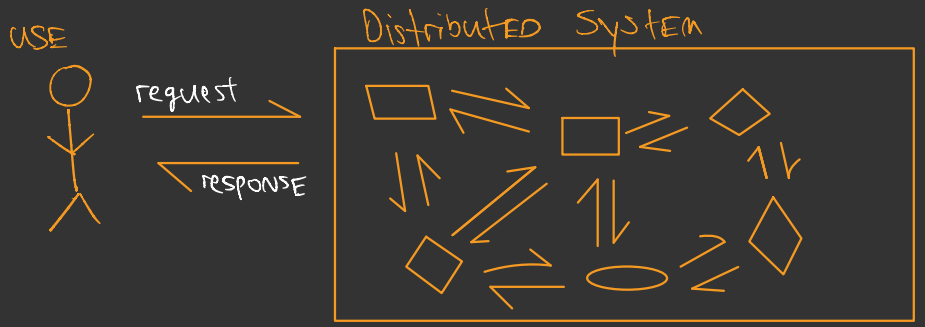
System Design

Ivan S

2022

Distributed Systems

- Always keep it simple
- A lot to manage, use when NEED to only
- group of computers working together, complexity hidden from users; feel like interacting w/ single comp.



Fallacies of Distributed Systems

Network is reliable
Latency is Zero
Bandwidth is infinite
Topology doesn't change
network is secure
only one administrator
Transport cost is zero

- Lots of things can go wrong, need to handle them somehow
  - Client can't find server
  - Server crash mid request
  - Server response is lost
  - Client crashes

Characteristics

- No shared clock → leads to clock drift
- No shared memory
- Shared resources
- Concurrency and Consistency

Communication

- Different parts of Distributed System need to be able to talk
- Requires agreed upon format or protocol

Benifits

- More reliable, fault tolerant
- Scalability
- Lower latency, increased performance
- Cost effective

Performance Metrics

Scalability

- Ability of a system to grow and manage increased traffic
- Increased volume of data or requests

Achieve growth w/ out risk to performance

Reliability

- Probability a system will fail during a period of time
- Slightly harder to define than hardware reliability

Keep working even in hardware or soft fails > automated testing, tooling to predict & compensate hardware failures

Measure of MTBF

$$\frac{\text{Total elapsed time} - \text{total downtime}}{\#\text{ of failures}}$$

24hr - 4hr / 4 failures = 5hr MTBF

Availability

- Amount of time a system is operational during a period of time
- Poorly designed software requiring downtime for updates is less available

most important for end-users

Avail % = (available time / total time) × 100

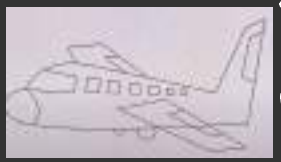
23 hr / 24 hr × 100 = 95.83%

Availability	Annual Downtime
99%	3 days, 15 hours, 40 minutes
99.8%	8 hours, 46 minutes
99.99%	53 minutes, 36 seconds
99.999%	5.26 minutes

common avail metric SLO

Reliability vs Availability

- Reliable system is always an available system
- Availability can be maintained by redundancy, but system may not be reliable
- Reliable software will be more profitable because providing same service requires less backup resources
- Requirements will depend on function of the software



Plane - once in the air, MUST be 100% reliable; tradeoff will always depend on use-case

Efficiency

- How well the system performs
- Latency and throughput often used as metrics

Manageability

- Speed and difficulty involved with maintaining system
- Observability, how hard to track bugs
- Difficulty of deploying updates
- Want to abstract away infrastructure so product engineers don't have to worry about it

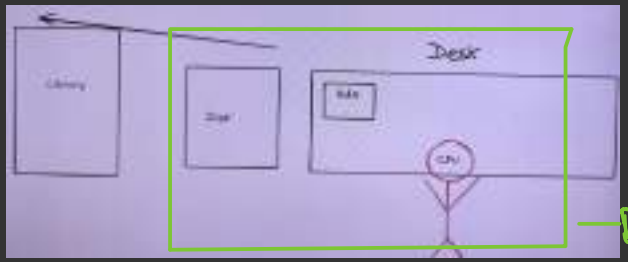
Numbers Programmers should know

Scaled to seconds to make relatable

LATENCY

Actual Time

Access Type	Time	Converted Time
CPU Cycle	.3 ns	1 S
CPU L1 Cache	1 ns	3 S
CPU L2 Cache	3 ns	9 S
CPU L3 Cache	13 ns	43 S
Main Memory(RAM)	120 ns	6 minutes
SSD	150 micro seconds	6 days
HDD	10 ms	12 months
SF to NYC	40 ms	4 years
SF to Australia	183 ms	19 years



