# Design Photo/Video Sharing Social N/W Platform

Design social networking service that enables its users to upload and share their photos and videos with other users or followers. Users can choose to share information either publicly or privately.

The system would be read-heavy, so we will focus on building a system that can retrieve photos quickly.

1. Practically, users can upload as many photos as they like; therefore, efficient management of storage should be a crucial factor in designing this system.
2. Low latency is expected while viewing photos.
3. Data should be 100% reliable. If a user uploads a photo, the system will guarantee that it will never be lost.

Functional Requirements:

1. Users should be able to upload and view the photos.
2. Users can follow any numbers of users.
3. User can view news feed ( photos/videos) from the other user.
4. User can search any other based on username or hashtag.

Non-Functional requirements:

1. Latency should be low.
2. Availability should be high
3. Performance should be good.
4. Fairly consistent
5. Read-heavy

Capacity calculations

Average media file size = 1MB

Active users = 1B

Avg upload of photos in a day per user= 3

Total uploaded photos per day = 3\*1 = 3B

Each media take 1MB, we will need about

3\*1000000\*1 = 3000GB(approx) of storage for media each day = 3TB per day

Database required : Relational Database(MySQL)

Tables: User, Photo

User Table

UserId(PK)

Name

Email

Mobile number

Photo table

Photo Id(pk)

UserId(fk)

Caption

Location

Followers table

FolloweeId(pk)

FollowerId(FK)(UserId)

API design

login(username, password)

searchUser(username)

followUser(userId, targetUserId)

addPost(userId, file, caption)

getFeed(userId, timestamp)

deletePost(userId,postId)

viewPhoto(postId)

Draw.io for diagram

**NOW THE REAL ONE**

FUNCTIONAL REQUIREMENTS

1. User should be able to upload an image/video on his profile.
2. User should be able to see uploads of other users followed the user.
3. User should be able to follow other users.
4. User can perform search for an image/video based on title.

## **Non Functional**

1. The user feed latency should be low.
2. We are okay with eventual consistency as uploaded image can be shown to another user after few milliseconds.
3. Our app needs to be highly available.
4. The Data store we will be using for storing image/video should be reliable and data should not be lost.

## **Additional Requirements**

1. Users can add tags to a photo/video.
2. Users can put comments on a post.
3. Users can search photo based on tags.

# **Capacity Estimation**

Daily active users = 500 million

Avg media file size = 1MB

Avg mdia upload per user = 3

Daily media upload = 3\*500 = 1500million

Daily media upload file size = 1\*1500000 000= 1.5PB/day

**API**

1. Upload Image(userId, mediaId, title)
2. Get feed(imageId, imageUrl, title)
3. FollowUser(userId, targetUserId)
4. searchUser(username)
5. **Data store for storing uploaded image** — Our System is read heavy, so we need a data store that can quickly fetch the uploaded image and render on user application. Couple of things that needs to be kept in mind is **1)** The data store should be reliable as we do not want user’s uploaded image to get lost. **2)** User can upload as many images as they so the data store should be scalable to handle billions of images. **3)** Latency should be low when retrieving the photos. We can consider an **object storage** to store the uploaded images by user something like AWS S3. There are other types of storage as well like file storage and block storage but considering the above factors object storage will be a right fit for our design as it gives low read latency and efficient management of huge number of records.
6. **Data store for storing user data and its uploads.** — Now we have a data store to store the uploaded image by users. We need a database to store the metadata of user uploads and user data. Things to keep in mind while deciding data store — **1)** The database should be highly available. **2)** It should have low read latency as our system is ready heavy. **3)** It should be scalable enough to handle billions of record. **4)** It should be reliable and should support sharding and replication. Considering the above factors we do not see a requirement for relational database. We can go for a key value based NoSQL database. For this system design we can choose AWS DynamoDB to store user data and image uploads metadata.

