Let us go over the 6-step framework that we have learned in the live classes.

Step 1: Requirements Gathering

Functional Requirements

- A user is able to
    - upload photos
    - download photos
    - view photos
    - search photos
- Generate news feed from all the people that user follows --> Discovery
- Users can follow each other

Non-functional Requirements

- System is reliable so that uploaded photos and videos are not lost
- Service should be highly available
- Consistency is not Paramount. Our system could be eventually consistent, so that uploaded photos or videos are discoverable after an acceptable lag(~few seconds)
- Basically our system is AP(but not C)
- Let's say latency should be between 100 to 300 ms

Step 2: Find the major components(Micro Services)

At the first look, it seems we have 2 major services:
1. Media Service - Taking care of upload, download and searching of photos
2. Feed Service - For generating news feeds from other users and serving it on the homepage or in the feed widget

But let's have a closer look at the media service. For social networking websites – usually read load is 1000x of write load. Also the read operations are expected to have lower latency for better user engagement and retention. For obvious reasons, uploads are slow as a heavy image has to be uploaded on storage servers.

Why does it matter if uploads slow? Please explain…
1. It means that the Media Service connections could be hogged by write requests. And that means read requests could be delayed(high latency) or rejected(503 errors) in worst case.
2. Also any bug on write flow could create an outage for read flow.

3. Scaling requirements for writes would be much lesser than a read. Imagine celebs posting pics after Oscar awards. Reads would be astronomically higher compared to writes.

Understood, so we should have separate services for read and write?
Absolutely! There is an added amazing benefit. We agreed that our system could be eventually consistent. So the upload service can write to the primary servers. While the download service could read from the replicas. This makes our reads horizontally scalable.

So we will have three micro services:
1. Upload service
2. Download service
3. Feed Service

Upload Service

What does an "Upload Flow" look like?

Let's explore a Naïve Approach:

1. User uploads photo from the device. Photo should be reliably saved in a storage.
2. Photo is transferred in form of bytes from client to backend web server on a persistent HTTP/S connection.
3. Backend server transfers the received bytes to a storage engine.

What's naive about this approach?
1. This process has a lot of network calls involved – higher latency
2. User keeps on waiting till the upload is completed.

Ok, so how do we transform it into a Better Approach:
Think, how we can eliminate the extra network hops. Can we directly upload from client to cold storage?
1. When user hits upload button on client, a temporary and secure connection is created with storage engine directly.
2. When upload is complete, secure connection is terminated.
3. Client sends the Storage location to backend web server. Backend server saves this information.
* Secure connections have a lease for a specific time period and can access only a particular storage bucket. See Using signed URLs for more details.

Metadata Storage

In point 3 above, you mentioned that the backend server saves upload information. Is it some kind of metadata?

Bull's eye!! It is a system design best practice to separate the object storage and metadata storage. Think of metadata storage in this case as a key-value hashmap. If we can make our metadata small, then we can store it in-memory for faster response.

Download Service

Okay, so what does the download flow look like? with the separate object storage and metadata?

It is a synchronous flow.
1. User requests a photo by sending photo_id in the request. Of course, the request has other fields as well like user_id, auth_token, etc.
2. Download Service accepts the request.
3. Makes a call to metadata storage to get the photo object's actual location/URL on storage engine.
4. Returns the URL to client along with some auth information so that client can make direct connection with storage engine
5. Client creates a temporary and secure connection and downloads the image directly from storage engine.
6. Secure connection is terminated by client. (If not, then it will automatically expire after a threshold)

What is photo_id?
In most simplistic way, it could be a simple auto increment id generated by a database. This id genrataion could be done by a "ID Generation Service".

ID Generation Service (IGS)

It would have a dedicated separate database instance to generate auto-incrementing 64 bit IDs.

To make this Database fault tolerant, we could have hot-standby servers. If and when this primary goes down, the standby could start acting on its behalf.
DB will generate a 64 bit ID. That means 2^64 ID. That's like (2^32)^2 - Square of 4 billion. That would be a gigantic number.

- Upload service makes a request to IGS - Give me 20000 new and unused ID
- IGS makes a call to DB -- Give me 20,000 ID for client "Upload Service"

1. DB finds the max ID that it has already generated. Let's say 100,000
2. DB updates this max to 120,000
3. DB inserts a new row in "id_audit_log" table. Columns could be:

   start_id, last_id, client, created_date
   1000001, 120000, "Upload Service", <some_datetime>
4. DB returns <100001,120000> tuple to IGS
5. Note that DB does all these operations with a lock. So it is thread-safe but with high latency.

