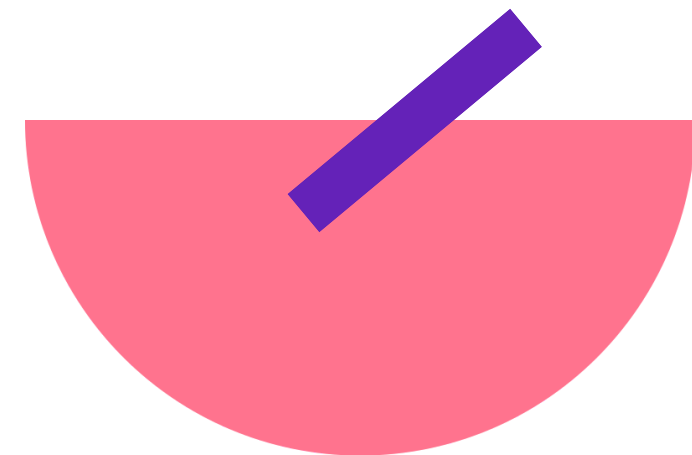
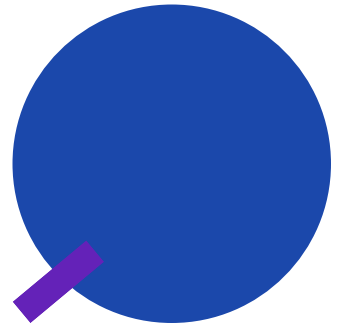


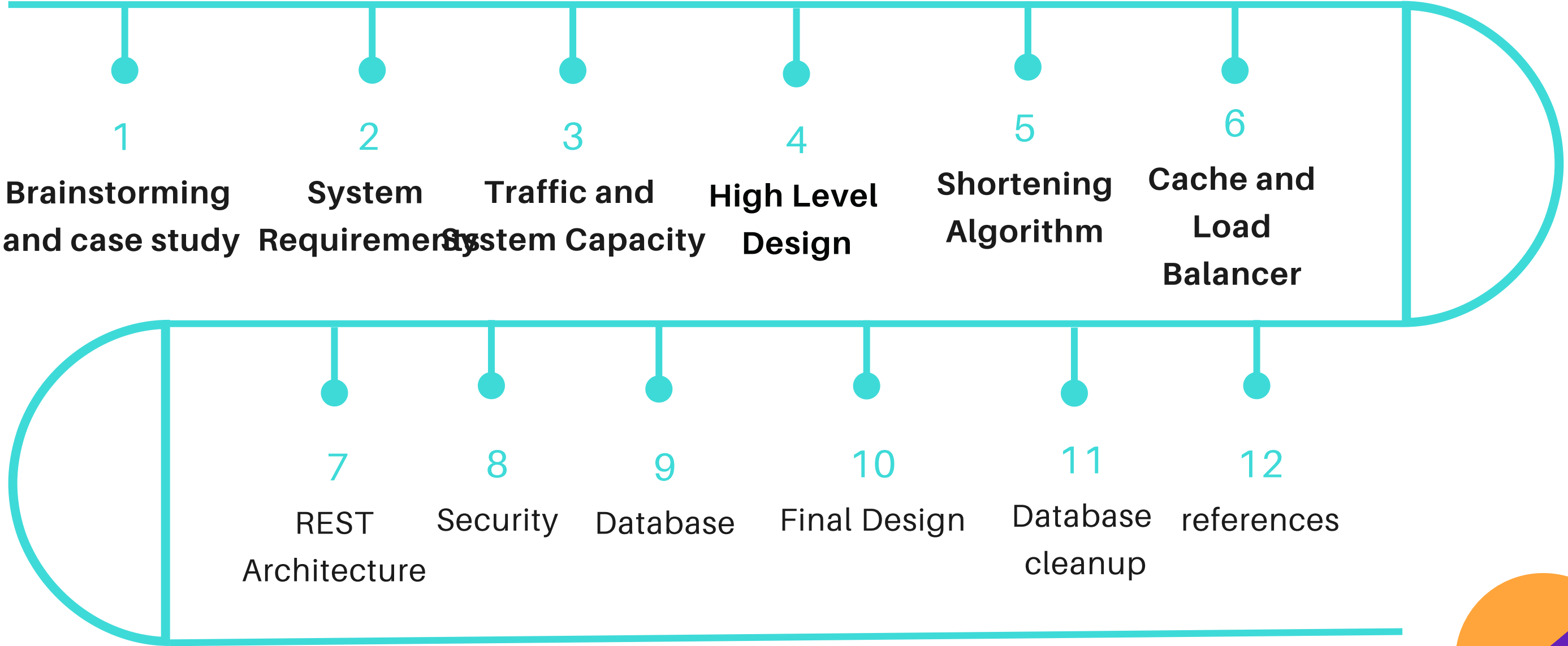


System Design

URL Shortening Service



Roadmap - URL Shortening Service



1- Brainstorming and case study



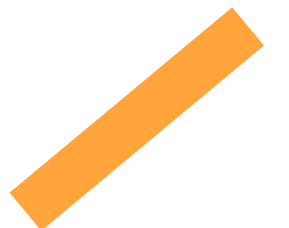
- An e-commerce startup has a very original idea, they want to create a URL shortening service.
- A visitor enters a very long URL into a field and gets a shortened version of it.
- Whenever the shortened version is used to request a web resource, the service will redirect to the original URL

LONG URL:

<https://www.reuters.com/article/urnidgns002570f3005978d8002576f60035a6bb>

SHORT URL

<https://shorturl.com/3sh2ps6v>



2- System Requirements

2.1- FUNCTIONAL REQUIREMENTS:

- Service should be able to **create shortened** url/links against a long url
- Click to the short URL should **redirect** the user to the original long URL
- Shortened link should be as **small** as possible
- that Links **can expire** after a default timespan



2- System Requirements

2.2- NON-FUNCTIONAL REQUIREMENTS:

- High availability: Service should be up and running all the time
- scalable & efficient & Minimal latency: URL redirection should be fast and should not degrade at any point of time (Even during peak loads)
- expose REST API's so that it can be integrated with third party applications
- Shortened links should not be predictable