- CPU = brain, info off the top of your head
- RAM = book on your desk, quick reference
- Disk = bookshelf in house, walk to get
- Network call outside datacenter = travel to a public library to get info

common data types

- Char – 1 byte
- Integer – 4 bytes
- UNIX Timestamp – 4 bytes

Data conversions

8 bits → 1 byte
1024 bytes → 1 kilobyte
1024 kilobytes → 1 Megabyte
1024 Megabytes → 1 Gigabyte
1024 Gigabytes → 1 Terabyte

LATENCY TAKEAWAYS

- Avoid Network calls whenever possible
- Replicate data across data centers for disaster recovery as well as performance
- Use CDNs to reduce latency
- Keep frequently accessed data in memory if possible rather than seeking from disk, caching

TIME CONVERSIONS

60 Seconds x 60 Minutes = 3600 Seconds per Hour

3600 x 24 hours = 86400 Seconds per Day

86400 x 30 days = 2,500,000 seconds per Month

INSTAGRAM EXAMPLE

- Estimate total number of requests app will receive
- Average Daily Active Users X average reads/writers per user

TRAFFIC ESTIMATES

10 Million DAU x 30 photos viewed = 300 Million Photo Requests

10 Million DAU x 1 photo upload = 10 Million Photo Writes

300 Million Requests // 86400 = 3472 Requests per Second

10 Million Writes // 86400 = 115 Writes per Second

MEMORY

- Read Requests per day x Average Request size x .3

300 Million Requests x 500 Bytes = 150 Gigabytes

150GB x .2(20%) = 30 Gigabytes

30GB x 3(replication) = 90 Gigabytes

STORAGE

- Writes per day x Size of write x Time to store data

10 Million Writes x 1MB = 10 TB per day

10 TB x 365 days = 10 years = 36 Petabytes

BANDWIDTH

- Requests per day x Request size

300 Million Requests x 1MB = 300,000 Gigabytes

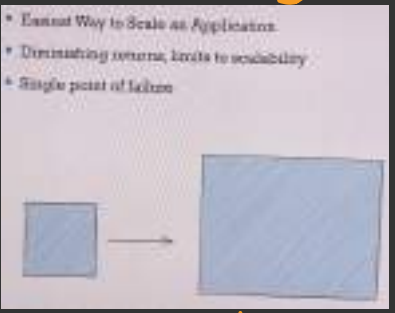
300,000GB // 86400 seconds = 3.4GB per second

Horizontal vs Vertical Scaling

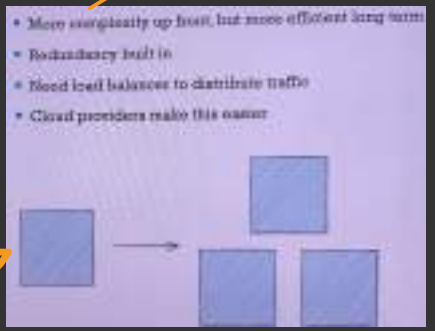
Why Scale?

1- require more processing power
2- need larger amounts of Data stored in RAM
3- faster read from storage
4- need bigger amount of Data pushed through network; streaming

