- c59 xlarge machines · 8 regions
- 108,000 CPUs · 216 TB RAM
- 200 Gbps out

PROACTIVE SCALING

- **Constraints**
  - 1M users per minute
  - application boot time : 1 min
  - EC2 provisioning and become healthy under a LB: 5-6 min
- By this time 5-6M users already added and we can’t handle load, hence proactive scaling is needed.
- Also have a buffer
- Reactive scaling simply can’t keep up with at this scale.

NO AUTOSCALING

- **Hotstar does not use AWS Autoscaling, instead uses in house to prevent:**
  - Insufficient Capacity Errors
  - Single Instance Type per auto-scaling group
  - Step size
  - retries and exponential backoffs (cannot assign in an AZ) → creates a large queue
- **In house scales based on RPS & RPM instead of CPU / RAM metrics**
- **Ladders: for x Mil users → have y number of servers**
- **Secondary scaling groups → use more powerful machines if unavailable**
- **Using SPOT instances**

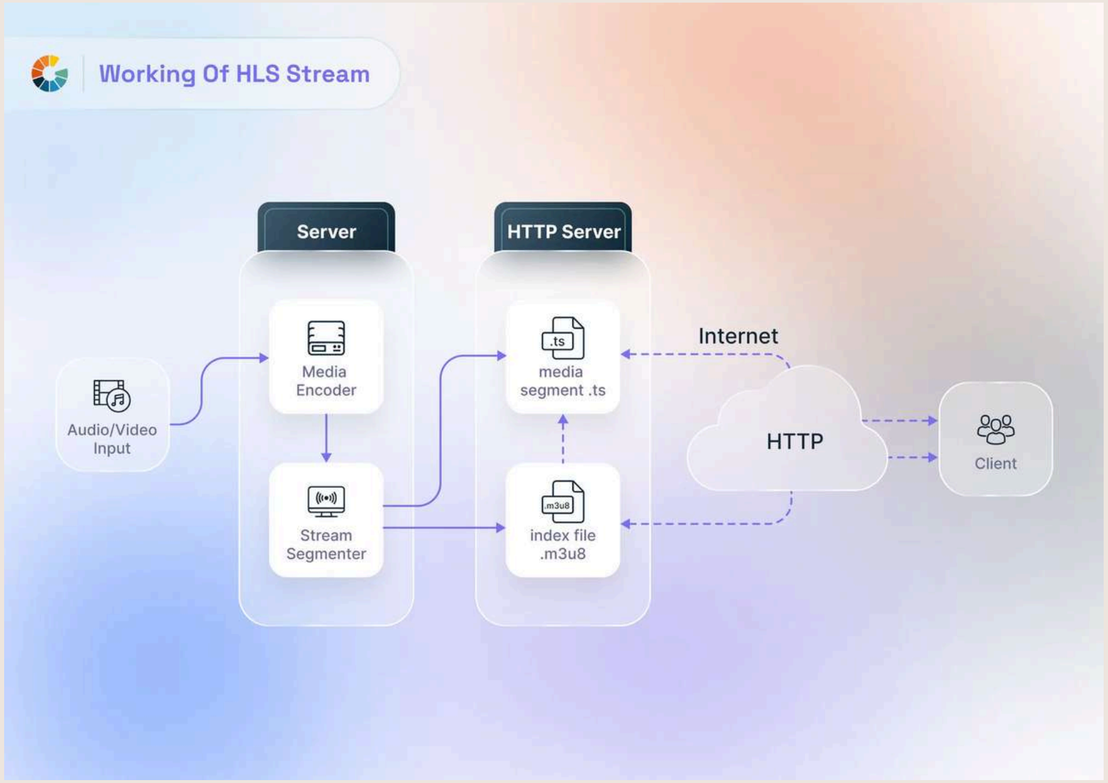
PANIC MODE

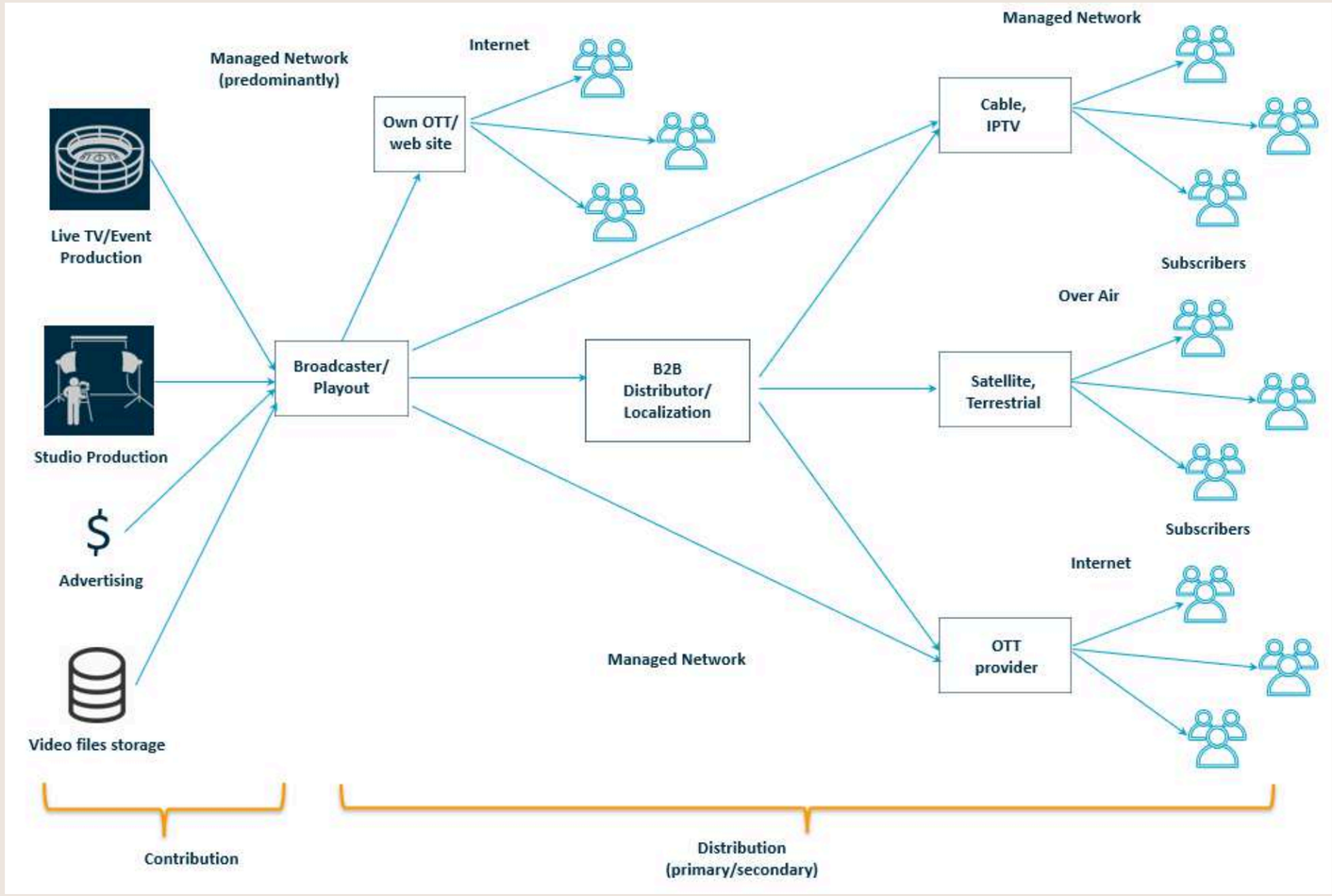
- Graceful degradation
- P-0 services must always be up
- Turn off non critical services
- Allow bypass to P-0 services if necessary

TYPICAL VIDEO STREAMING

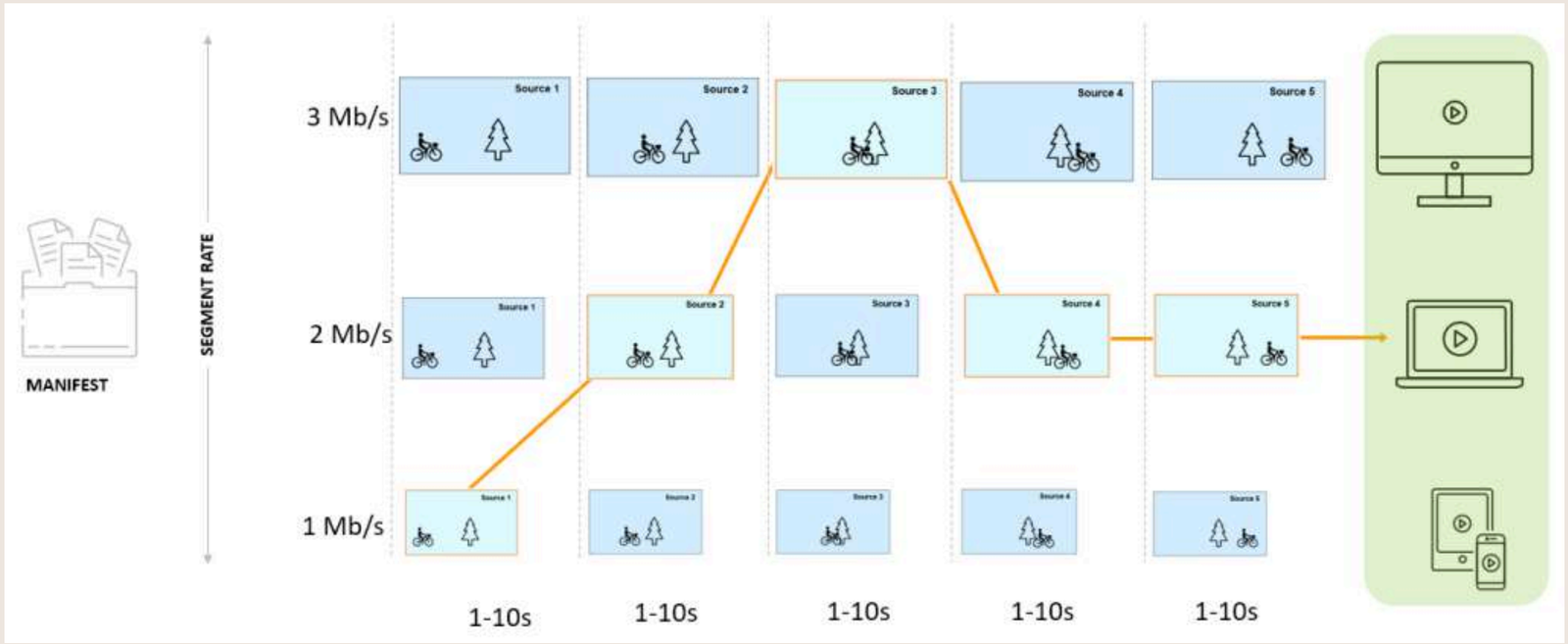
- The typical video streaming pipeline looks like:
  - capture input
  - media encoding
  - transcoding
  - segmentation
  - creating manifests and var bit rate files
  - send .ts and .m3u8 files to user over HTTP [HLS]
  - client player uses these assets to stitch and play the video
- Two parts [Transport Protocol & Encoding Protocol]
  - Transport: HLS, RTMP
  - Codecs: H.264(AVC), H.265(HEVC), VP9, AV1