**Database design**

1. **Table to store user data**

**userId: string[HashKey]**

**name: string**

**emailId: string**

**creationDateInUtc: long**

**2. Table to store follower data**

**followingUserId\_followerUserId: string [HashKey]**

**followingUserId: string [RangeKey]**

**followerUserId: string**

**creationDateInUtc: long**

**We can’t choose followingUserId as a hashKey because it can create an unbalanced partition since there can be user’s who will be having millions of follower’s. These type of user’s are known as hot users. Hence, to maintain a balanced partition we can choose a combination of followingUserId and followerUserId as a hashKey**

**3. Table to store user uploads**

**uploadId: string[Hashkey]**

**userId: string[RangeKey]**

**imageLocation: string**

**uploadDateInUtc: long**

**caption: string**

**4. Table to store the user feed data**

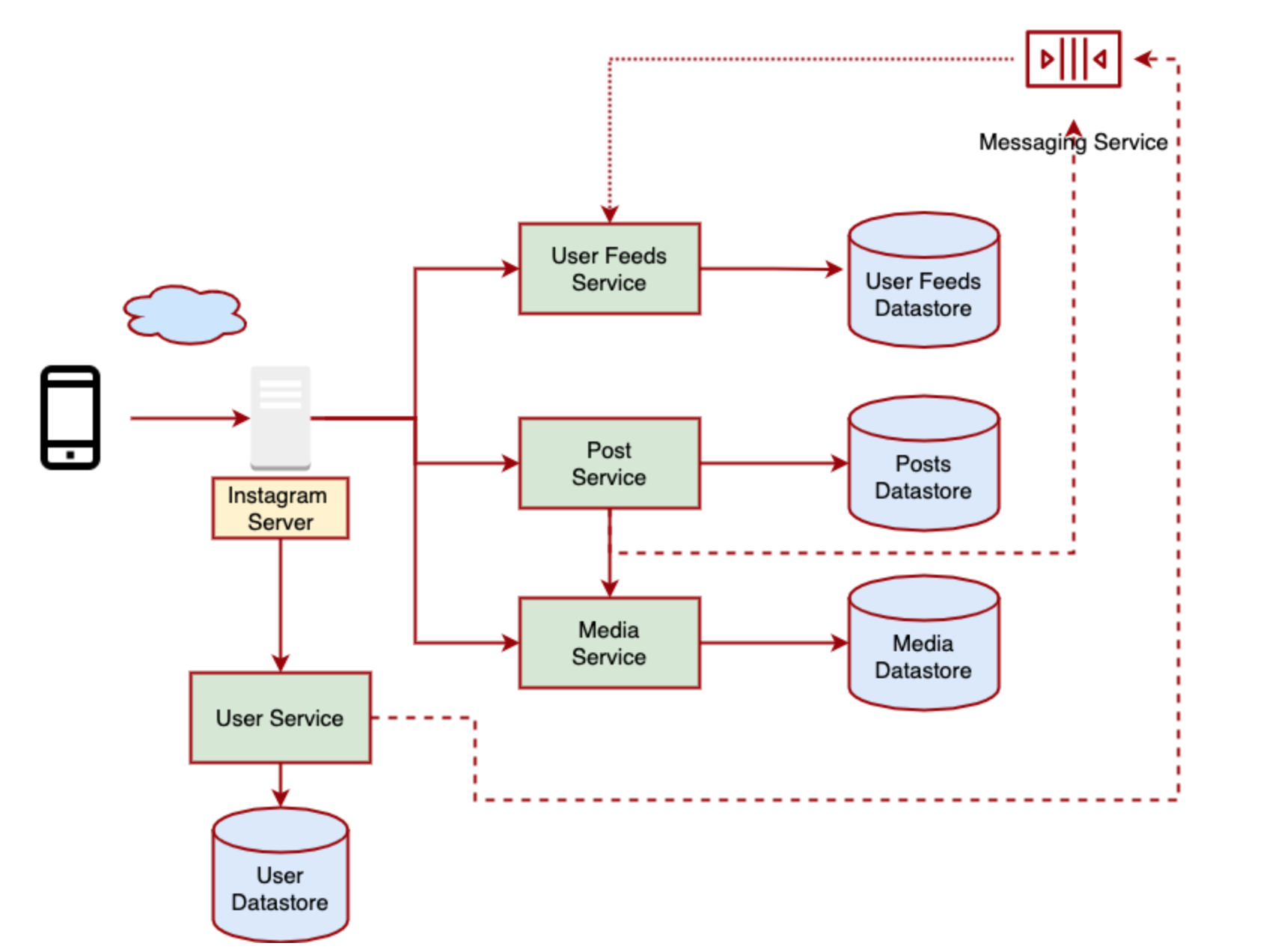
**userId: string[Hashkey]**

**uploadId: string**

**creationDateInUtc: long[RangeKey]**

# **Component Details**

1. **Client — These will be the mobile/desktop application that will connect to backend servers via REST API’s defined above.**
2. **Load balancer — We will use load balancer’s to distribute the traffic between different servers. This will make our System more available and in case a server goes down behind a load balancer, load balancer can distribute the traffic on different servers.**
3. **Image Service — Image service is responsible for providing API’s to upload image and get image meta data. The meta data API will return the image path in s3 which will be used by clients to load image on their application.**
4. **S3 — We are using object storage to store the uploaded images by users. AWS S3 is scalable and cheap object storage that we can use here. We can integrate it with AWS CloudFront so that the images can be rendered on user application much faster.**
5. **CloudFront — Amazon CloudFront is a content delivery network (CDN) service built for high performance, security, and developer convenience. With the help of CloudFront the images will be rendered faster on user application.**
6. **Image DDB — We use AWS Dynamo DB to store the user uploads image metadata. We have discussed about this in above section.**
7. **SNS — On every user upload we are publishing a notification with the help of AWS Simple Notification Service. This will be helpful in other processing like monitoring, feed generation, analytics, etc. Different SQS can subscribe to this SNS for listening the events.**
8. **SQS — We use AWS Simple Queue Service that will subscribe to upload event SNS and the feed generation service will listen to this SQS for processing the events.**