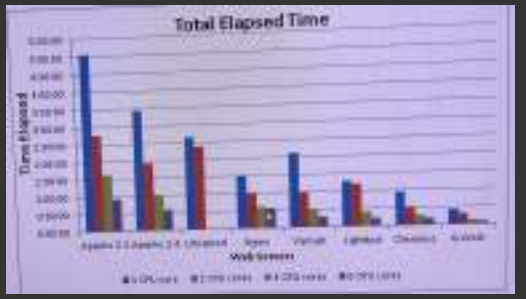
Vertical Scaling



Horizontal Scaling

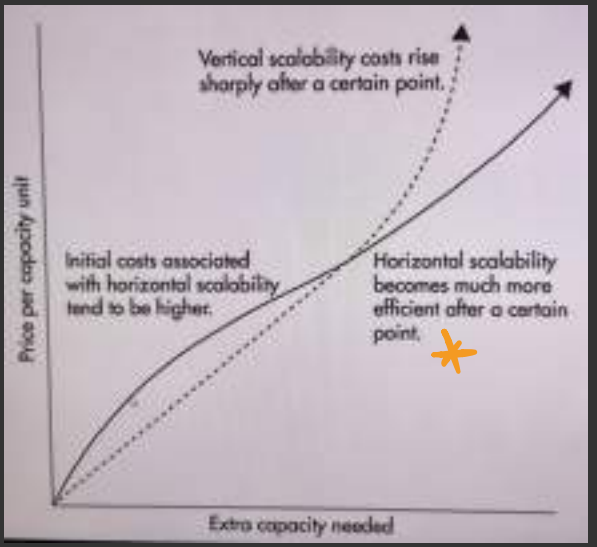


TradeOffs: Diminishing Returns



Latency, use Horizontal to improve & prevent SPOF

Summary Chart

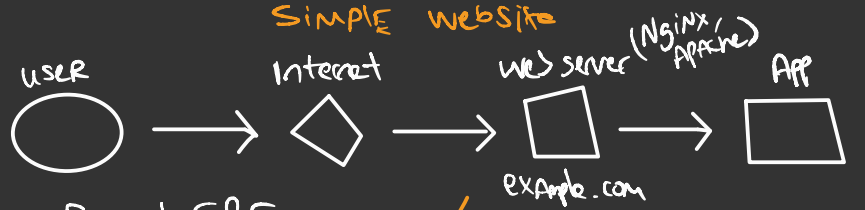


LOAD Balancers

- Balance incoming traffic to multiple servers
- Software or Hardware based
- Used to improve reliability and scalability of application
- Nginx, HAProxy, F5, Citrix

SOFTWARE HARDWARE (may lead to vendor lock-in)

Simple website



Prevent SPOF with LBs

Basic LB setup

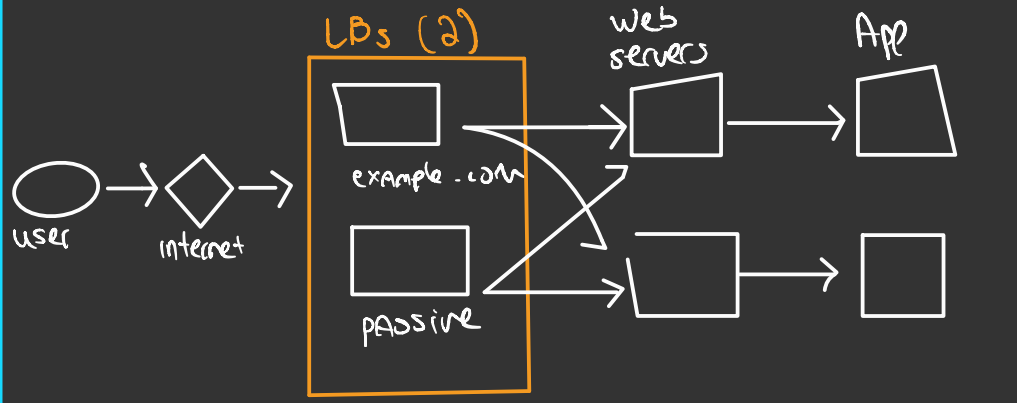
LB Types

- Layer 4
  - Only has access to TCP and UDP data
  - Faster
  - Lack of information can lead to uneven traffic
- Layer 7
  - Full access to HTTP protocol and data
  - SSL termination
  - Check for authentication
  - Smarter routing options

LB Routing Methods

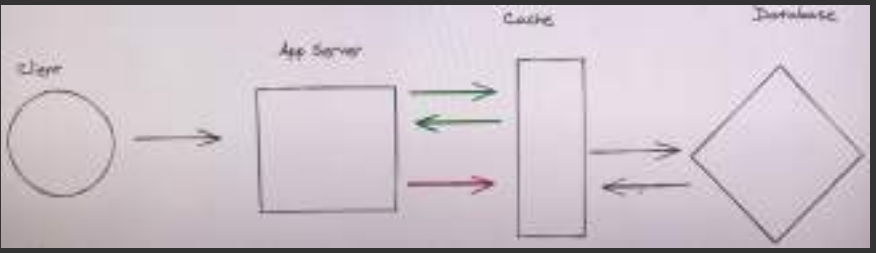
- Round Robin
  - Simplest type of routing
  - Can result in uneven traffic
- Least Connections
  - Routes based on number of client connections to server
  - Useful for chat or streaming applications
- Least Response Time
  - Routes based on how quickly servers respond
- IP Hash
  - Routes client to server based on IP
  - Useful for stateful sessions