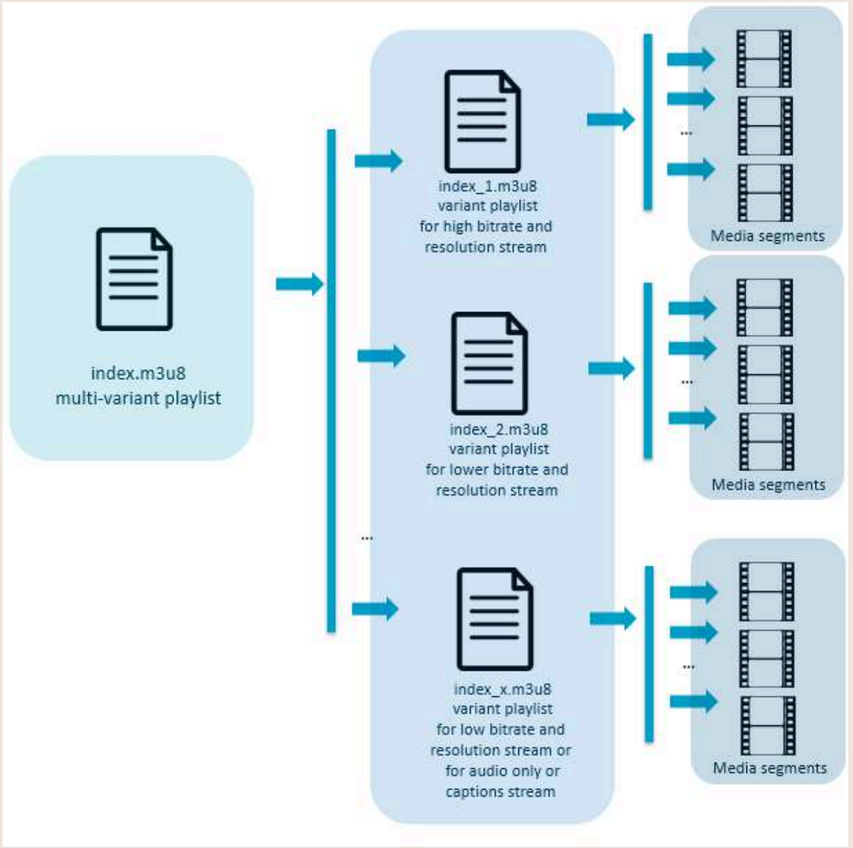
TYPICAL VIDEO STREAMING





ABR ENCODER


MANIFEST
SEGMENT RATE
3 Mb/s
2 Mb/s
1 Mb/s
Source 1
Source 2
Source 3
Source 4
Source 5
1-10s
1-10s
1-10s
1-10s
1-10s


index.m3u8
multi-variant playlist
index_1.m3u8
variant playlist
for high bitrate and
resolution stream
index_2.m3u8
variant playlist
for lower bitrate and
resolution stream
index_x.m3u8
variant playlist
for low bitrate and
resolution stream or
for audio only or
captions stream
Media segments
Media segments
Media segments

CONCLUSION

Lot of times the engineering behind such systems go unnoticed

Core principles:

- Plan for scale from day 1
- Measure & Observe everything
- Design for failure
- Optimize costs and improve efficiency
- Test at twice the target scale
- Segregate by requirements
- Automate

SMALL GAME

https://pshenok.github.io/server-survival/

THANK YOU

HOPE YOU LIKED IT!

Get in touch: https://mebin.in
mail@mebin.in