After passing the Amazon L6 TPM interview, I decided to put together a document helping candidates prep for the notorious system design interview and provide some insight into how I decoded it.

System design interviews can be intimidating. They are designed as open-ended questions and force candidates to take charge and lead the discussion with very little feedback or guidance from the interviewer. Even after studying the underlying technologies and technical aspects of scaling distributed systems, I found it hard to navigate this type of questions and quickly lost confidence in my ability to complete a system design in an interview setting.

When I analyzed my performance, I found that the main issues were: not knowing which parts were most important to focus on during the discussion, the challenge of keeping track of the tasks ahead, providing the right level of details in each stage of the interview and managing time.

To address these areas, I collected and compiled a list of best-known methods, structures and design patterns for system design interviews. These BKMs significantly improved my performance, helped boost my confidence and eventually got me a FAANG offer with limited system design background. In this document, I share the structures and BKMs I used in hopes of helping others ace their system design interviews as well!

To make it easier to read, I will be using an example of designing **Facebook** throughout this document.

Disclaimer: This is what worked for me and what I have received positive feedback from my interviews and mock sessions with other candidates. It is not meant to be an exhaustive list of system design methodologies and will not replace studying the underlying distributed systems technologies. Use this in addition to what you already know and hopefully you'll do great in your system design interview.

Feel free to reach out to me with feedback, questions or request for system design mock interviews!
mock.design.interview@gmail.com

Good luck!

Time Management

Time management is an essential factor one should be aware of when working on a system design question during an interview. FAANG interviews are typically 45-60 minutes long, and 10-15 minutes will be spent on introductions and behavioral questions before going into the design section. This leaves 30-40 minutes for us to resolve a complex system design question which has most likely taken numerous teams of talented engineers years to come up with (think: Instagram or Twitter).

Many candidates having taken *Grokking The System Design* or similar online courses believe they should recreate a similar level of depth in their system design interview. In the interview, they would start and perform elaborate capacity estimations and design beautiful and robust database schema only to find that they have no time left to discuss the actual backend design the interviewers were looking for, and fail the interview.

Remember, the interviewer will not hurry you through the steps! **You are in the lead and must manage your own time.**

Now that we understand that time limitation will be a challenge, let's divide our time into sections. I will provide detailed instructions on how to approach each of these sections below.

- **Functional and Non-functional Requirements** – 8-10 minutes
- **Capacity Estimations** – 2-3 minutes
- **API Definition** – 2-3 minutes
- **High Level Design** – 10 minutes
- **Low Level Design** – 10-15 minutes

Functional and Non-functional Requirements – 8-10 minutes

Functional Requirements (FRs):

Getting all of the FRs and getting them right are very important factors in your success in the interview. There are no magic solutions here, you will need to be well versed in the basic list of FRs for each of the systems listed in *Grokking the System Design*.

Expect to have 3-6 FRs. Start from the obvious ones (e.g., Facebook: Friends, Feed, post to feed) and try to suggest a couple of advanced ones (e.g., Facebook post ranking).

It would be extremely challenging to design a system that has more than 5-6 FRs within the timeframe of a system design interview. If your interviewer pushes for more, write them down and let them know that you may need to prioritize requirements in the design phase.
Listen to what the interviewer is telling you and make sure you completely understand what they say. If not, ask for clarifications. Getting an FR wrong is a sure way to design the wrong system and possibly fail the interview.

Example:

- User should be able to post to their feed (text only)
- User should be able to follow other users/friends
- User logs in and views their feed, posts shown in reverse chronological order

Non-FRs:

Non-FRs are typically standards and you should target getting through them very quickly. Here is the list I start with for almost every question:

1. **System must be highly available –** ever met anyone who wants downtime for their service?
2. **We are ok with eventual consistency to ensure high availability** (see CAP theorem) - except for financial or finite inventory type systems where we must ensure data consistency!
3. **System should be durable**- once data is persisted, it should never be lost
4. **System should have low latency -** assume 100-200ms
5. **System should scale** – ask your interviewer if they plan to scale and congratulate them on their upcoming financial success once they say it will 😊

Pointers:

- Good FRs are crucial for a successful system design. Invest time in practicing how to elicit FRs. Mock interviews are a great way of achieving proficiency here.
- Know the leading 2-3 FRs of each of the typical system design interview questions covered in *Grokking the System Design*.
- Propose 2-3 additional FRs and try to have an open conversation with your interviewer on what is expected of your system.