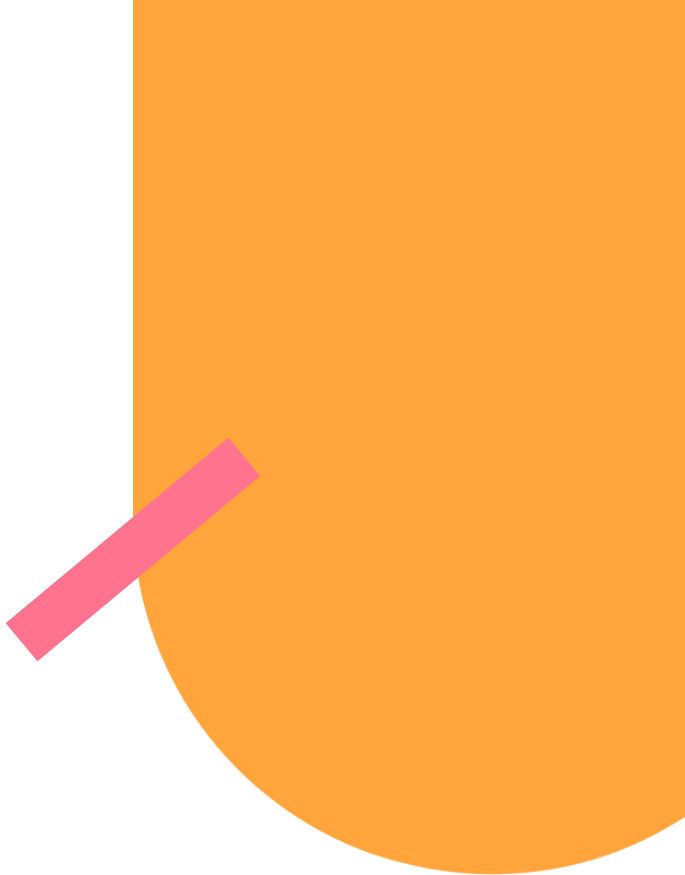
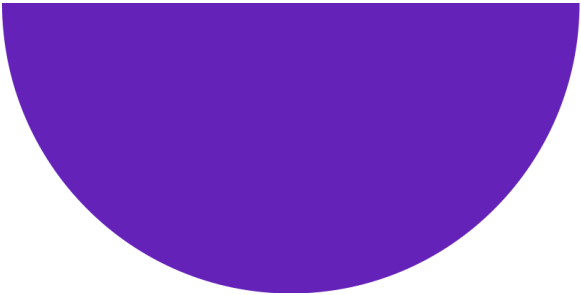


2- System Requirements


2.3- EXTENDED REQUIREMENTS:

- Service should collect metrics like most clicked links
- Optional Once a shortened link is generated it should stay in system for lifetime
- Users can create custom url with maximum character limit (EX: 7)





3- Traffic and System Capacity



3- Traffic and System Capacity



3.1- TRAFFIC

- Assuming 200:1 read/write ratio
- Number of unique shortened links generated per month = 100 million
- Number of unique shortened links generated per seconds
= $100 \text{ million} / (30 \text{ days} * 24 \text{ hours} * 3600 \text{ seconds}) \sim 40 \text{ URLs/second}$
- With 200:1 read/write ratio, number of redirections = $40 \text{ URLs/s} * 200 = 8000 \text{ URLs/s}$



3.2- STORAGE

- Assuming lifetime of service to be **100 years** and with **100 million** links /month
- total number of data points/objects in system will be
= $100 \text{ million/month} * 100 \text{ (years)} * 12 \text{ (months)} = \mathbf{120 \text{ billion}}$
- Assuming size of each data object (**Short url, long url, created_at and expiration_length_in_minutes**) to be **500 bytes** long.
- then total require **storage** = $120 \text{ billion} * 500 \text{ bytes} = \mathbf{60TB}$



3- Traffic and System Capacity

3.3- MEMORY

- Following Pareto Principle, better known as the 80:20 rule for caching. (80% requests are for 20% data)
- Since we get 8000 read/redirection requests per second, we will be getting 700 million requests per day:
- $8000/s * 86400 \text{ seconds} = \sim 700 \text{ million}$
- To cache 20% of these requests, we will need $\sim 70\text{GB}$ of memory.
- $0.2 * 700 \text{ million} * 500 \text{ bytes} = \sim 70\text{GB}$

3.4- BANDWIDTH

- if we assume each request is of size 500 bytes then the total incoming data
- then write requests would be $\Rightarrow 40 \times 500 \text{ bytes} = 20 \text{ KB/second}$
- Similarly, for the read requests, since we expect about 8K redirections,
- the total outgoing data would be: $8000 \text{ URLs/second} \times 500 \text{ bytes} = \sim 4 \text{ MB/second}$


-





3– Traffic and System Capacity summery

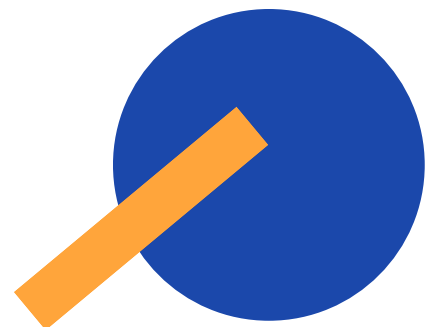
Type	Estimate	description
Writes (New URLs)	40/s	Shortened URLs generated
Reads (Redirection)	8K/s	Total URL requests/Reads
Bandwidth (Incoming)	20 KB/s	
Bandwidth (Outgoing)	4 MB/s	
Storage (100 years)	60 TB	
Memory (Caching)	~70 GB/day	





