System Design Note:

Single Server System:

Request Flow:
User (web browser, mobile app) sends the domain.com to the DNS (Domain Name System) who send back the IP address back to the device and then HTTP request is sent to the web server then the webserver returns an HTML page to the device.

When we talk about traffic to a web server, we mean the volume of requests and responses going between clients (browsers, mobile apps, etc.) and your server.

Source of Traffic:
Web application: service side languages and client side languages for presentation
mobile application: http protocol is the mean of communication between mobile app and the webserver

to the database but that should not be the case in a bigger system, this can be separated as the database can have its own server

Relational Database (RDBMS):
Stores data in tables with rows and columns and this tables are related to each other through relationships
E.g MySQL, PostgresSQL, Oracle, SQL Server
You can perform join operations using SQL across different tables

Non relational database (NoSQL):
It doesn't use traditional tables with rows and columns but in document like format, key-value pairs, graphs
You cant perform join operations

If you want to store a massive amount of data or data is unstructured or your application requires super low latency you can make use of a non relational database

Low latency means the response time for your application is fast

Vertical Scaling vs Horizontal scaling:
Vertical involves adding more power to your servers

Horizontal allows you to scale by adding more servers to your pool of resources

When traffic is low, vertical is the best option but it comes with serious limitation:
- It has a hard limit, it is impossible to add add an unlimited cpu power to one server
- If one server goes down, the whole server is down

in the case of the single system, if we are vertical scaling and the server fails, the whole application fails and if multiple users make use of the application and they get above the load limit, they will get slower response and this is where load balancing comes in
No 🙂, a load balancer cannot be used in vertical scaling since its just one server.

Load Balancer:
used to distribute loads (traffic) among servers

- If server 1 breaks, server 2 can take up its traffic
- If the traffic is too much for 1 then it can be evenly shared between 1 and 2

But our current structure has only one database meaning it does not support failover and redundancy so database replication helps to solve that

Master slave architecture is when there are two database master and slave whereas the master handles all writes (INSERT,UPDATE,DELETE) and the slave automatically replicates the masters data and handles read requests (SELECT)

It has some drawbacks in the case of slaves not having the latest data instantly, the master fails and no write can happen

```
             WRITE REQUESTS
               ┌──────────────────┐
              | Master |
               └────────┬─────────┘
      REPLICATION       |
    ┌────────────────────────────┴────────────────────────────┐
   |                            |
  ┌────────────────┐                      ┌────────────────┐
| Slave1|                    | Slave2|
  └───────┬────────┘                      └───────┬────────┘
    |                         |
  READ REQUESTS              READ REQUESTS
```

Advantages of database replication

Better Performance
Reliability in case of disaster since data is stored in various location
High availability

If only one slave database is available and it goes offline, read operation are sent to the master database temporarily
If the master is offline, a slave will be promoted to master

so now we have a situation where a user gets the ip address after sending the url of the site to the DNS then it moves down to the load balancer which distributes the load and sends the http request to either server 1 or 2 and then the servers routes write operation to the master and then a read to the slave

To improve response time is where the cache comes in

CACHE:
A cache is a temporary storage area that stores the result of expensive responses or frequently accessed data in memory so that subsequent responses are served more quickly

Once a response is received by the server, they check the cache is it has the data, if it has the data is sent back to the client else if not available in the cache the database checks and then stores it and it also gets stored in the cache before sending it back to the client

- Decide when to use the cache
- Expiration policy
- Consistency
- A single point of failure so there should be more than one cache across various data centers
- Eviction Policy

Content Delivery network (CDN):
A CDN is a network of geographically dispersed servers used to deliver static content.
When a user loads a site, the CDN server closest to the user delivers static content. it all depends on how far a user is from its CDN server, if very far, user will get content slower

if the image or static content is not yet in the CDN, they go to the server to fetch then store it in the CDN after

A CDN is like a cache but not a cache as they have some clear differences

CDN stores mainly static content but a cache stores any response or data. a CDN is global while a cache is local. A CDN is found in distributed servers across the globe but a cache is found in the infrastructure or device

Consideration for using a CDN:
- The cost
- expiry date(setting a proper expiry date)

so now a CDN has been added to our architecture, it is checked first before the load balancer then the two servers then the master and slaver database then the stores in cache if not currently present

Stateless web tier
In system design, a web tier refers to the layer responsible for handling HTTP requests and responses — basically, the frontend-facing servers that users interact with. Making this tier stateless is an important architectural principle for scalability, resilience, and fault tolerance.
A stateless web tier means that the web servers do not store any user session data or state locally.
Each incoming request is treated independently, as if it’s the first request from that client.

A stateful architecture remembers client data from one request but a stateless does not store or keep

In this stateless architecture, HTTP requests from users can be sent to any web servers, which fetch state data from a shared data store. State data is stored in a shared data store and kept out of web servers. A stateless system is simpler, more robust, and scalable

StateFullness
When a user logs in, the session is stored locally on Server A.

If the next request goes to Server B, B doesn’t know the user is logged in.

To fix this, you’d need sticky sessions (always send the same user to the same server).

This creates a single point of failure: if Server A crashes, the user’s session is lost.

Solution: Stateless web tier

Store session data in a centralized store or pass it in the request.

Any server can handle any request.

Stateful means the server does remember previous interactions while stateless means the server does not remember previous interactions
Stateful is harder than stateless to implement and is not that scalable. stateless is good for distributed systems while stateless is used for session heavy apps. In a stateful system, the server remembers the client's state between requests. This means that once you connect to a particular server, all your future requests must go to that same server for the session to work properly

Stateless architectures have a shared storage

User -> DNS -> USER (After checking the CDN) -> load balancer -> servers -> the master and slave database -> cache -> SQL database to store sessions in order to enforce statelessness

Data Centers:
So the bringing about of data centers presents us with a new dynamic where after the communication between the DNS, user and the CDN, the load balancer splits the load among various data canters. and this data canters which are geo based (different location geographically) have their own servers and their own databases and also their own caches with each of the servers connected to one general NoSQL database to store sessions

In the event of the failure of one data center, we direct all traffic to an healthy one

Several Technical Challenges must also be resolved:
- Traffic redirection: properly redirecting traffic to the right server
- Test and deployment: data centers should be tested in various locations

Message Queue:
It can be used to store, manage and deliver messages between different components
contains producers, queue and consumers

producers publish this messages and then publish them to the message queue and the consumers takes connects to the queue and performs actions based on the messages

Logging, metrics, automation
Logging is used to detect errors, since multiple users are using the app, issues might occur that we may not be able to detect. maybe a request jumping across several services or something fails

Metrics are numerical measurements that shows our systems health. can be used to monitor latency, availability or alerts or throughput (request per seconds)

Automation occurs when a system gets big and complex, we need to continually improve productivity

User -> DNS - User -> CDN -> Load balancer -> Data center(made up of the servers (which is also connected to the data base made up of either the master or the slaves) -> message queue -> workers ->) -> no SQL database to store sessions. Data center is connected to (logging,metrics,monitoring and automation)

Horizontal Scaling is also known as sharding
Sharding separates large databases into smaller, more easily managed parts called shards.
Each shard shares the same schema, though the actual data on each shard is unique to the
shard.
A hash function is used to store and fetch data. i.e user_id % 4 if id is 1, the data is sent to shard 1 and if 2 it is sent to shard 2

A shardkey helps you to retrieve data from a database efficiently by properly routing database query to the correct database.