Capacity Estimations – 2-3 minutes

2-3 minutes?! You must be joking! *Grokking the System Design* spends whole chapters on this! Yes, they do but if we do the same thing, we'll never get to design a system...

The main purpose of this section is to lead us to an understanding whether this is a **read** or **write** heavy system. We need to show the interviewer that we understand how to calculate the volume of queries per second but should do this quickly.

Example:

Interviewee: how many DAU does my Facebook system have?

Interviewer: 200M [this is usually all you get]

Interviewee: ok, I will assume each user logs into their feed 5 times and one in 5 users posts per day:

- Reads per sec: 200M*5 = 1B (reads day) / 100K (approx. seconds per day)= 10K queries/sec
- Writes per sec: 40M posts per day / 100K (approx. seconds per day) = 400 writes/sec
- Storage: 40M * XB per post = X GB per day
- Assuming 5 year data retention we'd need (5*365*X) + 30% buffer = Y TB

Interviewee: so, we have a **read heavy** system on our hands. We will have to make sure to support this load in our front and backend infrastructure. We will also need to allocate X GB storage per day and Y TB of storage for data retention and projected growth.

Pointers:

- Unless you are taking a product design interview (Facebook), capacity and storge estimation is only a step in the way to getting to designing the system.
- This is NOT a math test, use a calculator! I can't tell you how many people try to impress their interviewers by doing math in their head and getting it wrong in the end.
- Declare whether this is a read or write heavy system.
- Move on quicky.
- Over time, I have compiled a list of system design questions and capacity estimations. I then put this a table and created a cheat sheet. If you are reading this in 2021, chances are that you are interviewing in remote. Why not use this to your advantage and have cheat sheets ready?

Site	#users overall	#daily users	Activity per user (day)	Storage assumptions	Daily storage	Long term storage	Bandwidth
TinyURL							
Pastebin							
Twitter	1B	200M	post: 0.5 Tweets, 5 likes/favs: 0.5*200M = 100M new tweets per day view: 2 timeline visits, 5 other users w/ 20 tweets each: (2 timeline+5 users)* 20 * 200M = 28B	- AVG text tweet 140char+30byte meta data (user name, etc): 2 byte*140 + 30 =310 byte - 1/5 send 200KB pic - 1/10 send 2MB vid	Text tweets: 200M*0.5 tweets* 310 byte = 30GB/day Pics: (100M/5)*200KB = 4TB Vids: (100M/10)*2MB = 20 TB	5 years: 25TB * 365 * 5 = 45PB	In: 24TB / 24/3600 = 280 MB/s Out/reads: text: 28B*280byte/24/3600=96MB/s pics (show all): 28B*2MB*86400=13GB/s vids (show 1/3): 28B*2MB/86400=22GB/s total: 35GB/s
Facebook messanger		500M	40 messages per day				
Yelp	500M places +20% growth yearly		100K queries per second +20% growth yearly				
Webcrawler	1) crawl 1B sites 2) upper bound of 15B webpages to crawl			1) time to complete the crawl: 4wks 2) Pages in a sec: 15B/30days/86400sec = ~5800 pages/sec 3) storage requirements: assuming HTML only with an avg size of 100KB + 500byte metadata: 15B*(100KB+500byte) =1.5PB		70% capacity model (dont exceed 70%): 1.3*1.5PB = 2.14PB	

API Definition – 2-3 minutes

Assuming this is not an API design question, you should take a couple of minutes (but not much more) to design the APIs now. This helps in the HLD section which comes next.
Some people are more comfortable designing the APIs along with the HLD. I believe this is acceptable as long as you remember to do so.