(4) Initial High Level Design

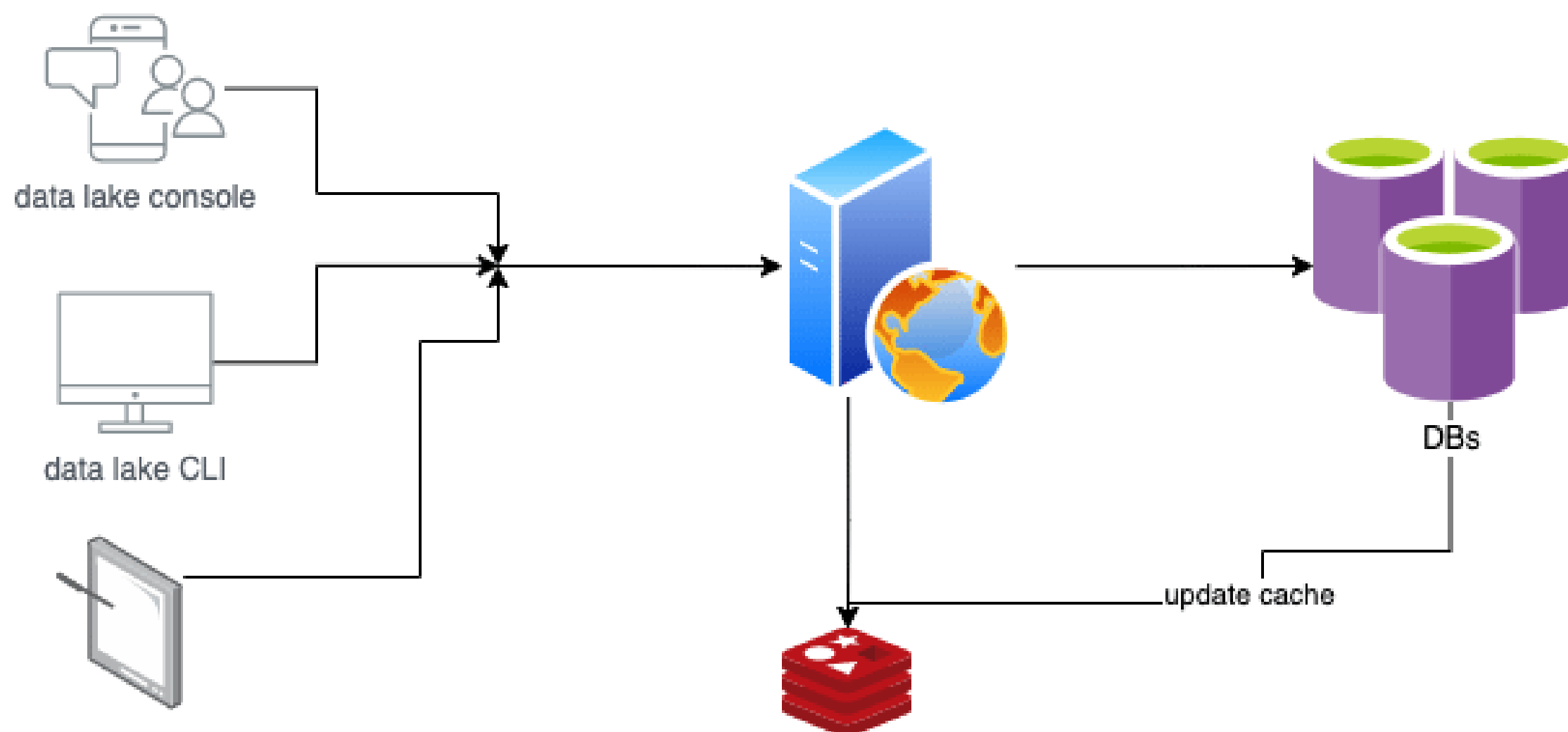


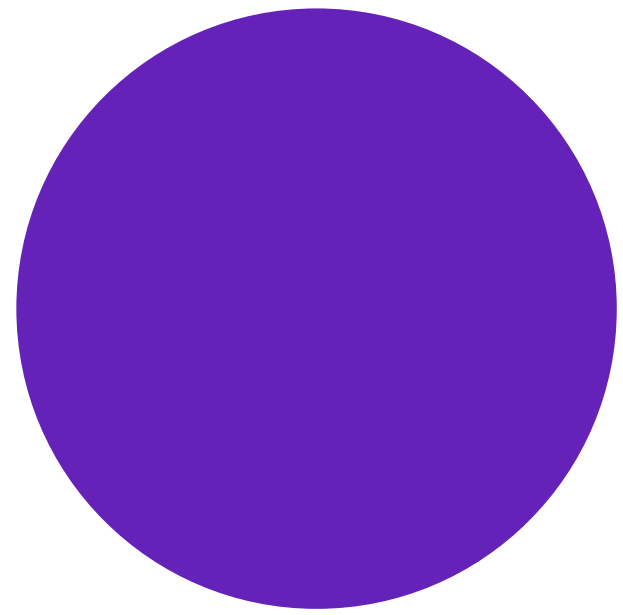


FOLLOWING IS AN INITAIL HIGH LEVEL DESIGN OF SERVICE

Problems with this design :

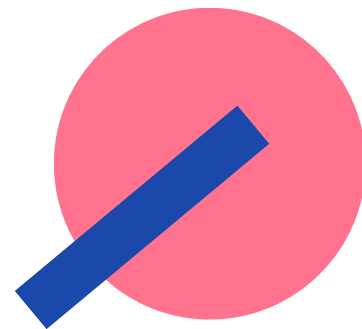
- There is only one WebServer which is single point of failure (SPOF)
- System is not scalable
- There is only single database which might not be sufficient for 60 TB of storage and high load of 8000/s read requests





(5) Shortening Algorithm

With all
shortening algorithms,
let's revisit our design goals!



REVISIT OUR DESIGN GOALS!


- Being able to store a lot of short links (**120 billion**)
- Our TinyURL should be as short as possible (**7 characters**)
- Application should **be resilient** to load spikes (For both url redirections and short link generation)
- Following a short link should **be fast**



#1

Technique

Base62 encoding



- A base is a number of digits or characters that can be used to represent a particular number.
- base 62 are $[0-9][a-z][A-Z]$ = total($26 + 26 + 10 = 62$)
- So for 7 characters short URL, we can serve $62^7 \approx 3500$ billion URLs which is quite enough
- in comparison to base10 (base10 only contains numbers 0-9 so you will get only 10M combinations).
- If we use base62 making the assumption that the service is generating 1000 tiny URLs/sec then it will take 110 years to exhaust this 3500 billion combination.
- We can generate a random number for the given long URL and convert it to base62 and use the hash as a short URL id.
- This is the simplest solution here, but it does not guarantee non-duplicate or collision-resistant keys.

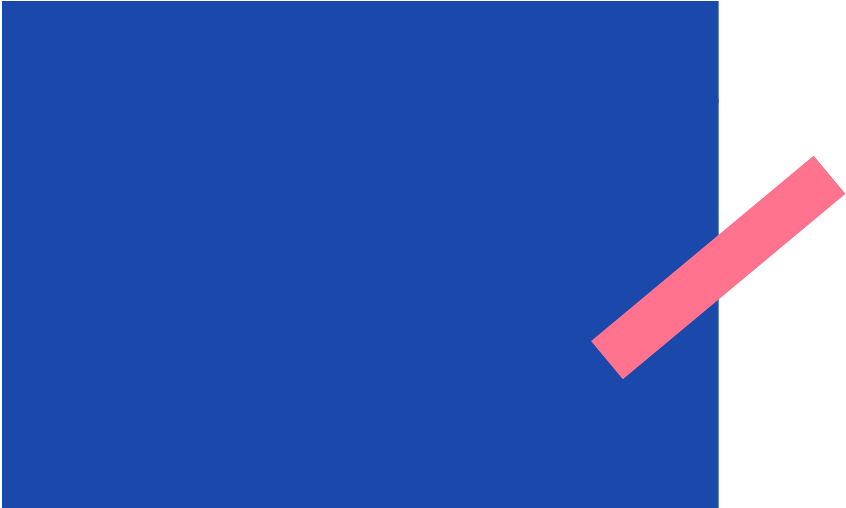



#2

Technique

MD5

Encoding



- 
- MD5 also gives base62 output but the MD5 hash gives a lengthy output which is more than 7 characters.
 - MD5 hash generates 128-bit long output so out of 128 bits
 - we will take 43 bits to generate a tiny URL of 7 characters.
 - MD5 can create a lot of collisions.
 - For two or many different long URL inputs we may get the same unique id for a short URL and that could cause data corruption.
 - So we need to perform some checks to ensure that this unique id doesn't exist in the database already