Problems with sharding:
The celebrity problem where one celebrity has too much data and sharing shard with other can cause server overload so we need to allocate a shard just to the celebrity

Resharding data: Resharding data is needed when 1) a single shard could no longer hold
more data due to rapid growth. 2) Certain shards might experience shard exhaustion faster than others due to uneven data distribution. When shard exhaustion happens, it requires updating the sharding function and moving data around. Consistent hashing, which will be discussed in Chapter 5, is a commonly used technique to solve this problem

Join operation is hard when we want to perform join operation

User -> DNS - User -> CDN -> Load balancer -> Data center(made up of the servers (which is also connected to the data base made up of shards and also to the caches) -> message queue -> workers ->) -> no SQL database to store sessions. Data center is connected to (logging,metrics,monitoring and automation)
Note: Multiple data centers

keep web teir stateless
sharding
cache data as much as you can to improve response
monitor your device through the use of automation tools
Make use of the CDN to store static content
Load balancer to properly share load among servers
multiple data centers
build redundancy
split teirs into individual services

CHAPTER 2:
BACK OF THE ENVELOPE ESTIMATION:

Power Of Two:
Data Volume Unit using power of two:
10
20
30
40
50

Availability numbers:
High availability is the ability of a system to continuously be operational for a desired long period of time.

Usefulness of estimation time:
- Helps in capacity planning

- sets realistic expectation
- Guides Trade-offs
- Enables Proper Monitoring and adjustments
- Improves reliability & scaling

Commonly asked back-of-the-envelope estimations: QPS, peak QPS, storage, cache,
number of servers, etc. You can practice these calculations when preparing for an
interview. Practice makes perfect.

For something like youtube, we consider

Daily average users
Views per day
Avg video length
Quality mix (bitrate): 480p 1 Mbps (30%), 720p 2.5 Mbps (50%), 1080p 5 Mbps (20%)
→ Avg bitrate = 0.3·1 + 0.5·2.5 + 0.2·5 = 2.55 Mbps

Peak Concurrency
Segment size
Uploads
Download usage

Inputs:
- DAU = ...
- Views/user/day = ...
- Avg length (s) = ...
- Bitrate mix (Mbps) = {...}
- Peak concurrency = ...%
- Segment length (s) = ...
- Uploads = ... hours/min
- CDN hit rate = ...%
- Server capacity = ... Gbps

Compute:
- Views/day = DAU * Views/user
- Avg bitrate = weighted sum
- Peak concurrent = DAU * peak%
- Peak bandwidth = concurrent * avg bitrate

- Origin bandwidth = (1 - hit_rate) * peak bandwidth
- Segment QPS = concurrent * ((length/segment) / length) = concurrent * (1/segment_length)
- Upload hours/day = uploads/min * 60 * 24
- Stored GB/hr (ladder) ≈ bitrate_total*3600/8/1024
- New storage/day = hours/day * GB/hr
- Edge/origin servers = bandwidth (Gbps) / server_capacity (Gbps) with headroom

CHAPTER THREE

TIPS to ace a system design interview:

Step 1- Understand the problem and establish design scope:

In a system design interview, giving out an answer quickly without thinking gives you no
bonus points. Answering without tiral understanding of the requirements is a huge red flag. As an engineer, we like to solve hard problems and jump into the final design; however, this
approach is likely to lead you to design the wrong system.

What kind of questions to ask?
- What specific features are we going to build
- How many users does the product have
- How fast does the company anticipate to scale up
- What is the company's technology stack

What kind of (wether app or site)
the kind of users and amount of users to attract
Traffic volume (also users)
what it contains

Ask questions about requirements related to the company

Step 2- Propose high level design and get buy in:
- Come up with an initial design and communicate with interviewer wether they like it or not
- Draw boxes and shapes
- Do back of the envelope calculations

Step 3- Design Deep dive:
At this stage, you and the interviewer already have an understanding on how to go about what it is that is being built

- The goals, the features
- High level blueprint
- Ideas on areas to focus on

In most cases, the interviewer may want you to dig into details of some system components. For URL shortener, it is interesting to dive into the hash function design that converts a long URL to a short one. For a chat system, how to reduce latency and how to support online/offline status are two interesting topics

Step 4:
Wrap Up:
Interviewer might ask you a follow up questions:
- The interviewer might ask about bottlenecks or problems you see and think they can be improved cause no system is perfect.
- Useful to give interviewer a recap of design
- Error cases
- Operation issues
- How to scale
- Propose other refinements you need if you had more time.

Dos
• Always ask for clarification. Do not assume your assumption is correct.
• Understand the requirements of the problem.
• There is neither the right answer nor the best answer. A solution designed to solve the
problems of a young startup is different from that of an established company with millions
of users. Make sure you understand the requirements.
• Let the interviewer know what you are thinking. Communicate with your interview.
• Suggest multiple approaches if possible.
• Once you agree with your interviewer on the blueprint, go into details on each
component. Design the most critical components first.
• Bounce ideas off the interviewer. A good interviewer works with you as a teammate.
• Never give up.

Don’ts
• Don't be unprepared for typical interview questions.
• Don’t jump into a solution without clarifying the requirements and assumptions.
• Don’t go into too much detail on a single component in the beginning. Give the high
level design first then drills down.

• If you get stuck, don't hesitate to ask for hints.
• Again, communicate. Don't think in silence.
• Don’t think your interview is done once you give the design. You are not done until your
interviewer says you are done. Ask for feedback early and often.

Time Allocation:
Step 1 Understand the problem and establish design scope: 3 - 10 minutes
Step 2 Propose high-level design and get buy-in: 10 - 15 minutes
Step 3 Design deep dive: 10 - 25 minutes
Step 4 Wrap: 3 - 5 minutes

Chapter 4:
Design A rate limiter

A rate limiter is used to controlthe rate o traffic sent by a client or a service. if the api request count exceeds the threshold defined by the rate limiter, all the calls will be blocked.

1- you can not create more than 10 posts a day
2- you can not upload more than 5 vidoes a day
3- You can not create max of 10 accounts a day

Before we start we look at the advantages of using a rate limiter:
- Companies use it because it reduce the number of requests they take in per day hence fewer servers
- Preventr server overload
- Prevent starvation that comes from DOS attack by blocking excess calls (Denial of service attack)

A DoS attack stands for Denial of Service attack. It’s a type of cyberattack where an attacker tries to make a system, service, or network unavailable to its intended users by overwhelming it with excessive requests or traffic.

Step 1 - Understand the problem and establish design scope
Rate limiting can be implemented using different algorithms, each with its pros and cons. The
interactions between an interviewer and a candidate help to clarify the type of rate limiters we
are trying to build.
Candidate: What kind of rate limiter are we going to design? Is it a client-side rate limiter or
server-side API rate limiter?

Interviewer: Great question. We focus on the server-side API rate limiter.
Candidate: Does the rate limiter throttle API requests based on IP, the user ID, or other
properties?
Interviewer: The rate limiter should be flexible enough to support different sets of throttle
rules.
Candidate: What is the scale of the system? Is it built for a startup or a big company with a
large user base?
Interviewer: The system must be able to handle a large number of requests.
Candidate: Will the system work in a distributed environment?
Interviewer: Yes.
Candidate: Is the rate limiter a separate service or should it be implemented in application
code?
Interviewer: It is a design decision up to you.
Candidate: Do we need to inform users who are throttled?
Interviewer: Yes.
Requirements
Here is a summary of the requirements for the system:

- Accurately limit excessive requests.
- Low latency. The rate limiter should not slow down HTTP response time.
- Use as little memory as possible.
- Distributed rate limiting. The rate limiter can be shared across multiple servers or
processes.
- Exception handling. Show clear exceptions to users when their requests are throttled.
- High fault tolerance. If there are any problems with the rate limiter (for example, a cache
server goes offline), it does not affect the entire system.