I don't recommend trying to get each and every field and piece of data to be sent in the API. Instead, it may be better focus on the most important data which needs to be used by the system:

- **User verification and authentication**
    - user_id, API_Dev_Key (rate limiting)
- **Business logic** – data used by our backend service to apply logic or add value to the content being served back
    - Location, last_login, search_term
- **Data to be persisted** – important information we need to store in our data stores as is
    - Text, description, like, comment, metadata for media, friend_id, etc.

If you want to make sure your interviewer isn't expecting more, you could say "we could also pass some additional information such as 1,2,3 but for the sake of time, we will define the important ones".

Example:

1. Post_to_feed (API_Dev_Key, user_id, post_text, user_location)
    a. Returns: Bool
2. Get_feed(API_Dev_Key, user_id, last_login, number_of_results)
    a. Returns: JSON
3. Add_Friend(API_Dev_Key, user_id, follow_user_id)
    a. Returns: Bool

Pointers:

- API_Dev_Key – call out that it's used for monitoring and rate limiting

High Level Design – 10 minutes

This is where things get interesting... More often than not, candidates start with a HLD and somewhere along the way dive deep into LLD considerations and lose track of time and requirements.

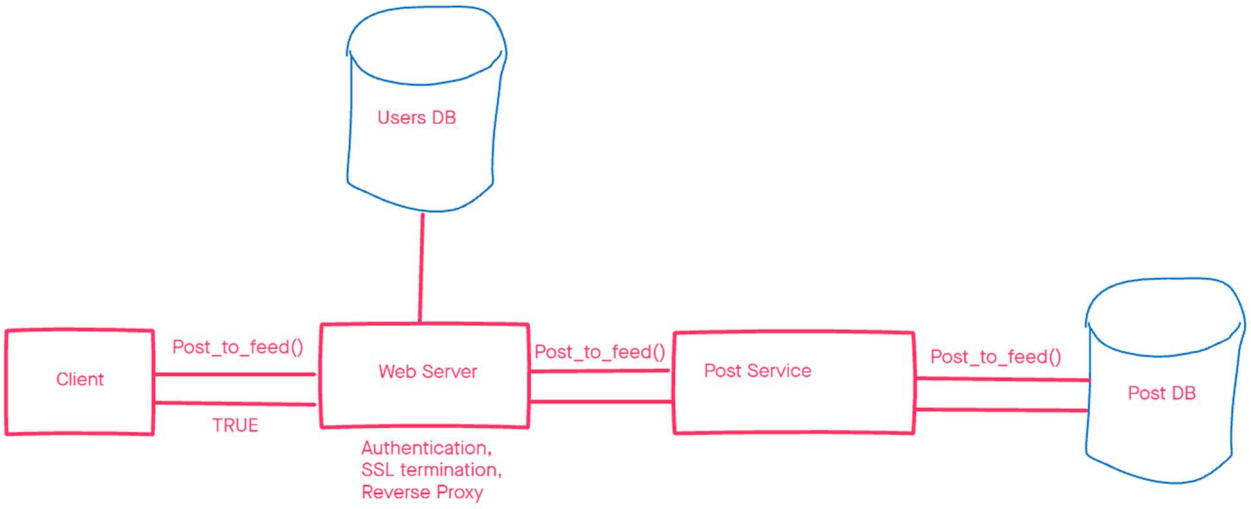
Most of the systems we are designing in these interviews deal with persisting a certain amount of information from the user into the backend and then serving it back to them after adding some business logic to it. The purpose of the HLD section is to show the interviewer that we understand the flow of data from the user into our backend and back.

We are going to start with a very simple, almost monolithic design and expand on it as we get into the LLD section. Each FR gets a HLD diagram with a web server, a service and the data store persisting or retrieving the data. You can reuse parts from a previous diagram (e.g. Post DB below) but you should try and approach each FR separately.

Remember, there are no Caches, Load Balancers, DB selections or any detailed design at this stage!