9. **Feed generation Service — This service is responsible for user feed generation. It will listen to the user upload events via SQS and start the process for user feed generation. This service will handle millions of events and there can be a separate discussion about the Low Level Design for this service. We will cover that in future article.**
10. **Feed DDB — We are using Dynamo DB to store the user feed data. The feed generation service will interact with DDB to update the user feeds.**
11. **Redis Cache — For keeping the read latency low for our users, we implement a caching layer in between our feed generation service and DDB. When a request will come to fetch a user’s feed, it will first check in the redis cache, if not available then it will fetch it from DDB and return the response.**



Below are some common database choices for each service in the Instagram system:

1. **UserService:** The UserService is responsible for managing user data, authentication, and user-related operations.
   * **Database Choice:** Relational Database (e.g., PostgreSQL or MySQL).
2. Explanation: The UserService typically deals with structured user data, user profiles, and authentication information. A relational database is a common choice to handle these requirements, ensuring data consistency and supporting complex queries for user-related operations.
3. **UserFeedService:** The UserFeedService is responsible for generating personalized user feeds that display posts from users a person follows.
   * **Database Choice:** NoSQL Database or In-Memory Database (e.g., Redis).
4. Explanation: To provide low-latency access to the user feed and efficiently handle large amounts of data, a NoSQL database or in-memory database (e.g., Redis) can be used. These databases can store and retrieve the posts and their metadata efficiently based on user preferences.
5. **Post Service:** The Post Service is responsible for managing user posts and their associated metadata.
   * **Database Choice:** NoSQL Database or a combination of Relational and NoSQL Databases.
6. Explanation: A NoSQL database like MongoDB or Cassandra can be used to store post data and its metadata. Additionally, a combination of Relational Databases (e.g., PostgreSQL or MySQL) and NoSQL Databases may be used for managing various aspects of post data, such as comments, likes, and hashtags.
7. **Media Service:** The Media Service is responsible for storing and serving media content, such as images and videos, uploaded by users.
   * **Database Choice:** Cloud-Based Object Storage (e.g., Amazon S3 or Google Cloud Storage).
8. Explanation: Cloud-based object storage solutions are preferred for efficient and scalable media storage. They can handle media files and serve them quickly to users, providing a seamless media sharing experience.

References

https://www.codercrunch.com/design/634265/designing-instagram