Let us use an example in Figure 4-3 to illustrate how rate limiting works in this design.
Assume our API allows 2 requests per second, and a client sends 3 requests to the server
within a second. The first two requests are routed to API servers. However, the rate limiter
middleware throttles the third request and returns a HTTP status code 429. The HTTP 429
response status code indicates a user has sent too many requests

. Here is a list of popular algorithms for creating a rate limiter:

- Token bucket
- Leaking bucket
- Fixed window counter
- Sliding window log
- Sliding window counter

Token Bucket:
a widely used algorithm for rate limiting. it involves passing tokens into a bucket and when it gets to the limit, it overflows

Tokens = permission to send data.
Bucket = storage for tokens.
Data packets = water flow.
If you have tokens → you can send data.
If you don't → you must wait until tokens refill.
Stripe and amazon uses this

How many buckets do we need? This varies, and it depends on the rate-limiting rules. Here
are a few examples.
- It is usually necessary to have different buckets for different API endpoints. For instance,
if a user is allowed to make 1 post per second, add 150 friends per day, and like 5 posts per
second, 3 buckets are required for each user.
- If we need to throttle requests based on IP addresses, each IP address requires a bucket.
- If the system allows a maximum of 10,000 requests per second, it makes sense to have a
global bucket shared by all requests.

Leaking Bucket Algorithm:
if the request arrives and the system checks if the queue is full. If it is not full, the request is added to the queue
Otherwise the request is dropped
Requests are pulled and processed at regular interval

```
requests -> bucket -> queue -> requests throughput
                           |
        if full it get dropped
```

It looks into two parameters (bucket size and the out flow rate)
It is used in network bandwidth management of video streaming use cases and

also server request handling

Fixed Window Counter Algorithm:
The algorithm assigns a fixed sized time windows
- each request increment tge counter
once the counter reaches its limit, new requests are dropped until a new window starts

A major problem with this algorithm is that a burst of traffic at the edges of time windows
could cause more requests than allowed quota to go through.
Users can exploit this by sending requests at the very end of one window and immediately at the start of the next → effectively doubling their quota within a short real-time span.

Sliding window log algorithm:
The fixed window counter has an issue with the timing and it being able to be exploited but the sliding window fixes this by keeping track of the request timestamps in the cache so when a new requests comes in, the outdated one are removed

It is more accurate and easier to use but an issue is with the storing of timestamps for each requests

Sliding window counter algorithm:
A hybrid of fixed window and sliding window log

High-level architecture:
The basic idea of rate limiting algorithms is simple. At the high-level, we need a counter to
keep track of how many requests are sent from the same user, IP address, etc. If the counter is
larger than the limit, the request is disallowed.
Where shall we store counters? Using the database is not a good idea due to slowness of disk
access. In-memory cache is chosen because it is fast and supports time-based expiration
strategy. For instance, Redis [11] is a popular option to implement rate limiting. It is an in
memory store that offers two commands: INCR and EXPIRE
The client sends a request to rate limiting middleware.
• Rate limiting middleware fetches the counter from the corresponding bucket in Redis and

checks if the limit is reached or not.
 • If the limit is reached, the request is rejected.
 • If the limit is not reached, the request is sent to API servers. Meanwhile, the system
 increments the counter and saves it back to Redis.

Step 3 - Design deep dive
for here we have to answer two question?
- how to handle request that are rate limited
- How limiting rules are created

In case a request is rate limited, APIs return a HTTP response code 429 (too many requests)
 to the client. Depending on the use cases, we may enqueue the rate-limited requests to be
 processed later. For example, if some orders are rate limited due to system overload, we may
 keep those orders to be processed later

Client -> rate limitter middle war -> api server or -> redis or -> cached riles -> workers -> rules

The two challanges of building a rale limitter in a distributed system

Race Condition:
A race condition occurs when two or more threads, processes, or requests try to access or modify the same shared resource at the same time, and the final outcome depends on the timing of their execution. Locks are used to prevent this but locks generally slow your system

Synchronization:
one rate limitter might not be able to handle all traffic so having moe than one should be made possible so if one cant handle traffic, the other comes in and then they both need to be synchronized cause of statelessnaess
One possible solution is to use sticky sessions that allow a client to send traffic to the same
 rate limiter. This solution is not advisable because it is neither scalable nor flexible. A better
 approach is to use centralized data stores like Redis. The design is shown in Figure 4-16.

Performance optimization is another issue and first way of walking around it is by
- setting up multiple data centers
- Proper synchronization

After the rate limiter has been done, we jave to properly monitor to see if we can gather analytics data to check if the rate limiter is effective

CONSISTENT HASHING (Chapter 5):
Hash Problem:

This occurs when you use a hash function to distribute load but changes in the system cause imbalance or data loss.
Maybe when you add or remove servers, re-shard database, move cache partitions

Hashing is used in load balancer, caches, databases, cdn's and distributed systems

so now lets say we have 3 servers and our hash function is hash(id) % 3 each servers gets proper values but if we had a 4th server, this tends to break everything and manykeys tend to be redistributed or remapped

The solution is to use consistent hashing

How Big Companies Use It

Netflix → Uses consistent hashing to distribute movies across Open Connect servers.

YouTube → Uses consistent hashing for video storage.

Amazon DynamoDB → Uses consistent hashing for global scalability.

Facebook → Uses it in their caching infrastructure (TAO).

CONSISTENT HASHING:
In consistent hashing, only k/n keys need to be remapped where k is the number of keys and n is the number of slots

CHAPTER 6:DESIGN A KEY VALUE STORE
In this chapter, we design a key-value store that comprises of the following

characteristics:
• The size of a key-value pair is small: less than 10 KB.
• Ability to store big data.
• High availability: The system responds quickly, even during failures.
• High scalability: The system can be scaled to support large data set.
• Automatic scaling: The addition/deletion of servers should be automatic based on traffic.
• Tunable consistency.
• Low latency.

SINGLE SERVER KEY VALUE STORE
Developing a key-value store that resides in a single server is easy. An intuitive approach is
to store key-value pairs in a hash table, which keeps everything in memory. Even though
memory access is fast, fitting everything in memory may be impossible due to the space
constraint. Two optimizations can be done to fit more data in a single server:
• Data compression
• Store only frequently used data in memory and the rest on disk
Even with these optimizations, a single server can reach its capacity very quickly. A
distributed key-value store is required to support big data.

DISTRIBUTED KEY VALUE STORE
It uses distributed key value pairs across many servers. It makes use of the CAP theorem Consistency -> Availability -> Partition tolerance

Partition tolerance meaning how it can function if there are network partitions

Nowadays , just two of the CAP characteristics can be supported
CP,AP,CA

A CA System cant exist in the real word because there would surely be network failure of some sort

If we have three systems n1, n2 and n3. and n3 fails or has network issue, we must block all write requests to n1 and n2 cause if request move and want to pass n3.. it has network failure and that is impossible and can cause consistency issues

In this section,Versioning and vector locks we will discuss the following core components and techniques used to build a
key-value store:
• Data partition

• Data replication
• Consistency
• Inconsistency resolution
• Handling failures
• System architecture diagram
• Write path
• Read path

DATA PARTITION:
For large application, it is impossible to fit all data into one server so the simplest way to achieve this is by splitting the data into partitions and storing them but there are two issues

- The distribution of data across multiple servers
- How to minimize data movements when nodes are removed or added

This is where consistent hashing comes in place where servers and keys are places in a clock wise direction

DATA REPLICATION:
It is used to achieve high availability and reliability as data can be replicated over N servers
Nodes in the same data center often fail at the same time due to power outages so it is advisable to place replicas at distinct data centers and data centers are connected through high speed networks

CONSISTENCY
Quorum consensus ensures consistency by requiring overlapping replicas between reads and writes, and by tuning N, W, and R, systems can trade latency for consistency.