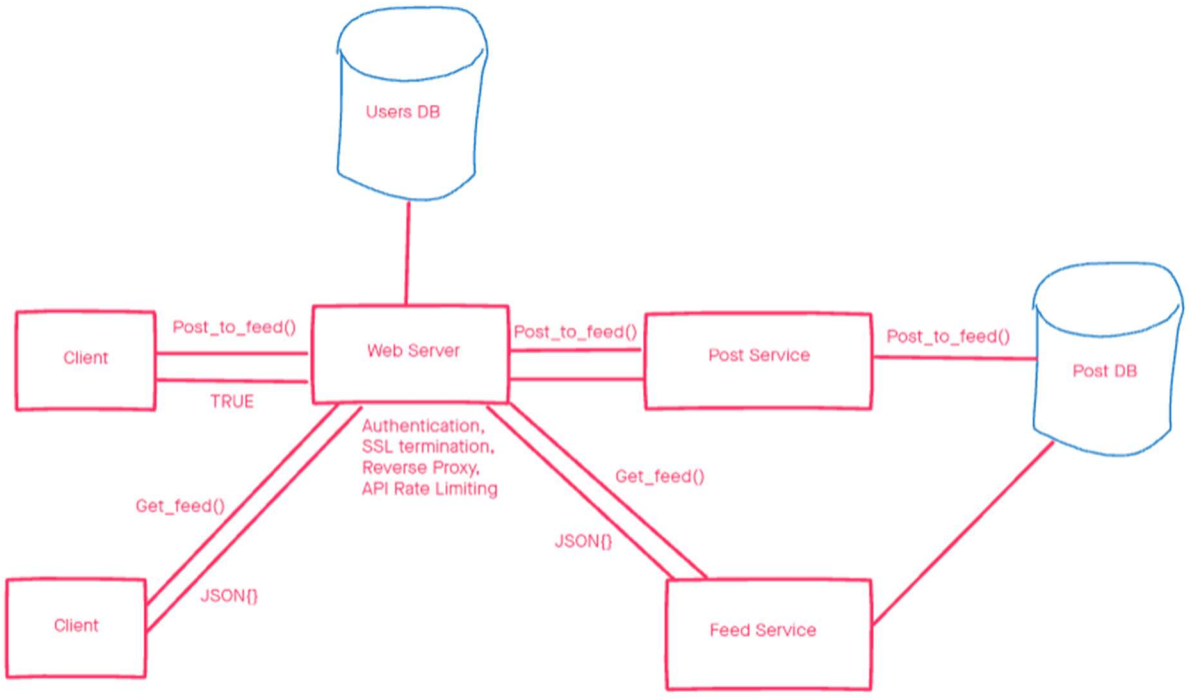
Example:

- FR: User should be able to post to their feed (text only):