Prod LB setup (redundant)



CACHING: TAKE data stored on hard drive/disk and place it in memory

- Improve performance of application
- Save money

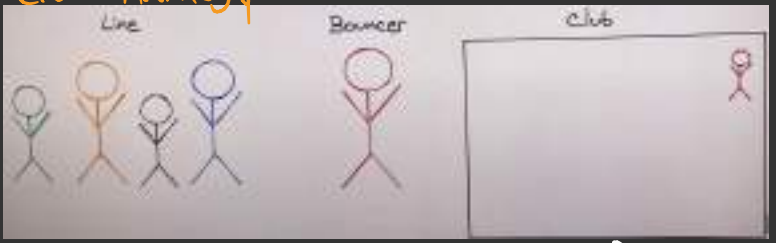


Speed & Performance

- Reading from memory is much faster than disk, 50-200x faster
- Can serve the same amount of traffic with fewer resources
- Pre-calculate and cache data
- Most apps have far more reads than writes, perfect for caching

- request quicker since retrieving from cache, efficient
- request slower if pulling from DB

CLUB ANALOGY:



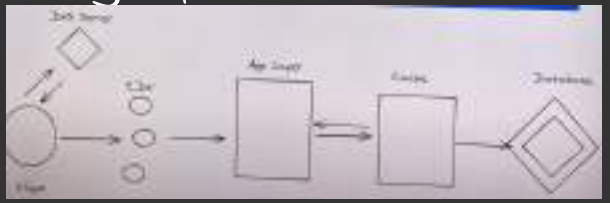
- Bouncer has password (in memory)
- Line requests entrance to club, Bouncer verifies pass
- ✗ Bouncer goes inside club each time to Ask Boss for password → idiot, would piss off line
- Bouncer = cache, Boss = DB, club = App, Line = Clients

YES/NO ↑

* CACHE is usually a giant in memory HASH TABLE

Caching layers, can sit in front of

- DNS
- CDN
- Application
- Database



CACHE Eviction

- Needed for 2 reasons:
- Preventing stale data
- Caching only most valuable data to save resources

TTL

- Set a time period before a cache entry is deleted
- Used to prevent stale data

LFU / LRU

- Least Recently used
  - Once cache is full, remove last accessed key and add new key
- Least Frequently used
  - Track number of times key is accessed
  - Drop least used when cache is full

Strategies

- Cache Aside – most common
- Read Through
- Write Through
- Write Back

Distributed CACHE

- Works same as traditional cache
- Has built-in functionality to replicate data, shard data across servers, and locate proper server for each key

Active, Passive

CACHE consistency

- How to maintain consistency between database and cache efficiently
- Importance depends on use case

DATABASE Scaling

- Most web apps are majority READS, around 95%+

- Indexes
- Denormalization
- Connection pooling
- Caching
- Vertical scaling

Scaling Techniques

Vertical Scaling

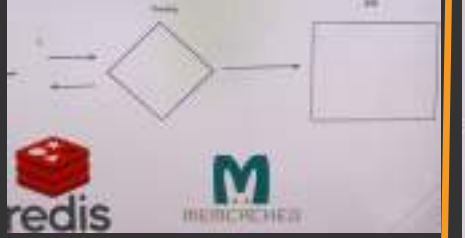
- Get a bigger server
- Easiest solution when starting out

Connection Pooling

- Allow multiple application threads to use same DB connection
- Saves on overhead of independent DB connections

Caching -> BEST WAY

- Not directly related to DB
- Cache sits in front of DB to handle serving content
- Can't cache everything



INDEX

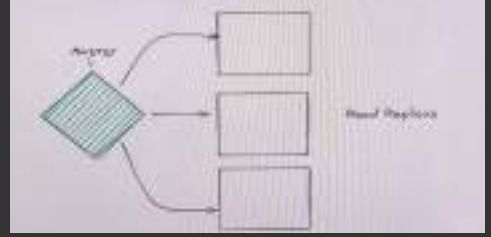
- Index based on column
- Speeds up read performance
- Writes and updates become slightly slower
- More storage required for index