Eventual consistency model
Consistency model is other important factor to consider when designing a key-value store. A
consistency model defines the degree of data consistency, and a wide spectrum of possible
consistency models exist:
• Strong consistency: any read operation returns a value corresponding to the result of the
most updated write data item. A client never sees out-of-date data.
• Weak consistency: subsequent read operations may not see the most updated value.
• Eventual consistency: this is a specific form of weak consistency. Given enough time, all
updates are propagated, and all replicas are consistent.
Strong consistency is usually achieved by forcing a replica not to accept new

reads/writes
until every replica has agreed on current write. This approach is not ideal for highly available
systems because it could block new operations. Dynamo and Cassandra adopt eventual
consistency, which is our recommended consistency model for our key-value store

INCONSISTENCY RESOLUTION: VERSIONING:
Replication gives high availability but causes inconsistencies among replicas. are the two fixes for this. Versioning treats each data modification as new immutable versions of themselves.
Vector clocks track causal relationships between updates in a distributed system, allowing detection of concurrent writes so conflicts can be resolved without data loss.

Failure detection:
The best way of detecting failures in a distributed system is through the use of failure detection methods like gossip protocol:
Gossip protocol follows:

- Each node has a membership list that has member id, heartbeat and the rest
- Each node increments its heartbeat
- Once each node receives heartbeat, membership list is updated to the latest info
- If the heartbeat has not been increased, the member is considered offline

* Node s0 maintains a node membership list shown on the left side.
* Node s0 notices that node s2's (member ID = 2) heartbeat counter has not increased for a
* long time.
* Node s0 sends heartbeats that include s2's info to a set of random nodes. Once other
* nodes confirm that s2's heartbeat counter has not been updated for a long time, node s2 is
* marked down, and this information is propagated to other nodes.

Handling temporary failures:
After failures have been detected through the gossip protocol, the system needs to deploy
certain mechanisms to ensure availability. In the strict quorum approach, read and write
operations could be blocked as illustrated in the quorum consensus section.
A technique called "sloppy quorum" [4] is used to improve availability. Instead of enforcing

the quorum requirement, the system chooses the first W healthy servers for writes and first R
healthy servers for reads on the hash ring. Offline servers are ignored.

Handling permanent failures:

Handling data center outage
Data center outage could happen due to power outage, network outage, natural disaster, etc.
To build a system capable of handling data center outage, it is important to replicate data
across multiple data centers. Even if a data center is completely offline, users can still access
data through the other data centers

System architecture diagram:
Main features of the architecture are listed as follows:
• Clients communicate with the key-value store through simple APIs: get(key) and put(key,
value).
• A coordinator is a node that acts as a proxy between the client and the key-value store.
• Nodes are distributed on a ring using consistent hashing.
• The system is completely decentralized so adding and moving nodes can be automatic.
• Data is replicated at multiple nodes.
• There is no single point of failure as every node has the same set of responsibilities.

CHAPTER 7: DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS

Normal interview steps:
1- Ask Questions related to what you are asked to design
2- propose high level design
3- Design deep dive

Multiple options can be used to generate unique IDs in distributed systems. The options we
considered are:
• Multi-master replication
• Universally unique identifier (UUID)
• Ticket server
• Twitter snowflake approach

Multi master replication: The auto increment feature but instead of incrementing by 1, you increment by k where k is the number of database serves in use. The drawbacks include the fact that it is hard to scale and it also doesnt scale well

UUID:
A UUID is another easy way to obtain ID's. It is a 128 digit used to identify information in a computer system. Example of a UUID : 09c93e62-50b4-468d-bf8a-c07e1040bfb2. According to wikipedia, a UUID made now.

Generating UUID is simple. No coordination between servers is needed
UUID's are scalable because the web server is responsible for generating ID's they consume

TICKET SERVER:
A **Ticket Server** is a **centralized service** whose only job is:
Hand out unique, increasing numbers (tickets)
Every client that needs an ID:

- Sends a request to the ticket server
- Receives a unique ID
- Uses it as a primary key

```
Clients
  |
  v
Ticket Server
  |
  v
Database / Cache
```

It suffers from scalability and also single point of failure although it is easy to implement

TWITTER SNOWFLAKE APPROACH:
Twitter's Snowflake generates globally unique, time-ordered 64-bit IDs by combining
timestamp, machine ID, and sequence number, allowing distributed systems to scale without centralized coordination.

Snowflake uses a **machine ID**, so **each device (server)** generates IDs from its **own unique space**.

The Snowflake ID generator splits a 64-bit ID into timestamp, datacenter ID, machine ID, and sequence number. Datacenter and machine IDs are fixed at startup to ensure uniqueness across servers, while timestamps ensure time ordering. The sequence number allows multiple IDs to be generated within the same millisecond on a single machine. With 41 bits for timestamp and 12 bits for sequence, the system can generate millions of unique, time-sortable IDs per second without coordination..

CHAPTER 8: DESIGN A URL SHORTENER

The Golden Rule (MEMORIZE THIS)
Always estimate in this order:

1. Traffic
2. Storage
3. Bandwidth
4. Memory / Cache
5. Growth

If you follow this order, you'll never get lost. (This here is for back of the envelope estimation)

STEP 1:
First we ask questions about what the interviewer really wants from the interviewer and then start the bacd of the envelope estimation

Why do people want url shortener? **People want URL shorteners to make links easier to share, track, control, brand, and manage at scale.**

Back of the envelope estimation
• Write operation: 100 million URLs are generated per day.
• Write operation per second: 100 million / 24 /3600 = 1160
• Read operation: Assuming ratio of read operation to write operation is 10:1, read operation per second: 1160 * 10 = 11,600
• Assuming the URL shortener service will run for 10 years, this means we must support
100 million * 365 * 10 = 365 billion records.
• Assume average URL length is 100.
• Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB
It is important for you to walk through the assumptions and calculations with your interviewer so that both of you are on the same page.

STEP 2: THE HIGH-LEVEL DESCRIPTION
This section we will be talking about api endpoints, url redirecting and url shortening flows

API ENDPOINTS:
To create a new short URL we have to consider two things which is the url shortener and also the url redirecting which is redirecting the user to the long url site

URL REDIRECTING:
When a short url is inputed, it changes it to the long url with a 301 redirect status code or a 302 status code

A 301 status code means the redirected long url has been permanently cached so any request via the short url will automatically direct there.

A 302 status code means the redirected long url is temporarily kept there.

The best way for url redirecting is through the use of hash tables. The best way to implement a URL shortener is to use a **hash table** (or key-value store) to map short codes to long URLs. In production, this is often a combination of an **in-memory hash table for caching** and a **persistent database for storage**, with short codes generated either randomly or via a hash of the long URL.