- IGS returns this tuple to "Upload Service"
- "Upload Service" caches it in its memory for faster access
- Even if one of the "Upload Service" servers goes down, we are just losing 20k keys at a time. So it is fine.
- Whenever "Upload Service" server is running low on ID supply, it makes a proactive call to DB to fetch next 20k ID

Feed Service

To create the News Feed for a user, we will fetch the latest/popular top N photos from each person that user follows.

Approach 1:
1. When a user opens her feeds widget or refreshes it, frontend client makes a call to backend feed service to get feeds.
2. Feed service queries "UserFollows" table to get the list of people that user follows. (In actual implementations, this list of people could be sorted by an "Affinity Score", so that the persons frequently communicated are on top of list)
3. Feed Service will fetch metadata of each followed user's latest 100 photos.
4. Feed Svc will use a ranking algorithm to determine the top 100 photos (based on recency, likeness, etc.) and return them to the user.

Sounds good but we are querying a DB table on each call?

Exactly!! That's the issue with this approach. It will have high latency. Any guesses, how we can optimize this approach?

Answer lies in our requirement itself. We are OKAY with eventual consistency. We are OKAY if some of the latest published photos reach the audience with a lag. That means we can pre-generate the News Feed!!
We can have dedicated servers that pre-generate News Feed for the requested user and store feeds in a ``UserNewsFeed`` table. Steps are:

- Trigger received by Feed Service to generate news feed for user `X`

- Feed Service finds out the last run time of the feed generation for user `X` -- Let's say it is `T1`
- Feed service includes only those photos whose creation_time > T1
- Store feeds in a 'UserNewsFeed' table. Update last run time to T2

So our new approaches are as follows:

Approach 2:

- Request received by Feed Service to return feeds for user X
- Feed service queries 'UserNewsFeed' and gets fetches latest feed
- Send it to client

By 80-20 rule, we can pre-cache the feeds for top 20% users. So we would serve 80% requests from the cache itself.

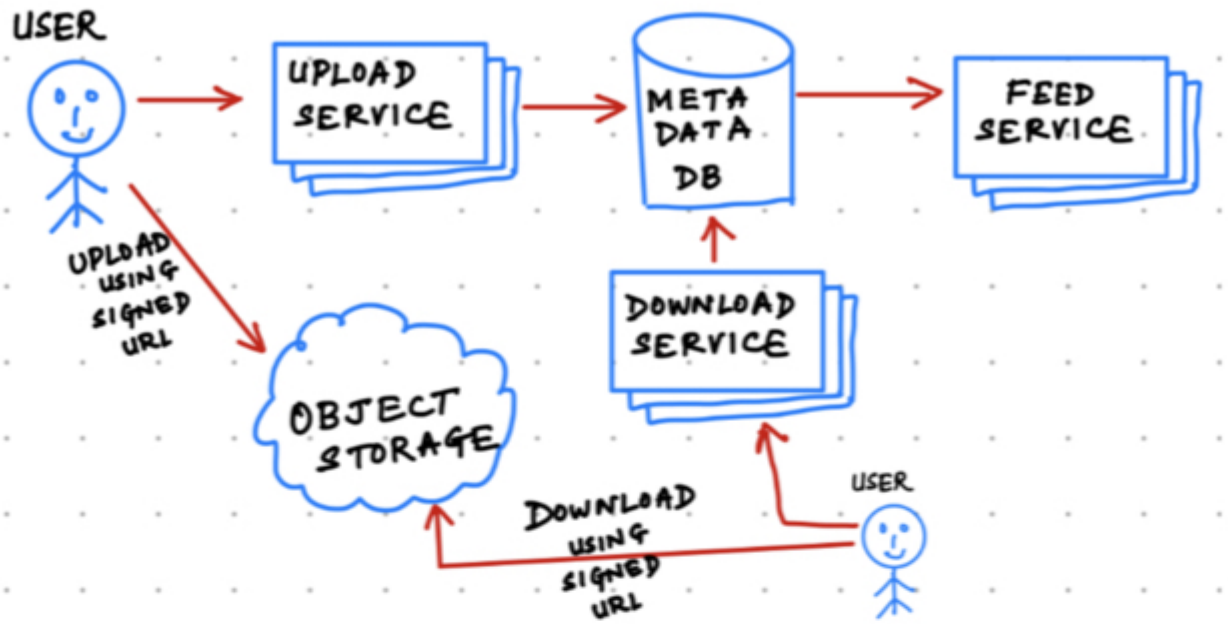
But this is not a perfect solution. Because the user needs to fetch the feeds every time. Also, many times there won't be new feeds if the user fetches new feeds continuously.
So we can go for a slightly modified approach. The servers can keep on generating the news feed at each time interval. For example, our servers can generate feeds at each 60 seconds. In this case, User can maintain a Long Poll request with the server for receiving the updates.