#3

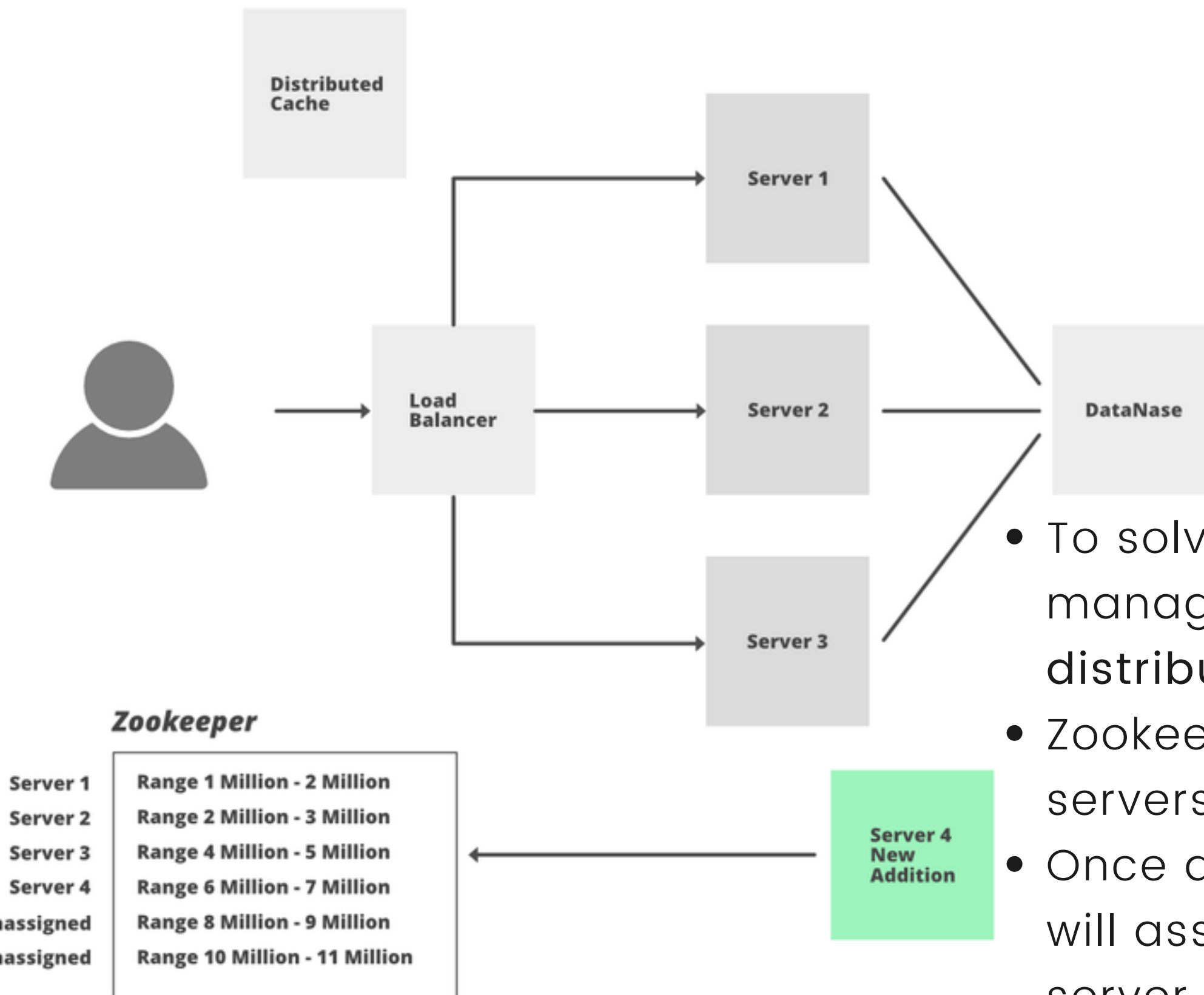
Technique

Counter Approach



- In this approach, we will start with a single server which will maintain the count of the keys generated.
- Once our service receives a request, it returns a unique number and increments the counter.
- Counter(0-3.5 trillion) → base62encode → hash
- The problem with this approach is that it can quickly become a single point for failure.
- if we run multiple instances of the counter we can have collision as it's essentially a distributed system
- if one of the counters goes down then for another server it will be difficult to get the range of the failure counter and maintain it.
- if one counter reaches its maximum limit then resetting the counter will be difficult because there is no single host available for coordination among all these multiple servers

Technique #3 Counter Approach Contu.



- To solve this issue we can use a distributed system manager such as **Zookeeper** which can provide distributed synchronization.
- Zookeeper can maintain multiple ranges for our servers.
- Once a server reaches its maximum range Zookeeper will assign an unused counter range to the new server.
- This approach can guarantee non-duplicate and collision-resistant URLs. Also, we can run multiple instances of Zookeeper to remove the single point of failure.