URL SHORTENER;
Done with the use of a hash map

STEP 3: DESIGN DEEP DIVE
We will be talking about the data model, the shortener, the hash function, the hashing and the url shortening

DATA MODEL:
The high level design, everything is stored in a hash table but we can take a different approach by using <long url, shorturl>. The hash table method, the memory usage tend to be very expensive
At a conceptual level, a URL shortener is a hash table mapping short URLs to long URLs. However, storing everything in memory is not feasible at scale due to cost, durability, and scalability concerns. Therefore, the mappings are persisted in a relational database, while an in-memory hash table is used as a cache to speed up frequent lookups.

HASH FUNCTION:
A hash function is used to hash a long url to a short url also known as the hash value

Hash + collision resolution:
To shorten a long url, we should implement a hash function that hashes a long url to a 7 char string then encode in base 62 then you check the db in case of if there is any collision and this can be expensive.

This is when the production based hashing use by tiny url and the rest is used:
Base 62 hash:

While hash functions can generate short URLs by hashing the original URL, most production systems prefer generating unique IDs and encoding them in Base62 to avoid collisions and ensure scalability. Hashing is still widely used for caching and data distribution.

HASH + COLLISION RESOLUTION , BASE 62 CONVERSION
No unique id generator   unique id generator as it uses the master
replication,   twitter snowflake and the likes

Collision will likely occur
Fixed short url length

URL SHORTENING DEEP DIVE
1. longURL is the input.
2. The system checks if the longURL is in the database.
3. If it is, it means the longURL was converted to shortURL before. In this case, fetch the
shortURL from the database and return it to the client.
4. If not, the longURL is new. A new unique ID (primary key) Is generated by the unique
ID generator.
5. Convert the ID to shortURL with base 62 conversion.
6. Create a new database row with the ID, shortURL, and longURL.

The distributed id generator is worth mentioning cause its main function is to generate unique id which is used to create short urls

URL REDIRECTING DEEP DIVE:
A user clicks on a long url, the load balancer forwards the request to the server and then the server checks the cache if the long url has been returned directly. If the short url is not in the cache, fetch the long url from the database. If it is nor in the database, it is likely a user entered an invalid short url if it exist then the long url is retuned

```
Short URL
   ↓
API Gateway
   ↓
[ Redirection Service | Shortening Service ]
   ↓
Database
```

What the API Gateway does:

- Routing (where should this request go?)
- Rate limiting (prevent abuse)
- Authentication (if needed)
- Logging & monitoring
- Load balancing
- TLS termination (HTTPS)

Client
↓
API Gateway
↓
Redirection Service
↓
Cache (hit → return)
↓
DB (miss → cache it)
↓
HTTP 301 redirect

Client
↓
API Gateway
↓
Shortening Service
↓
ID Generator
↓
Database
↓
Return short URL

Analytics service will go to the in memory count

CHAPTER 9: DESIGN A WEB CRAWLER:
A web crawler is known as a robot or spider. It is widely used by search engines to discover new content that they can upload

It can be used for many purposes:

- Search engine indexing by collecting local indexes for search engines

- Web archiving for collecting resources and then storing them for later purposes
- Web mining for searching and getting resources on the web
- Web monitoring to monitor copyright and the like

HIGH LEVEL DESIGN OF A WEB CRAWLER:
SEED URLS:
A web crawler uses seed URL's as a starting point for the crawl process. Mainly popular websites that can also link to other smaller ones

URL FRONTIERS:
Most modern web crawlers divide the crawl state into two, the one to be downloaded and the one already downloaded

HTML DOWNLOADER:
It downloads web pages from the internet. The urls are provided by the url frontiers

DNS RESOLVER:
The dns resolver provides the IP address that will be needed for the html downloader to download the page

CONTENT PARSER:
After a webpage has been downloaded, it must be parsed and validated because malformed web pages will provide storage and web issues

CONTENT SEEN:
After a webpage is downloaded, we do get an issue of duplicated content so the content seen helps us to eliminate data redundancy and shorten processing time. It also helps detect new content previously stored in the system

CONTENT STORAGE:
It is a storage system for storing html content. The choice of storage depends on data type, data size, access frequency e.t.c

URL EXTRACTOR:
It parses and extract links from html pages

URL Filter
The URL filter excludes certain content types, file extensions, error links and URLs in
"blacklisted" sites.
URL Seen?
"URL Seen?" is a data structure that keeps track of URLs that are visited before or

already in
the Frontier.
“URL Seen?” helps to avoid adding the same URL multiple times as this can increase server load and cause potential infinite loops.
Bloom filter and hash table are common techniques to implement the “URL Seen?” component. We will not cover the detailed implementation of the bloom filter and hash table
here. For more information, refer to the reference materials [4] [8].

URL Storage
URL Storage stores already visited URLs
Step 1: Add seed URLs to the URL Frontier
Step 2: HTML Downloader fetches a list of URLs from URL Frontier.
Step 3: HTML Downloader gets IP addresses of URLs from DNS resolver and starts downloading.
Step 4: Content Parser parses HTML pages and checks if pages are malformed.
Step 5: After content is parsed and validated, it is passed to the “Content Seen?” component.
Step 6: “Content Seen” component checks if a HTML page is already in the storage.
• If it is in the storage, this means the same content in a different URL has already been
processed. In this case, the HTML page is discarded.
• If it is not in the storage, the system has not processed the same content before. The
content is passed to Link Extractor.
Step 7: Link extractor extracts links from HTML pages.
Step 8: Extracted links are passed to the URL filter.
Step 9: After links are filtered, they are passed to the “URL Seen?” component.
Step 10: “URL Seen” component checks if a URL is already in the storage, if yes, it is
processed before, and nothing needs to be done.
Step 11: If a URL has not been processed before, it is added to the URL Frontier.

URL → Download → Parse → Check → Extract links → Add new URLs → Repeat

DESIGN DEEP DIVE
The major components for us to design are

- DFS VS BFS
- URL FRONTIER
- ROBUSTNESS

- EXTENSIBILITY
- DETECT AND AVOID PROBLEMATIC CONTENT

DFS VS BFS:
We make use of bfs and not dfs cause dfs is deep and we need to access how wide and then we make use of queues in a FIFO nature. The problem with dis and bfs is that suppose our crawler is on wikipedia and it has like 50 links then all our internal links will be to wikipedia, you add all 50 links at the end of the queue so now the queue has mostly wikipedia links

Imagine you're mailing letters to different friends:

- You have 50 letters for Alice, 3 for Bob, 2 for Charlie
- If you send all Alice's letters first, she gets spammed
- Better → send **1 to Alice, 1 to Bob, 1 to Charlie**, then repeat

URL FRONTIER:
It helps to address the problems

Priority Queue Diagram
Idea: Crawl more important pages first.

```
    URL Frontier (Priority Queue)

 ┌──────────────────────────────┬─────────────────────────────┬──────────────────────────
───┐
 │ wikipedia │ cnn.com   │ blog.xyz  │
 │  High     │Medium     │Low        │

 └──────────────────────────────┴─────────────────────────────┴──────────────────────────
───┘
        │
        ▼
    Crawler Worker
        │
        ▼
  Fetch & Parse Page
```

Key:

- High priority URLs (like Wikipedia) get crawled **first**
- Low priority URLs wait in queue

2️⃣ Freshness Diagram

Idea: Crawl pages more frequently if they change often.

```
  URL Frontier (Freshness Scheduling)
```

```
 ┌──────────────────────────┐     ┌──────────────────────────┐
| news.com    | --> | crawl every 5 min |
| blog.xyz    | --> | crawl every 24 hr|
| wiki.org    | --> | crawl every 12 hr|
 └──────────────────────────┘
        |
        ▼
    Crawler Worker
        |
        ▼
    Update Storage
```