Approach 3:
Problem with the above approach is that there is a delay of 60 seconds. Hyper-active users would be quite frustrated with this delay.

So we could have a hybrid approach, where the hyper active user's clients can fetch the feeds at a shorter duration like 15 seconds.

Step 3: High Level Design

All right, with all the above information – we are in a good place to create a high level architecture diagram



Step 4: Deep Dive into each service

APIs

- **upload_image**(user_id, image_url, timestamp) -> success/failure
- **download_image**(user_id, image_id, requested_resolution) -> <image_url>
- **download_image_by_userid**(user_id, requested_resolution, page_size) -> Array of image_url ordered by time, limit by page_size
- **get_follow_feed**(user_id, timestamp, page_size) -> list of newest posts from user follow list, ordered by time, limit by page_size

Schema

We can have 4 tables:

User

id, name, email, creation_date

Photo

id, user_id, location_url, creation_date, title

UserPhoto

--> store the relationships between users and photos to know who owns which photo
id, user_id, photo_id

UserFollows

user1, user2

Choice of Database

1. For Metadata: We do not need any kind of transactions or relations(joins) here. Therefore, we can ignore RDBMS. We can go for a distributed key-value datastore here. For example, in case of UserPhoto table, key would be user_id and value would be a list of photos that user has uploaded.
Wide-column databases like Cassandra/BigTable would serve the purpose well. Since they have sharding and replications built-in, our system would be highly available and reliable. In requirements, we agreed that consistency is not a key requirement. Therefore, we won't have any issue with the eventual consistency of wide-column databases.

Cassandra or key-value stores, in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don't get applied instantly; data is retained for certain days (to support undeleting) before getting removed from the system permanently.

2. For Photos: We can store them in a distributed file storage like HDFS, GCS, or S3. Our database would have replication so that data is stored reliably. Having multiple replicas will make the data fetching faster -- as more and more servers would be able to read concurrently. These servers are basically a Linux Server. So these servers need not cache file data because objects are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory. Free cache without any extra effort!!

Step 5: Identify the need for scale

Capacity Estimation

- 500 million daily active users

- 100 million photos uploaded per day
- Average photo size = 200KB

- Space required for photo storage:

    - 1 day = 200KB * 100million = 200*10^3*100*10^6 = 20*10^12 = 20TB
    - 6 years = ~1000 days in 3 years = 2000*20TB = 40 PB

- Space required for Metadata:

    1-- For User Table
    id(4 bytes) + name (50 bytes) + email(50 bytes) + creation_date(4 bytes) ~= 100 bytes
    500 Million Users * 100 Bytes = 5*10^2*10^6*10^2 = 50*10^9 = **50 GB**

    2-- For Photo Table (Space required for 6 years)
    id(4 bytes) + user_id(4 bytes) + location_url(256 bytes) + creation_date(4 bytes) + title(100 bytes) = 368 bytes ~= 350 bytes
    100 million photos uploaded per day * 350 bytes * 2000 days =
    10^2*10^6*35*10^1*2*10^3 = 70*10^12 = **70 TB**

    3-- For UserFollows Table
    500 million users * 500 followers * 8 bytes =
    5*10^2*10^6*5*10^2*8 = 4*10^12 = **4 TB**

    4-- For UserPhoto Table (Space required for 6 years)
    id(4 bytes) + user_id(4 bytes) + photo_id(4 bytes) ~= 10 bytes
    Let's say each user has 10 photos per day
    500 million users * 10 photos * 10 bytes * 2000 days =
    5*10^2*10^6*10^2*2*10^3 = 10*10*13 = 100*10^12 = **100 TB**

    Total = 50GB + 70 TB + 4 TB + 100 TB ~= **175 TB**
    If one DB shard is 4TB, we will need 44 shards. We would fill our shards to 70% capacity only to take care of bursts in traffic. So we would need 63 shards. For high availability, we can have 2 replicas for each shard(that means a replication factor of 3). So we need to have 189 servers. Let's call 189 as N to be generic.