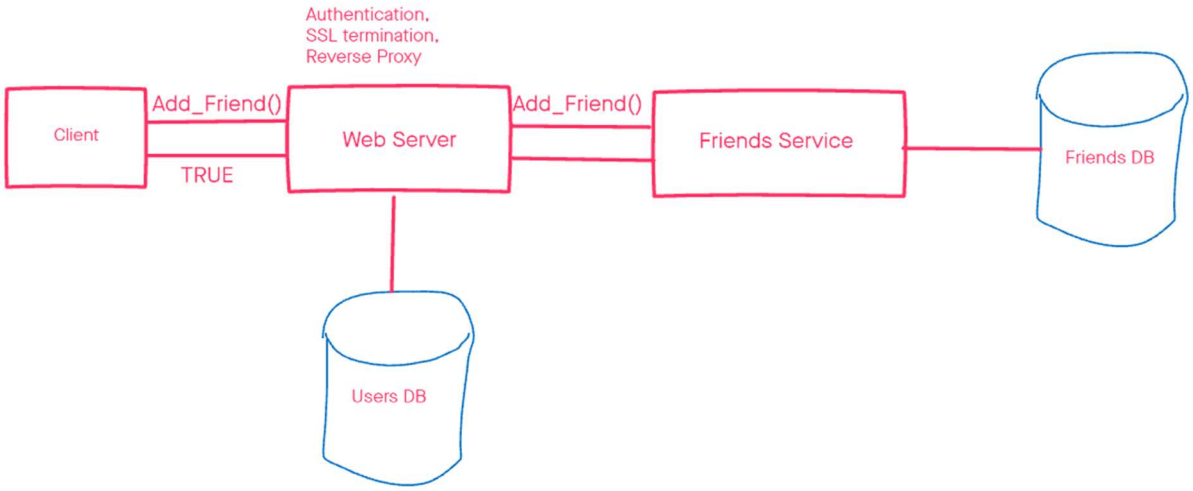


As you can see above, we are keeping things very simple and straightforward at this stage. This shows the interviewer that we think in a structured way and know how to take the FRs into account when building a solution.

- FR: User logs in and views their feed, posts shown in reverse chronological order; here we only add the feed service.



- FR: User should be able to follow other users/friends



Pointers:

- You should call out that your web servers handle authentication, SSL termination and Reverse Proxy as well as Rate Limiting. Suggest not to dive deep into these topics unless they are parts of the FRs of your design.
- At this stage, I advise not to optimize data stores or trying to decide which tables can share database servers. Each logical piece of data gets persisted in a DB/data store of its own for now. For example: user information should not be stored in the same DB as friends information even if you believe they may end up being stored on a single DB in the end.
- Add API calls on your HLD design to show how the data flows.

- Once you finish the HLD, you say: "now that we have completed the HLD, let's check that we have met all of our FRs" and go back up to the FRs. You'd be surprised how often people miss FRs in interviews...

Low Level Design – 10-15 minutes

All of the work we have done so far was done in order to get to the LLD and show our interviewers how great our design really is!

Most system designs I have seen being asked in interviews to date focus on Service Oriented Architecture, persisting, processing and serving data to the user in an efficient manner. As such, I recommend to start the design work with how to store and handle data in the backend and work our way towards the user: data storage --> services --> web servers --> user

My recommendation here is to start by writing a to-do list on the screen, this serves several purposes:

1. Ensures you don't miss anything in your LLD – candidates often do...
2. Show the interviewer we know how to do a LLD and that we've thought about everything, even if the time doesn't permit us to do it all.
3. Allow (or even directly ask) the interviewer to pick where they would like us to focus on.

Our to-do list is going to be comprised of two main sections:

1) A standard list of items most system design required. As a guiding principle, we start from the backend and move towards the user:

- **Databases: SQL/NoSQL, Choose DB technology, Database/table schema, Partitioning/sharding, Indexing, Replication, Database Caching**
- **Queuing**
- **Service resiliency**
- **LB for services**
- **LB for web servers**
- **Monitoring and analytics**

2) Specific considerations for the system we are designing. Since many of these considerations are reused across multiple design problems, I view them as **design patterns:**

Design patterns are a set of premade tools or solutions that we can selectively use according to the system we are being asked to design. Here are some of the design patterns I extracted from *Grokking the System Design* and some other online courses, and the systems using them:

- **CDN** (YouTube/Netflix/anything with media)
- **Fanout service** (Facebook posts /Twitter tweet)
- **Notification Service** (Facebook Newsfeed,/Twitter)
- **Sharding by object ID** (Twitter/Facebook Newsfeed/Yelp/Instagram)
- **Key Generation Server** (TinyURL/Pastbin/Instagram)
- **Transcoding/Encoding videos** (YouTube/Netflix)
- **Grids for location (**Uber/Yelp)

Once you learn how to use these patterns efficiently, you will be able to easily apply them in any system design, even on systems you've never practiced on!

Even cooler is that now that you know these patterns, you can use them as quick reminders for what needs to be done for each system design during the interview!