Key:

- Fresh content → short crawl interval
- Less-changing content → long interval

3️⃣ Politeness Diagram

Idea: Avoid hitting a single host too frequently.

```
    URL Frontier (Host-based Queues)

 ┌──────────────────────────┐  ┌──────────────────────────┐
 ┌──────────────────────────┐
| wikipedia.com|  | cnn.com     |  | espn.com    |
| [URL1,URL2] |  | [URL3,URL4] |  | [URL5,URL6] |
 └───────────┬──────────────┘  └───────────┬──────────────┘
 └───────────┬──────────────┘
      |          |          |
      ▼          ▼          ▼
   Crawler Worker (1 URL per host at a time)
      |
      ▼
   Fetch & Parse
```

Key:

- Rotate hosts → avoid flooding any single site
- Respect crawl delays (robots.txt)

HTML DOWNLOADER

- robots.txt is a **text file placed at the root of a website** to tell web crawlers where they can and cannot
- Performance optimization: To achieve this, crawl jobs are distributed into multiple servers and each server runs multiple threads
- Cache DNS Resolver DNS Resolver is a bottleneck for crawlers because DNS requests might take time due to the synchronous nature of many DNS interfaces. DNS response time ranges from 10ms to 200ms. Once a request to DNS is carried out by a crawler thread, other threads are

blocked until the first request is completed. Maintaining our DNS cache to avoid calling DNS frequently is an effective technique for speed optimization. Our DNS cache keeps the domain name to IP address mapping and is updated periodically by cron jobs.

- Locality: Distribute crawl servers geographically. When crawl servers are closer to website hosts, crawlers experience faster download time. Design locality applies to most of the system components: crawl servers, cache, queue, storage, etc.
- Short timeoutSome web servers respond slowly or may not respond at all. To avoid long wait time, a maximal wait time is specified. If a host does not respond within a predefined time, the crawler will stop the job and crawl some other pages.

Robustness

Besides performance optimization, robustness is also an important consideration. We present
a few approaches to improve the system robustness:
• Consistent hashing: This helps to distribute loads among downloaders.
• Save crawl states and data: To guard against failures, crawl states and data are written to
a storage system. A disrupted crawl can be restarted easily by loading saved states and
data.
• Exception handling: Errors are inevitable and common in a large-scale system.
The crawler must handle exceptions gracefully without crashing the system.
• Data validation: This is an important measure to prevent system errors.

Extensibility

As almost every system evolves, one of the design goals is to make the system flexible
enough to support new content types. The crawler can be extended by plugging in new
modules.
• PNG Downloader module is plugged-in to download PNG files.
• Web Monitor module is added to monitor the web and prevent copyright and trademark
infringements.

Detect and avoid problematic content

This section discusses the detection and prevention of redundant, meaningless, or harmful
content.
1. Redundant content

As discussed previously, nearly 30% of the web pages are duplicates. Hashes or checksums
help to detect duplication [11].
2. Spider traps
A spider trap is a web page that causes a crawler in an infinite loop. For instance, an infinite
deep directory structure is listed as follows:
www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/…
Such spider traps can be avoided by setting a maximal length for URLs. However, no one-
size-fits-all solution exists to detect spider traps. Websites containing spider traps are easy to
identify due to an unusually large number of web pages discovered on such websites. It is
hard to develop automatic algorithms to avoid spider traps; however, a user can
manuallyverify and identify a spider trap, and either exclude those websites from the crawler or apply
some customized URL filters.
3. Data noise
Some of the contents have little or no value, such as advertisements, code snippets, spam
URLs, etc. Those contents are not useful for crawlers and should be excluded if possible.

Dont use the same host every time
Prioritizer moves high value url to the front of the queue

CHAPTER 10: DESIGN A NOTIFICATION SYSTEM:
First step will be to ask questions relating to what is being requested for by the interviewer

Step 2- High level design and buy in:
We look at the different types of notifications, the contact info gathering and then the notification sending and receiving

iOS push notifications:
Provider -> APNS -> Client device

iOS push notifications:
It uses the same flow but the APNS is replaced by firebase cloud messaging.

SMS notification:
Same flow but sms service replace the APNS

Email notification:
Same flow but email service replace the APNS

Contact info gathering flow:
This occurs when the user installs an app and then the api collects the phone number and other details and store in the database

User -> load balancer -> api server -> Database

High level device:
Service -> notification system -> Third party system -> Client device

A service can be a micro-service, a cron job, or a distributed system that triggers notification sending events. For example, a billing service sends emails to remind
customers of their due payment or a shopping website tells customers that their packages will be delivered tomorrow via SMS messages.

Notification system: The notification system is the centerpiece of sending/ receiving
notifications. Starting with something simple, only one notification server is used. It provides
APIs for services 1 to N, and builds notification payloads for third party services.

Third-party services: Third party services are responsible for delivering notifications to
users.

The problems associated with this include
Single point of failure
Performance bottleneck
Hard to scale

In order to fix this we improve the high level design we
Move the database and cache out of the notification server.
• Add more notification servers and set up automatic horizontal scaling.
• Introduce message queues to decouple the system components

Next, let us examine how every component works together to send a notification:
1. A service calls APIs provided by notification servers to send notifications.

2. Notification servers fetch metadata such as user info, device token, and notification
setting from the cache or database.
3. A notification event is sent to the corresponding queue for processing. For instance, an
iOS push notification event is sent to the iOS PN queue.
4. Workers pull notification events from message queues.
5. Workers send notifications to third party services.
6. Third-party services send notifications to user devices.

Besides the high-level design, we dug deep into more components and optimizations.
• Reliability: We proposed a robust retry mechanism to minimize the failure rate.
• Security: AppKey/appSecret pair is used to ensure only verified clients can send notifications.
• Tracking and monitoring: These are implemented in any stage of a notification flow to
capture important stats.
• Respect user settings: Users may opt-out of receiving notifications. Our system checks
user settings first before sending notifications.
• Rate limiting: Users will appreciate a frequency capping on the number of notifications
they receive.

```
Client / Service
      |
      ▼
 Notification API
      |
      ▼
 Message Queue (Kafka / SQS / RabbitMQ)
      |
      ▼
 Notification Workers
      |
  ┌─────────┴──────────┬────────────┬──────────────┐
 ▼          ▼      ▼      ▼
Push     Email    SMS    In-App
Service  Service  Service Service


Or Client Services
       |
      ▼
   Auth Service
      |
      ▼
 Notification API
      |
      ▼
   Message Queue
      |
      ▼
 Notification Workers
      |
  ┌──────────┬──────────┬────────────┬─────────────┐
  ▼          ▼    ▼    ▼
Template   Push  Email   SMS
Service   Service Service Service
```

DESIGN A CHAT SYSTEM:
Initial questions to be asked includes questions revolving around what message to send, water it is end to end encryption, mobile or web app, type of message just to know what kind of features to add

HIGH LEVEL DESIGN:

SENDER -> CHAT SERVICE (STORE MESSAGE , RELAY MESSAGE) -> RECEIVER

Time tested http response which is what the illustration above shows : "It refers to the traditional stateless HTTP request–response model that has been widely used and proven at internet scale. It is reliable, easy to scale, cache-friendly, and works well with load balancers and CDNs."