    2^10 = 10^3 = 1 Thousand ~= 1KB
    2^20 = 10^6 = 1 Million ~= 1MB
    2^30 = 10^9 = 1 Billion ~= 1GB
    2^40 = 10^12 = 1 Trillion ~= 1TB
    2^50 = 10^15 = 1000 Trillion ~= 1PB

~1000 days in 3 years

In power of 2	In power of 10	In Words	Memory
2^10	10^3	1 Thousand	1 KB
2^20	10^6	1 Million	1 MB
2^30	10^9	1 Billion	1 GB
2^40	10^12	1 Trillion	1 TB
2^50	10^15	1000 Trillion	1 PB

Step 6: Propose the distributed architecture

Identify Bottlenecks:

We check bottlenecks for following criteria
STPHAG --- Storekeeper Threw Pizza Hamburger And Gravy
Storage, Throughput, Parallelism, Hotspot, Availability, Geo

Storage

For Photos:

We need to store 40PB for 6 years. Of course, this data requirement would only grow as our service grows in popularity. This much data can not exist on one server. That's why we need to have a distributed file storage. Let's say we have N servers for this storage. Then a photo will be stored on "photo_id%N" server location. We are sharding on photo_id. With consistent hashing, our server location will become agnostic to addition or removal of a server.

For Metadata:

As per our calculations, 175 TB data would be there for Metadata in 6 years. Definitely, this much data can not be stored in 1 server. So we need Data Sharding.

Let’s discuss different approaches:

1. Sharding on User ID

This helps us to keep all photos of a user on the same shard.
We can find shard by doing user_id%N. With consistent hashing, our server location will become agnostic to addition or removal of a shard.

Pro: Simple to implement as based on user_id
Cons: For hot users(who post lot of photos) and for celebrities(who are followed by millions of users), their particular shard would be accessed manifold times. Therefore, this approach is not scalable.

2. Sharding on Photo ID

When the request comes for storing a photo, "upload service" would contact a "ID Generation Service" to request a photo_id.
Then it would save the photo on cold storage and save the information in Metadata shard using photo_id%N strategy.

Throughput

- For "Upload Service" and "Download Service" - we would have multiple servers running behind the load balancers.
- Also, we are running our servers at 40% capacity. So even sudden 2x traffic could be handled.
- If requests increase more than 2x then we can have an autoscaling setup for the system. Autoscaling could be triggered when:
    1. CPU crosses a threshold like 70% for more than 5 minutes OR
    2. Requests per second increase by more than 2x for 5 minutes
- Since we have a Load Balancer, new servers will be added automatically. We will also have service discovery in place, so that new servers start receiving requests as soon as they are registered and pass health checks(simple TCP pings and more complex Application level health checks).

For storage servers(index metadata DB and cold storage), our servers are running at 40% capacity and we have sharding enabled. With consistent hashing strategy, shard servers in the storage layer could be scaled up and down automatically.

Caching for Metadata DB

Since our system is read heavy, we need to have a solid caching mechanism. Going by the 80/20 principle, we can have a distributed caching solution which caches 20% data in memory -- This should suffice for serving our 80% requests ultra-fast from memory.

Also, reducing the calls to metadata DB by 80%. This would help in handling the throughput and helping our system to be available.

Parallelism

This comes into picture if there are some bulky requests being made to the system. Since the client uploads/downloads to the cold storage directly, we don't have this bottleneck in system.

Hotspots

With our sharding strategy on photo_id, we have removed the danger of hotspots.

Availability

We have multiple servers running for our services with auto-scaling baked in. So our service would be available.
Similarly, sharding and replications would help make our storage layer available.

Geo

If our servers and storage are located only in one region - let's say North America. Then the requests coming from Asia would have very high latency - as the inter-continental latency can be as high as 500ms.
So we have 2 ideas here:

1. We will have our servers and DB instances in multiple regions spread across the world. Then using DNS policy, we make sure that a request is directed to the nearest server.
2. We will have CDN servers for each region which will cache the frequently requested photos. This will help us to tackle celebrity situations as well - for example if Justine Beiber posts a photo of Selena Gomez. In that case, a sudden burst of millions of requests would be efficiently served by region specific CDN servers. It will also safeguard our system from burst.