At the end of this document, I added a table with systems designs I studied, their HLD and design patterns. **This is a simple, yet super-efficient cheat sheet for interviews!** 😊

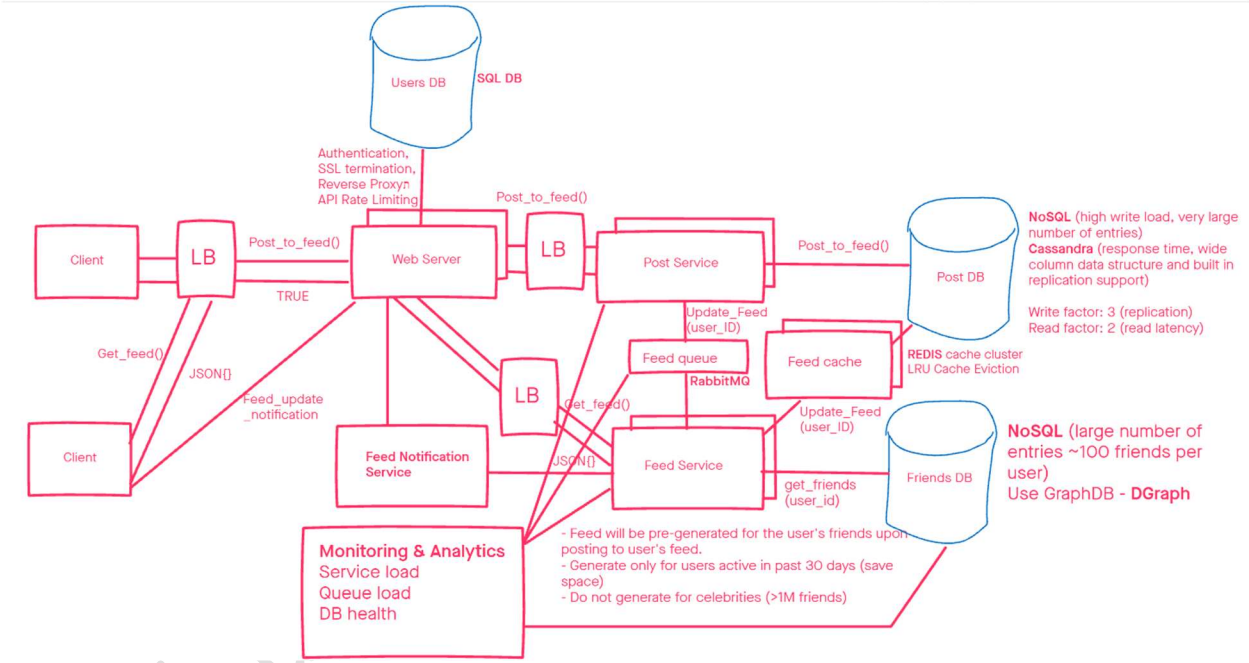
Now that we have a complete to-do list, we can start going over each item in a methodical way and add it to our design. For each item you'll want to call out a couple of options, write down the one you choose for you design and explain why you chose it. Be sure to walk the interviewer through your thought process! This is how they learn most about how smart you are and how deep your understanding of system design is!

Here is some high-level guidance and an example for the items discussed above, refer to technical literature for in depth information on each of these topics.