#4

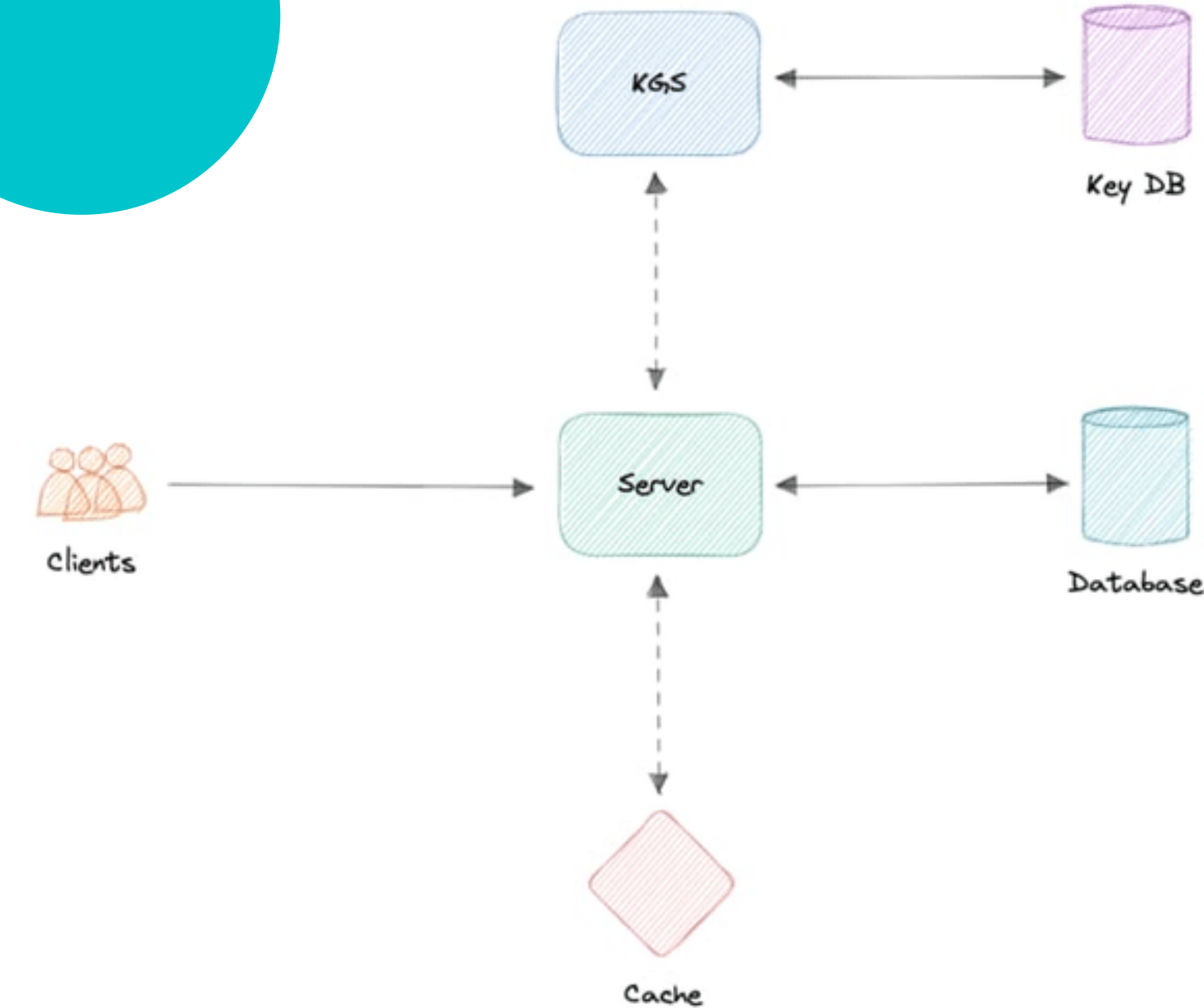
Technique

#KGS: Key Generation Service

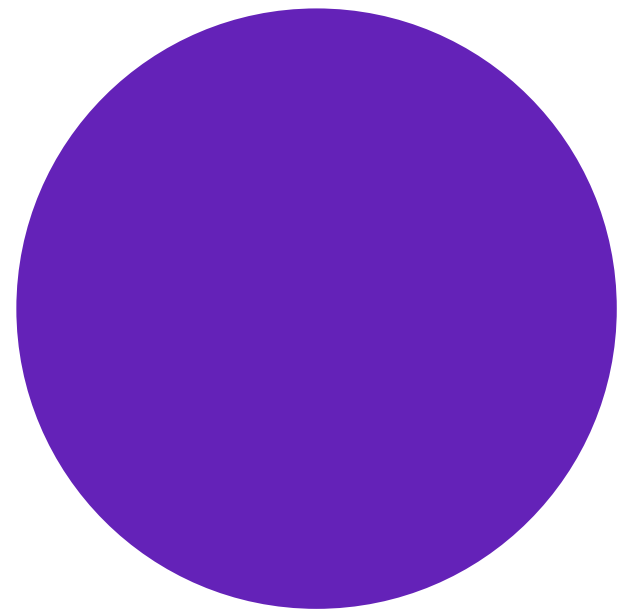


- 
- create a standalone Key Generation Service (KGS) that generates a unique key ahead of time and stores it in a separate database for later use.
 - KGS will make sure all the keys inserted into key-DB are unique
 - How to handle concurrent access?
 - if there are multiple server instances reading data concurrently, two or more servers might try to use the same key
 - KGS can use two tables to store keys: one for keys that are not used yet, and one for all the used keys.
 - As soon as KGS gives keys to one of the servers, it can move them to the used keys table.
 - KGS can always keep some keys in memory to quickly provide them whenever a server needs them

Technique #4 KGS

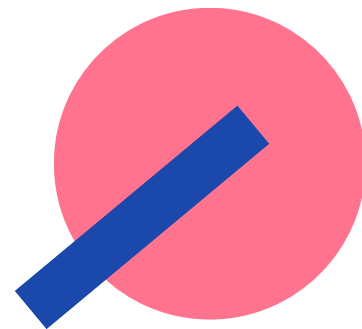


- Can each app server cache some keys from key-DB?
- Yes, this can surely speed things up. Although, in this case, if the application server dies before consuming all the keys, we will end up losing those keys. This can be acceptable since we have 68B unique six-letter keys.
- Isn't KGS a single point of failure?
- Yes, it is. To solve this, we can have a standby replica of KGS. Whenever the primary server dies, the standby server can take over to generate and provide keys.



(6) Caching

We can cache URLs
that are frequently accessed



CACHING

- We can cache URLs that are frequently accessed.
- We can use some off-the-shelf solution like **Memcached**, which can store full URLs with their respective hashes

HOW MUCH CACHE MEMORY SHOULD WE HAVE?

- We can start with 20% of daily traffic and, based on clients' usage patterns, we can adjust how many cache servers we need
- Since a modern-day server can have 256GB memory,
- we can easily fit all the cache into one machine.
- Alternatively, we can use a couple of smaller servers to store all these hot URLs.