Denormalization

- Add redundant data to tables to reduce joins
- Boosts read performance
- Slows down writes
- Risk inconsistent data across tables
- Code is harder to write

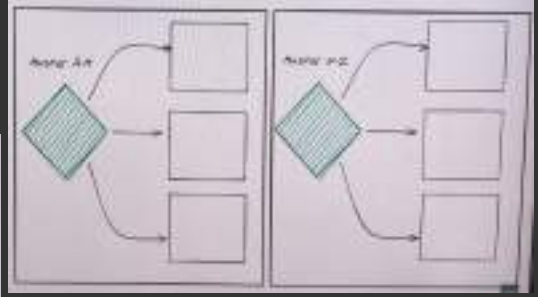
Replication & partitioning

- Create replica servers to handle reads
- Master server dedicated only to writes
- Have to handle making sure new data reaches replicas

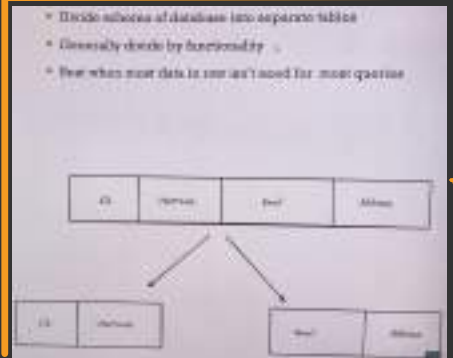


READ Replicas

- Horizontal partitioning
- Schema of table stays the same, but split across multiple DBs
- Downsides - Hot Keys, no joins across shards



Partitioning / Sharding



Vertical Partition

* choose NOSQL DB to scale, you upfront know the tradeoffs

DESIGN YouTube

1. Gather Reqs:

- Video upload
- View uploaded videos
- Search for videos
- Track stats of videos
- Comment on videos

2. Capacity Estimates:

- storage, bandwidth

3. Key Data:

600 hours of video uploaded per minute

1 Billion hours of video watched per day

Storage Estimates:

600 hours of video uploaded per minute

600 * 60 = 36000 minutes of video per minute

36000 * 60MB // 1000 = 2160GB uploaded per minute

2160GB * 60 * 24 // 1000 = 4.147TB per day

4.1GPB per day

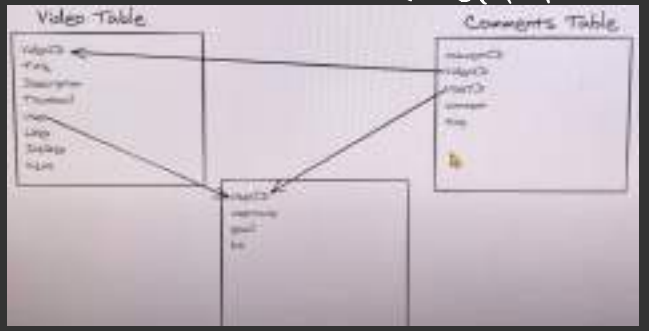
Bandwidth Estimates

1 Billion hours of video watched daily on YouTube

1000000000 hours * 3MB per hour // 1000 = 3000000 TB

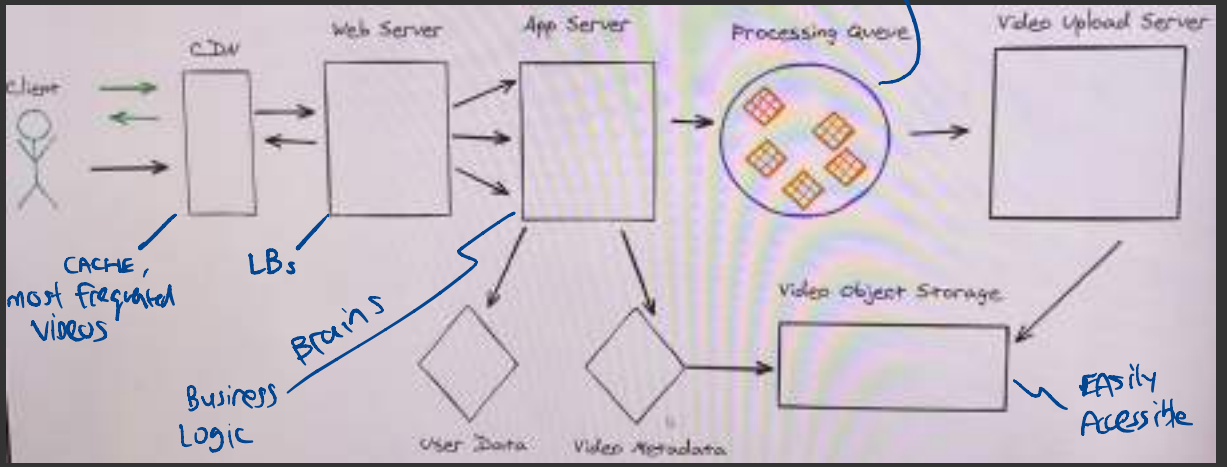
3,000 PB bandwidth/day

DB Design: YouTube uses custom RDS setup with high scaling SQL



System Architecture

makes uploading video less fragile, TASK ID, user ID



SUMMARY

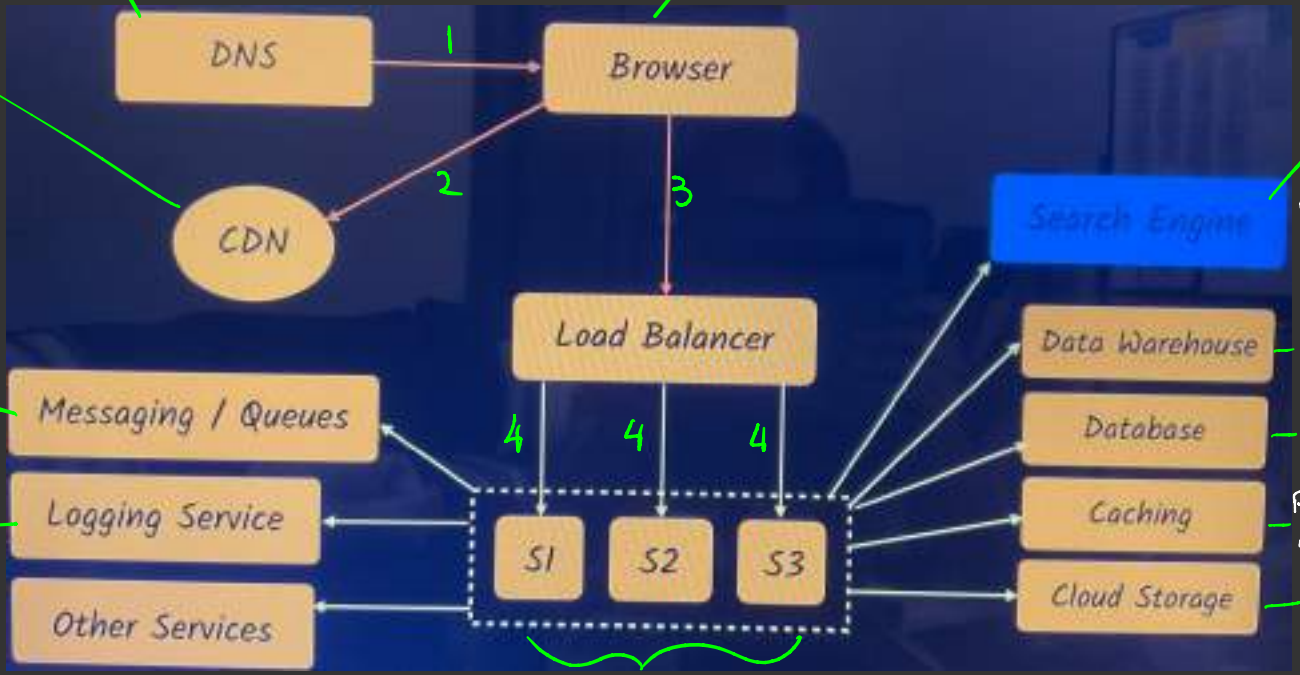
- Define Requirements
- Make estimates
- High level design
- Specific components

System Design 101

Browser uses DNS to get IP of the server

USER → google.com where is google.com?

2. without CDN = slow response, All request go to our single server, too much load & bandwidth



Server static Assets
- faster serving requests than single server

Full Txt search capabilities

SQS, SNS

snowplow

SQL, NoSQL, Graph

Elk Stack, view & search Logs

Redis, memcached

S3

SERVERS
- utilizing horizontal scaling, No SPoF