- **Databases:**
    - **SQL/NoSQL** - talk about both options and show you understand the tradeoffs.
        - *Example:* for Posts DB, we could choose SQL which will be simpler to implement and can service high read load through Master-Server replication. However, a SQL DB may struggle storing the amount of records we plan to store and would require sharding to scale. Sharding logic would have to be implemented in the client – complicated.
          My recommendation in this case is then to use a NoSQL DB which scales well with both reads and writes.
    - **Choose DB technology** - characterize the structure of data being stored, read/write load, consistency requirements, replication needs and choose the right data store for it.
        - *Example*: Cassandra for Post storage will persist many millions of records per day. It fits in a wide column data structure, scales well with both read and write capacity and offers great support for data partitioning.
    - **Partitioning/sharding** - in many cases we will need to partition our data in order to scale read and write. Consider using consistent hashing as it is an excellent solution for dividing the load uniformly and scaling, but be ready to explain how it works. Read Dynamo DB whitepaper for this. Shard by a field which doesn't get partitions hot.
        - *Example*: shard Posts DB by post ID as opposed to User ID which could lead to hot partitions. Add an aggregation layer to collect results from multiple DB servers.
    - **Replication** - consider 3x replication to ensure durability of data. Leverage built in capability when using Cassandra/DynamoDB. Propose a cross geo replication strategy.
        - *Example*: we will set a write replication factor of 3 for the Posts Cassandra DB. If consistency can be traded off, set read factor to 2 or even 1 to facilitate fast reads.
    - **Database Caching -** I really can't think of a case where you don't want to have a cache layer in front of your DB. Think of your DB as the queen and cache as your rook defending it! Add a cache box between the service and the database in your design drawing.
        - *Example*: add a cache layer in front of the Posts DB to improve read latency and reduce the DB load. Use LRU eviction policy to ensure latest data is kept in cache. Consider using REDIS as a scalable, semi-persistent cache solution.
- **Queuing** – great design pattern for decoupling long processing operations. In other words – where we add business logic or backend processing like image processing, feed fanout, etc.

    - *Example*: add a queue for processing post fanout into the Feed service in order to ensure the request to fan out is persisted while the response to the user posting to their feed is instantaneous.
- **Service resiliency** – call out that we will have multiple service instances, implement rate limiting in web servers
    - *Example*: we will run our service in docker containers on EC2 instances and use AWS built in Auto Scale functionality. Another option is to use Function As a Service and run the service on AWS Lambda - as long as we can ensure our service is stateless.
- **Monitoring and analytics** - add analytics on services and queues to ensure the system is servicing the users well and that the backend is performing well.

Here is a drawing of a detailed LLD for our system using some of the design patterns discussed previously:



Pointers:

- In most cases you'll want to add a cache layer in front of each of your data stores. Remember, defend your queen!

Additional tips:

- Pick a software tool you like and do all of your prep work and training with it. Don't be that person coming struggling to convey their thoughts over the share during the interview.
- Ask your recruiter which software their company uses for system design interviews (Google uses Google Draw; Amazon uses InVision) and practice with it.
- Create some cheat sheets to help you focus on what really matters during the interview. If you are like me, the act of compiling this information will deepen your understanding of the material and you probably won't need it in the end...
- Have multiple mock interviews with other candidates or experts! They are very important to building confidence and getting constructive feedback on how you are performing.

System Design – HLD and Design Patterns [use at your own risk]

Site	HLD	Design Patterns
Twitter /Facebook Newsfeed	Post: Client --> web server -> Post service --> Post DB Feed: Client --> Web Server --> Newsfeed service Friend Post Notification: Notification service --> Client2	**Fanout service**: - Update on write - pre generate friends feed when new posts are written [fast load time for friends] - Update on read - generate feed on demand [works well for inactive users - saves resources; slower for load] - Decision: hybrid! pre-generate for active friends; generate on read for inactive users or celebs with millions of friends **Queue:** Feed generation done async to maintain write perf **Notification Service** - Push notification to friends - Maintain connections using Long Poll/Web Sockets **Sharding by object ID:** - user_id - easiest but then hot users will load servers - tweet_id - better idea. Tweets distributed among DBs. But when we're looking for one user's tweets, we need to aggregate from all servers. - If need to sort by time, share by tweetID=epoch time+serial number (to avoid dups). This allows easily sorting top tweets since time X. Still need to aggregate results from all servers though.
Facebook messanger	Send message: Clien1-->Web Server -->Message Service --> Message DB -->Message Service --> Notification Service --> Client2 Update user status: Client1-->Web Server --> Status Service --> Status DB	**Notification Service:** - Long polling or Web Sockets **Session storage:** - Persist user session data in cache **Message sequencing:** - Keep a sequence number of every message for each client to determine the exact ordering of messages for EACH user. - Clients will see a different view of the message sequence, but this view will be consistent for them on all devices. **Sharding:** - Message DB: shard by user_id