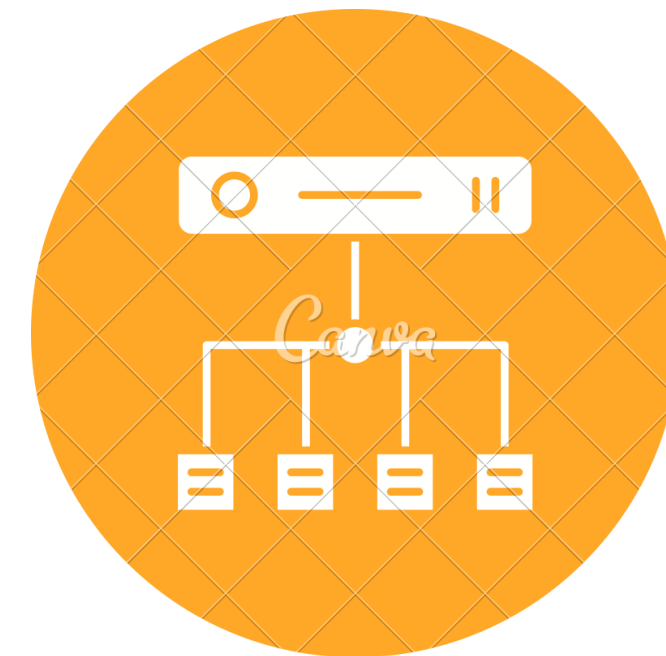
WHICH CACHE EVICTION POLICY WOULD BEST FIT OUR NEEDS?

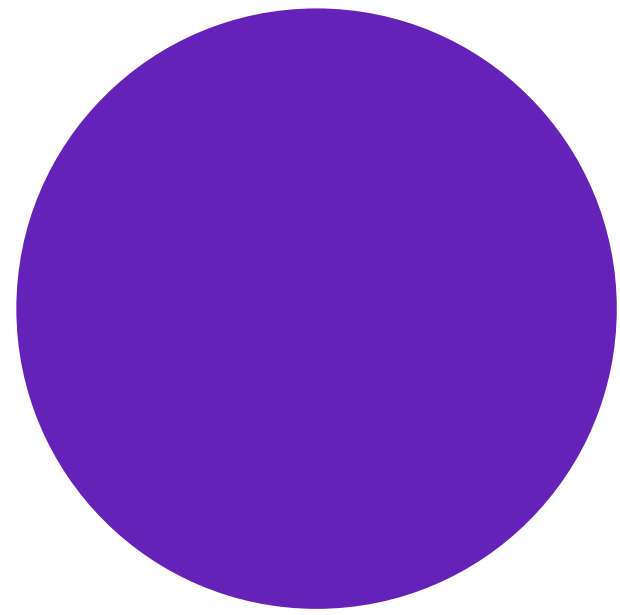
- When the cache is full, and we want to replace a link with a newer/hotter URL,
- how would we choose?
- Least Recently Used (LRU) can be a reasonable policy for our system

6- LOAD BALANCER (LB)

We can add a Load balancing layer at three places in our system:

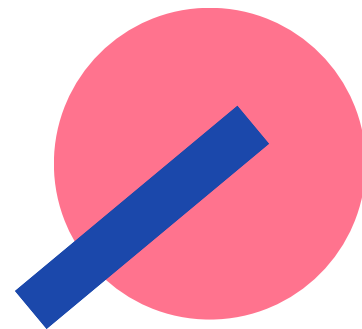
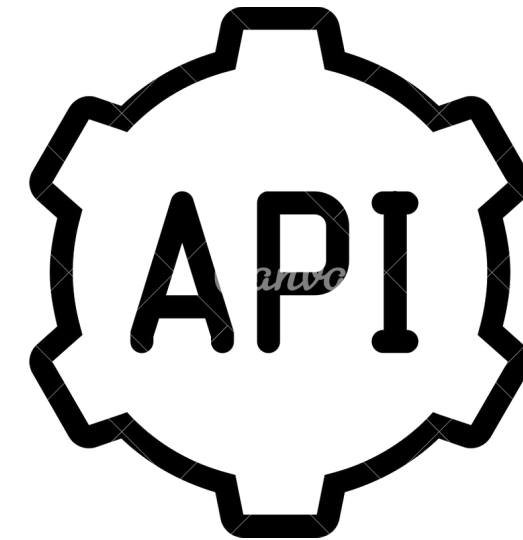
- Between Clients and Application servers
- Between Application Servers and database servers
- Between Application Servers and Cache servers





(7) REST APIs

Provide REST Endpoints





REST Endpoints



1-createURL

`createURL(apiKey: string, originalURL: string, expiration?: Date): string`

- Return OK (200),
 - with the generated short_url in data
 - `api_key`: A unique API key provided to each user,
 - to protect from the spammers, access, and resource control for the user, etc.
- 
- 

REST Endpoints

2-getURL

`getURL(apiKey: string, shortURL: string): string`

- Return a http redirect response(302)
- Note : “HTTP 302 Redirect” status is sent back to the browser instead of “HTTP 301 Redirect”.
- A 301 redirect means that the page has permanently moved to a new location. A 302 redirect means that the move is only temporary.
- Thus, returning 302 redirect will ensure all requests for redirection reaches to our backend and we can perform analytics (Which is a functional requirement)
- short_url: The short URL generated from the above function.
- Return Value: The original long URL, or invalid URL error code.

REST Endpoints

3-deleteURL

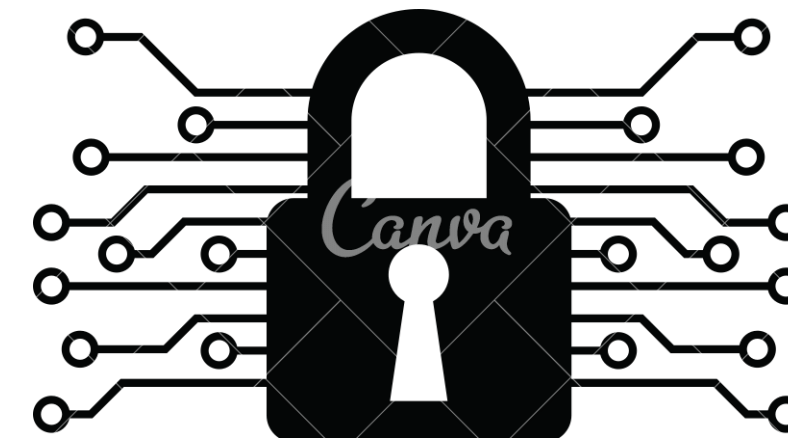
`deleteURL`(apiKey: string, shortURL: string): boolean

- just delete specific URL from DB & Cache
- Return a boolean
-

8- SECURITY

For security steps,

- we can introduce private URLs and authorization.
- A separate table can be used to store user ids that have permission to access a specific URL.
- If a user does not have proper permissions, we can return an HTTP 401 (Unauthorized) error.
- We can also use an API Gateway as they can support capabilities like authorization, rate limiting, and load balancing out of the box.