Think of HTTP as:
A reliable postal service
You send a letter, you get a reply. No open phone line

POLLING:
It is a technique where a client periodically asks a server if there are messages available.
But this consumes server resources because why ask a question that might always be no

LONG POLLING:
A long polling is when a client holds the connection open until there are actually

new messages available or a time out threshold is met
- Sender and receiver may not connect to the same chat server

WEB SOCKETS:
The most common solution for sending asynchronous updates from server to client. It is bidirectional and can also be upgraded via some well defined handshake.

The high level design is divided into stateless and stateful:

STATELESS:
User -> load balancer -> (service discovery, auth service, group management, user profile)

STATEFULL:
User 1 and User 2 —> Chat service with a third party integration handling the push notifications

Stateless service are basically the ones behind a load balancer such as little features like login, signup, user profile and the rest.

Stateful service is the chat service. The service is stateful because each client maintains a persistent network connection to a chat server

Third-party integration
For a chat app, push notification is the most important third-party integration. It is a way to
inform users when new messages have arrived, even when the app is not running. Proper
integration of push notification is crucial. Refer to Chapter 10 Design a notification system
for more information

STORAGE:
In a chat system, we have to make the right decision on what kind of database we want and read write patterns.
There are two types of data in a chat system and he first is generic such as user profile and the likes but the second is the unique chat system data and chat history and chat related things. We recommend using a key - value store to store data so we can allow for horizontal scaling, and it provides a low latency to access data

messages table for 1 to 1 chat
----

message_id (PK)
conversation_id (FK)
sender_id
content
message_type   -- text, image, file, etc.
status   -- sent, delivered, read
created_at

\messages table (shared for 1-on-1 & group)
💡 Best practice: use one messages table

messages
----
message_id (PK)
conversation_id
sender_id
content
message_type
created_at

DESIGN DEEP DIVE
Service discovery
User A tries to log in to the app.
2. The load balancer sends the login request to API servers.
3. After the backend authenticates the user, service discovery finds the best chat server for
User A. In this example, server 2 is chosen and the server info is returned back to User A.
4. User A connects to chat server 2 through WebSocket.

1 ON 1 CHAT FLOW:
User send a message, the chat server picks it up and generate an id for the message from the id generator, the message is then sent to the sync queue which stores the message in a key - value store . If user b is online, the message is sent to the server user b is connected to else , a push notification is sent . This shows how web socket work

"Each device maintains its own cur_max_message_id to track the latest message it has received, allowing it to fetch only new messages from the key-value store and synchronize independently across multiple devices without duplicates."

This is a situation where a user has multiple devices

Small group chat flow
“In a group chat, each message sent by a user is copied into the message queues of all group members, ensuring that each member receives the message independently and can process or store it from their own queue.”

User login
The user login flow is explained in the “Service Discovery” section. After a WebSocket
connection is built between the client and the real-time service, user A’s online status and
last_active_at timestamp are saved in the KV store. Presence indicator shows the user is
online after she logs in.

User logout
When a user logs out, it goes through the user logout flow as shown in Figure 12-17. The
online status is changed to offline in the KV store. The presence indicator shows a user is
offline.

A heartbeat mechanism is used to detect user disconnections by having each client periodically send a small ‘I’m alive’ message to the server, allowing the system to update user presence and clean up resources if a client goes offline unexpectedly.

For online status, we use a publish subscribe model
When User A’s online status changes, it publishes the event to three channels,
channel A-B, A-C, and A-D. Those three channels are subscribed by User B, C, and D,
respectively. Thus, it is easy for friends to get online status updates. The communication
between clients and servers is through real-time WebSocket.

User A -> presence servers -> subscribe -> user b ,cod

Caching
Error handling
Improving load time
End to end encryption

And the likes

```
┌─────────────────────┐
│  Mobile Client   │
│ (Sender / Receiver│
│  iOS / Android)   │
└──────────┬──────────┘
      │  (Persistent TCP / WebSocket)
      ▼
┌─────────────────────┐
│  Load Balancer   │
│ (L4 / L7 Routing) │
└──────────┬──────────┘
      ▼
┌────────────────────────────────┐
│      Chat Servers          │
│  - Maintain user sessions   │
│  - Handle presence (online) │
│  - Route messages          │
└───────┬───────────┬──────────┘
     │     │
     │     │
     ▼     ▼
┌────────────────┐  ┌────────────────────┐
│  Message Queue│   │ Presence Service  │
│ (Kafka-like)  │   │ - Online/Offline │
│ - Buffer msgs │   │ - Heartbeats     │
└───────┬────────┘  └────────────────────┘
     │
     ▼
┌────────────────────────────────┐
│     Message Processor       │
│ - Dequeue messages          │
│ - Check recipient status     │
│ - Ensure ordering           │
└───────┬───────────┬──────────┘
     │     │
     │     │
     ▼     ▼
┌─────────────────┐
┌───────────────────────┐
│ Chat Servers   │  │ Notification Service │
│ (Recipient)   │ │ (APNs / FCM)          │
```

```
| - Push message|  | - Alert offline users|
 └────────────────┬────────────────┘
 └──────────────────────────────────────────────┘
       |
       ▼
 ┌──────────────────────────────────────────────────────┐
|   Mobile Client (Recv)   |
| - Decrypt message        |
| - Send delivery ack      |
 └──────────────────────────────────────────────────────┘
```

CHAPTER 13 - DESIGN A SYSTEM AUTOCOMPLETE SYSTEM:
Understand the problem and establish the design scope: then do the back of the envelope estimation.

For this, we ask about the auto complete feature, the scalable and the high availability and it the results returned must be sorted or not, the qm=amount of users and the speed

HIGH LEVEL DESIGN :
At the high level, the system is divided into the data gathering and also the query service

Data gathering service: emphasizes on how user input queries and aggregates them in real time. It shows a frequency table and the frequency table should look like with it containing a term and the frequency at which it appears

Query service:
The query service contain the query to be stored and the frequency at which it occurs

DESIGN DEEP DIVE:
Trie Data Structure
Data gathering service
Query Service
Scale the storge
Trie Operations

User Search
  ↓
API Gateway
  ↓

Data Gathering Service
  ↓
Message Queue (Kafka / PubSub)
  ↓
Aggregation + Processing
  ↓
Trie Builder / Updater
  ↓
Autocomplete Service

Trie Data Structure:
A trie is a tree like data structure that can comparatively store strings=, it basically involves a single word or s prefix string. Autocomplete with a trie works by **walking the trie using the typed prefix**, then **returning all words that exist below that point**.

```
(root)
 └─── t
      └─── h
           ├─── i
           │    └─── s (✔)
           │         └───  (space)
           │              ├─── j
           │              │    └─── o → b (✔)
           │              │         └─── l → l → o → f (✔)
           └─── a
                └─── t (✔)
```

“To generate autocomplete suggestions for a prefix, the system locates the corresponding trie node, traverses its subtree to collect valid words, ranks them by frequency, and returns the top K results.”

The above algorithm is straightforward. However, it is too slow because we need to traverse
the entire trie to get top k results in the worst-case scenario. Below are two optimizations:
1. Limit the max length of a prefix
2. Cache top search queries at each node

Data gathering service:
In our previous design, A user types a search query, data is updated in real time. The data gathering service **collects, aggregates, and updates search queries**

and their frequencies, which are then used to build and update the autocomplete trie.