YouTube/Netflix	Upload video: Client --> Web Server --> Ingest Service --> Metadata DB --> Ingest Service --> S3 (signed URI) --> Web Server --> Client --> S3 --> Ingest Service --> Transcoding/encoding service --> S3 --> CDN	**Transcoding/Encoding:** - Convert into multiple formats and resolutions - Break video into chucks, store in temp storage --> queue --> metadata update, audio and video processing --> task worker threads collect from the queue and deliver: video encoding, audio encoding and a thumbnail **CDN:** - Get the media closer to the client. User LRU eviction policy
Yelp		**Grids for location:** - Static grid size - easier solution. Could have many locations in some grids while others are empty (e.g., over sea) - Dynamic grid of 500 locations - much more efficient for storage and searching. **Sharding by object ID:** - Regions - easiest. Could lead to certain regions growing hot. In this case we'd need to repartition or use consistent hashing - LocationID (GridID) - we'll have to aggregate the data where we query (use aggregation servers) but ensures even distribution.
API rate limiter	Client -->\ --> Web Server --> Rate limiter --> Web Server --> API server (if call allowed)	**Algo for rate limiting:** - Bucket of Tokens - bucket gets filled at a set rate (# calls per min / 60, per sec). Every API call takes a token out of the bucket. If the bucket is empty, reject. **Data store selection**: REDIS - Very fast and has built in increment/decrement functionality. **API call flow:** 1) Add tokens to bucket: (time now - last update time) * number of calls allowed per sec 2) If bucket = 0, reject; else remove one token from bucket (tokens --)

Instagram	Client --> Web Server --> Ingest Service --> Metadata store --> Ingest Service --> Object store	**Availability**: - Split image read and write servers. Write is slow and will block fast reads from cache **Queue**: - Introduce message queue for writing/ingesting posts - ensures a quick response to the user and backend follow through **Sharding by object ID:** -user_id - easiest but then hot users will load servers - photo_id- better idea. Image metadata distributed among DBs. But when we're looking for one user's tweets, we need to aggregate from all servers. **Key Generation Service:** - photo_id offline key generation - photo_id = epoch time + unique ID for easy sorting for feed retrieval (31 bits: 86400 sec/day * 365 * 50 years = 1.6B sec) **CDN:** - Get the media closer to the client. User LRU eviction policy
TinyURL	Client --> Web Server --> URL Service --> DB	Two options: 1) Hash URL--> MD5 --> Base64 and cut first 6 letters (64^6 = ~68.7 billion possible strings) - ok but can have duplicates as only first 6 letters of MD5 hash used. 2) **Key Generation Service:** offline key generations + sync to ensure no dups. - Can also have several services distribute keys and get key ranges from a Zookeeper. - Faster, offline key generation
Pastebin	Client --> Web Server -> paste_service -> metadata store paste_service -> object store	Paste service - generates a short URL for each request. Better yet, move to a KGS service (cache) + key DB - Metadata store: can be a relational database (MySQL) or Cassandra - Object store: S3 - Add cleanup service (expires pastes)

Notification Service	User registers for push notificaition: Client --> Web Server -> Registration Service --> Contact DB Send notification: Service1--> Notification System --> APN --> iOS --> FCM --> Android --> SMS Service --> SMS --> Email Server --> Email	iOS push: provider -> APN (Apple Push Notification Service); needs a unique device token; syntax: JSON Android push: provider -> FCM (Firebase Cloud Messaging) SMS: provider -> SMS service Email: provider -> email service. Can use your own or off the shelf (mailchimp, SendGrid) Queue: - Producers (Services/auto schedulers/Apps) log messages in dedicated queue per message type (e.g. iOS) - Worker threads/services consumer the message and attempt to send. If fail, retry following jitter methodology. Reliability: - Call out issues: need to easily replace TPV, single server == SPOF - Workers log tasks to ensure data is not lost Cache: User info, device info, notification templates Rate limiting - Implement rate limiting to avoid spamming users Analytics: - Monitor and analyze queue length