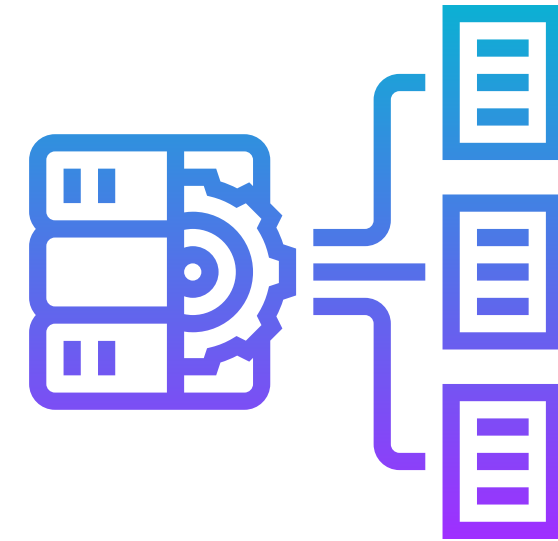




9- DATABASE

With database we have two options :

- Relational databases (RDBMs) like MySQL and Postgres
- “NoSQL”-style databases like MongoDB and Cassandra



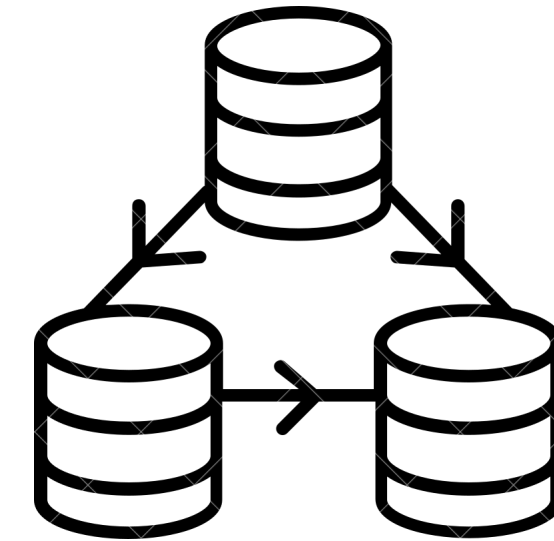




A) RDBMS

- We can use RDBMS which uses ACID properties but you will be facing the scalability issue with relational databases.
- Now if you think you can use sharding and resolve the scalability issue in RDBMS then that will increase the complexity of the system.
- There are 30M active users so there will be conversions and a lot of Short URL resolution and redirections.
- Read and write will be heavy for these 30M users so scaling the RDBMS using shard will increase the complexity of the design when we want to have our system in a distributed manner.
- You may have to use consistent hashing to balance the traffics and DB queries in the case of RDBMS and which is a complicated process.
- So to handle this amount of huge traffic on our system relational databases are not fit and also it won't be a good decision to scale the RDBMS.

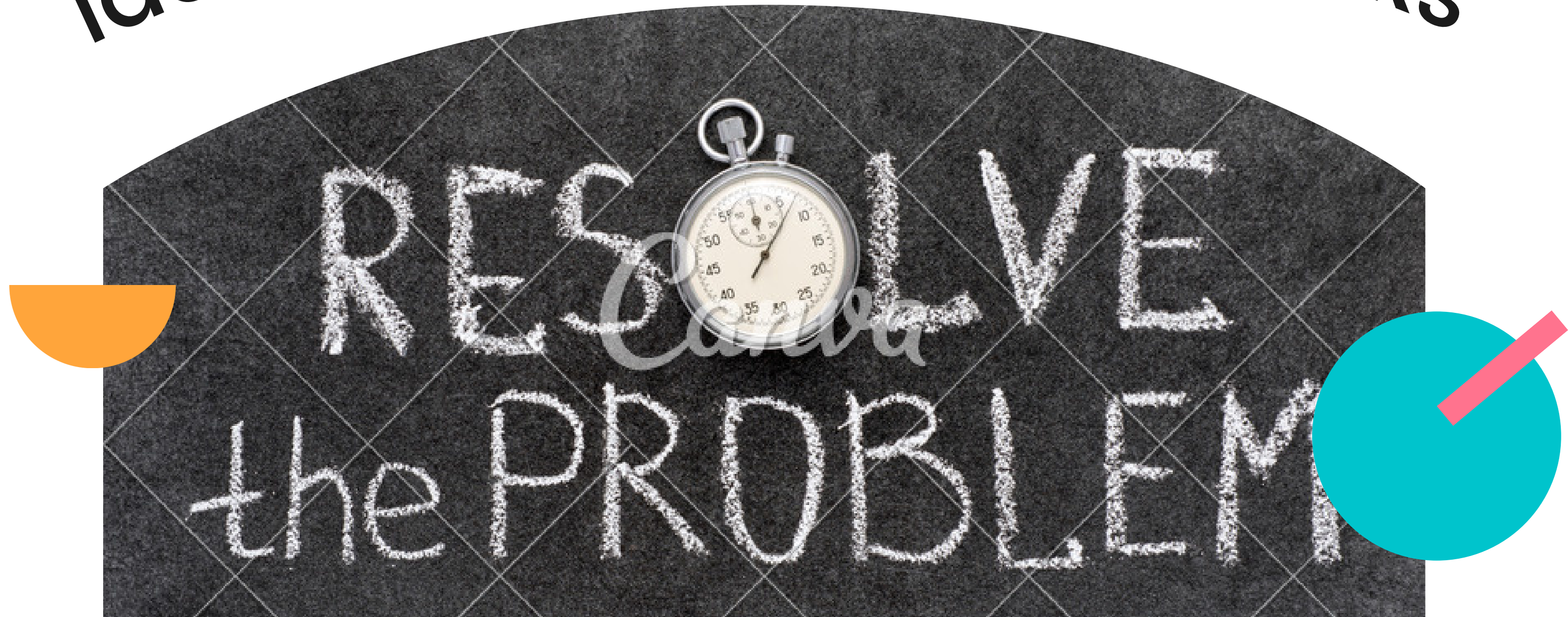


B) NOSQL!