Query service
User sends a search query
2️⃣ **Load balancer** routes request to an autocomplete server
3️⃣ Server checks **cache** for suggestions
4️⃣ If cache hit → return suggestions immediately
5️⃣ If cache miss → fetch from **trie / index store**, update cache, return result
6️⃣ **Separately**, the query is logged and sent to the **aggregator pipeline**
7️⃣ Aggregator updates frequencies and improves future suggestions

Caches and Ajax requests

Trie Operations:
Update the trie weekly. Once a new trie is created, the new trie replaces the old one.
Update the trie node directly

Update, delete and create operations

Autocomplete systems handle user queries by sending **AJAX requests** from the client to the server, where the system first checks a **cache** (e.g., Redis) for top suggestions to ensure low-latency responses. If the cache misses, the server queries the **trie index** and updates the cache accordingly. The **trie**, which stores all queries character by character, can be updated either **weekly** by rebuilding the entire trie and atomically replacing the old one, or **directly in-place** by inserting new queries, updating frequencies, or deleting outdated entries. To handle the memory-intensive nature of tries, the system uses techniques such as **sharding by prefix**, **compressed tries**, or keeping only hot prefixes in memory while storing cold prefixes on disk. All query events are also sent asynchronously to a **data gathering and aggregation pipeline**, which calculates query frequencies and ranking scores to update the trie over time. This combination of caching, AJAX requests, trie operations, and scalable storage ensures the autocomplete system is both **fast and capable of handling millions of users and queries efficiently**.

Or "Autocomplete systems handle user queries via AJAX requests that first check a cache for top suggestions, ensuring low-latency responses. If the cache misses, the trie index is queried, which can be updated weekly with a full rebuild or in real-time by inserting, updating, or deleting nodes. To scale memory usage, tries can be sharded, compressed, or store only hot prefixes in memory. Query events are asynchronously aggregated to update frequencies and ranking, making the system fast, scalable, and responsive to trending searches.

CHAPTER 14: DESIGN YOUTUBE:

Basically, the first set of things would be to understand what we are going to do such as feature and other important things and this is where understanding the design scope comes in place. We have to establish the feature needed (uploading of videos), liking, commenting , sharing videos, the video quality encryption, authentication, video size and the limit to what video can be uploaded, video streaming and others

Back of the envelope estimation
The following estimations are based on many assumptions, so it is important to communicate
with the interviewer to make sure she is on the same page.
• Assume the product has 5 million daily active users (DAU).
• Users watch 5 videos per day.
• 10% of users upload 1 video per day.
• Assume the average video size is 300 MB.
• Total daily storage space needed: 5 million * 10% * 300 MB = 150TB
• CDN cost.
• When cloud CDN serves a video, you are charged for data transferred out of the CDN.
• Let us use Amazon's CDN CloudFront for cost estimation (Figure 14-2) [3]. Assume
100% of traffic is served from the United States. The average cost per GB is $0.02.
For simplicity, we only calculate the cost of video streaming.
• 5 million * 5 videos * 0.3GB * $0.02 = $150,000 per day.

Propose high level design and get buy: (STEP TWO):
At the highest level, the client is connected to can and api servers

Client -> one who is using YouTube
CDN -> where videos are stored
API Servers ->everything except video streaming goes through here

User -> load balancer

-Videos are uploaded to the original storage
-Transcoding servers fetch videos from the original storage and start transcoding
"YouTube uploads videos to original storage, transcodes them asynchronously into

multiple formats, distributes them via CDN, updates metadata through event-driven workers, and finally notifies the user when the video is ready to stream."

Video streaming flow:
"A video streaming system uploads videos to storage, transcodes them asynchronously into multiple bitrates, distributes them through CDNs, and streams them using HTTP-based adaptive protocols like HLS or DASH for low-latency and scalable playback."

Design Deep Dive:
Video transcoding is the process of converting a video from one format, codec, resolution, or bitrate into multiple versions so it can be efficiently streamed across **different devices, network conditions, and screen sizes**.

Video transcoding architecture
The architecture has six main components: preprocessor, DAG scheduler, resource manager,
task workers, temporary storage, and encoded video as the output. Let us take a close look at
each component.
The video upload flow starts with client authentication and metadata creation, followed by direct upload to object storage. Upload completion triggers asynchronous transcoding via a message queue, after which transcoded videos are stored, distributed via CDN, and made available for streaming

DESIGN GOOGLE DRIVE:
1- We understand what we are to build and ask questions related to it

Client
  ↓
API / Metadata Server
  ↓
Block Servers (Chunk Storage)
  ↓
Replication & Backup

High level design:
APIs
What do the APIs look like? We primary need 3 APIs: upload a file, download a file, and get
file revisions.

After putting files in S3, you can finally have a good night's sleep without worrying

about
data losses. To stop similar problems from happening in the future, you decide to do further
research on areas you can improve. Here are a few areas you find:
• Load balancer: Add a load balancer to distribute network traffic. A load balancer ensures
evenly distributed traffic, and if a web server goes down, it will redistribute the traffic.
• Web servers: After a load balancer is added, more web servers can be added/ removed
easily, depending on the traffic load.
• Metadata database: Move the database out of the server to avoid single point of failure.
In the meantime, set up data replication and sharding to meet the availability and scalability requirements.
• File storage: Amazon S3 is used for file storage. To ensure availability and durability,
files are replicated in two separate geographical regions.
After applying the above improvements, you have successfully decoupled web servers,
metadata database, and file storage from a single server.

Design Deep Dive:

In Google Drive, block servers store files as fixed-size immutable chunks, enabling efficient uploads, deduplication, fault tolerance, and parallel data transfer. Files are logically represented as ordered block lists managed by metadata servers, allowing partial updates, resumable uploads, and scalable storage across distributed systems.

This system requires **strong consistency**, meaning a file must never appear differently across clients at the same time. To achieve this, strong consistency is enforced at the **metadata database and cache layers**. Since in-memory caches are eventually consistent by default, we ensure consistency by keeping cache replicas aligned with the master and **invalidating cache entries on every database write**. A **relational database** is chosen for metadata storage because it natively supports **ACID guarantees**, unlike most NoSQL databases, which would require custom synchronization logic.
The **metadata database** maintains core system state. Users own namespaces (root directories), files live inside namespaces, and each file has immutable version records. Files are stored as ordered collections of blocks, allowing reconstruction of any version. Devices are tracked to support multi-device sync and push notifications.
During the **upload flow**, the client sends two parallel requests: one to register file

metadata and another to upload file content. Metadata is stored first with a “pending” status, and other clients are notified that an upload is in progress. Meanwhile, block servers chunk, compress, encrypt, and upload file blocks to cloud storage. Once upload completes, a callback updates the file status to “uploaded,” and clients are notified that the file is ready.

The **download flow** is triggered when clients are notified of file changes (via long polling) or when offline clients reconnect. The client first fetches updated metadata, then downloads required blocks from block servers, which retrieve data from cloud storage if needed, allowing the client to reconstruct the file locally.

A **notification service** ensures file consistency across devices. It uses **long polling** instead of WebSockets because communication is one-way and infrequent. Clients keep an open connection and are notified only when changes occur, prompting them to pull the latest metadata.

To **save storage space**, the system deduplicates blocks using hashes, limits stored file versions, prioritizes recent versions, and moves infrequently accessed data to **cold storage**. This significantly reduces storage cost while preserving reliability and version history.

Finally, the system is designed for **fault tolerance**. Load balancers, API servers, block servers, caches, databases, queues, and notification servers are all replicated. Failures are handled through leader promotion, traffic rerouting, re-replication, and client reconnection strategies, ensuring high availability and data durability.