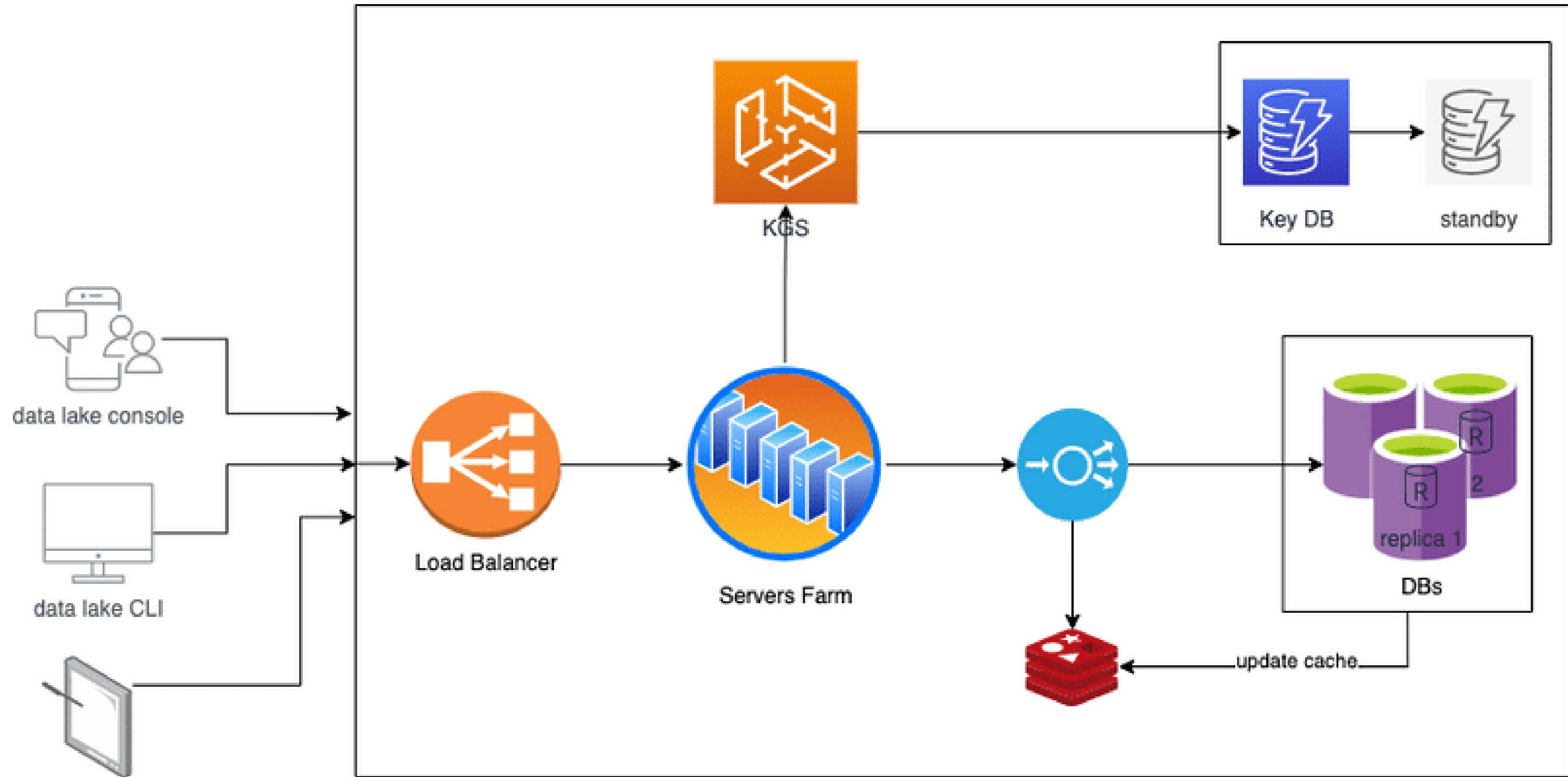


- The only problem with using the NoSQL database is its eventual consistency.
 - We write something and it takes some time to replicate to a different node but our system needs high availability and NoSQL fits this requirement.
 - NoSQL can easily handle the 30M of active users and it is easy to scale.
 - We just need to keep adding the nodes when we want to expand the storage
- 
- 

Identify + resolve bottlenecks



10- Final Design



11.Database cleanup

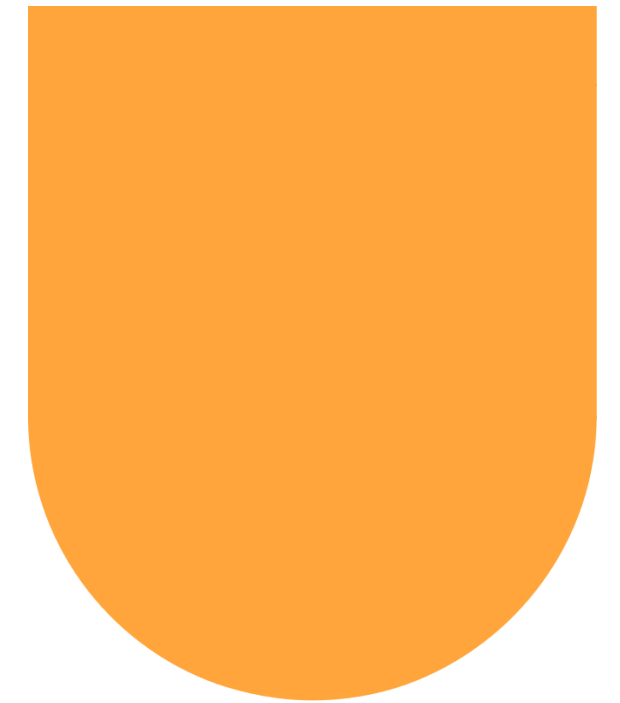
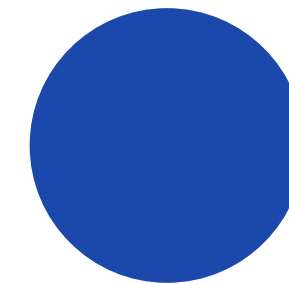
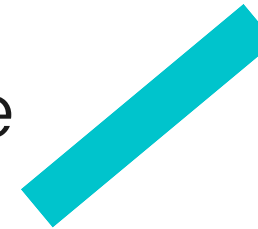
This is more of a maintenance step for our services and depends on whether we keep the expired entries or remove them. If we do decide to remove expired entries, we can approach this in two different ways:

1- Active cleanup

In active cleanup, we will run a separate cleanup service which will periodically remove expired links from our storage and cache. This will be a very lightweight service like a cron job.

2- Passive cleanup

For passive cleanup, we can remove the entry when a user tries to access an expired link. This can ensure a lazy cleanup of our database and cache.



Thank you for your time!



12- REFERENCES

- 1- <https://medium.com/@sandeep4.verma/system-design-scalable-url-shortener-service-like-tinyurl-106f30f23a82>
- 2- <https://www.youtube.com/watch?v=AVztRY77xxA>
- 3- <https://www.geeksforgeeks.org/system-design-url-shortening-service/>