**Instagram**

1. Solidify the requirements – both Functional and Non-Functional

* Functional Requirements:
  + **Post photos and videos**: The users can post photos and videos on Instagram.
  + **Follow and unfollow users**: The users can follow and unfollow other users on Instagram.
  + **Like or dislike posts**: The users can like or dislike posts of the accounts they follow.
  + **Search/ View photos and videos**: The users can search photos and videos based on captions and location.
  + **Generate news feed**: The users can view the news feed consisting of the photos and videos (in chronological order) from all the users they follow. Users can also view suggested and promoted photos in their news feed.
* Non-Functional Requirements:
  + **Scalability**:
    - The system should be scalable to handle millions of users in terms of computational resources and storage.
    - By Incorporating Horizontal Scaling.
  + **Minimum Latency**:
    - The latency to generate a news feed should be low.
    - By Incorporating Redundancy of servers (application and storage)
  + **Highly Availability**:
    - By Incorporating Fault tolerance (failover mechanisms for load balancers) and Repetition of System Bottlenecks.
  + **Durability:** Any uploaded content (photos and videos) should never get lost.
  + **Consistency:**
    - We can compromise a little on consistency. It is acceptable if the content (photos or videos) takes time to show in followers’ feeds located in a distant region.
  + **Reliability**: The system must be able to tolerate hardware and software failures.

1. Scope the Problem

* What kind of clients? Mobile Apps, Web Browsers, Smart TVs?

1. Capacity/ Resource Estimation
2. Traffic Estimates
3. DAU – 500M
4. Total Users – 1B
5. Photos shared per day – 60M
6. Videos shared per day – 35M
7. Max Photo Size – 3MB
8. Max Video Size – 150MB
9. Read/ Write Ratio – 100 : 1
10. Requests by each user per day = 20
11. Daily User handling limit of a server (assume that a typical Instagram server handles 100 requests per second) – 8640000
    * PhotoUploaded/ Sec –
    * PhotoViewed/ Sec –
    * Total Servers Required – (500M x 20) / 8640000 = 1157
12. Storage Estimates
13. Time Duration for which objects are required to be stored – 1 year
14. Total Data storage per day – (60M x 3MB + 35M x 150MB) = 180. 10^12B + 5250. 10^12B = 5430TB

* Total Storage Required for 1 year = 5430 x 365 = 1981.95PB (Ignoring data for comments, likes, user’s information and post metadata)

1. Bandwidth Estimates

* Incoming Data – 502.78 Gbps
* Outgoing Data – 50.278 Tbps

1. Memory (Cache) Estimates
2. 80/20 Rule? Yes

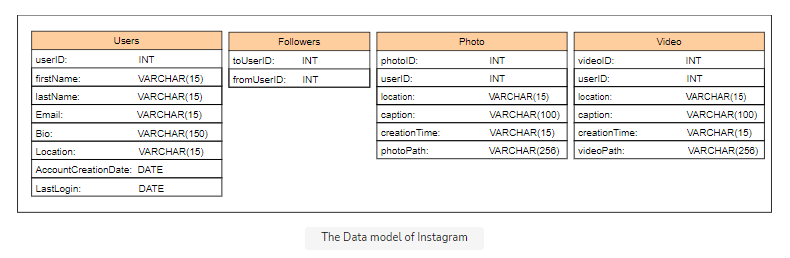
* Cache Size = (5430TB) x 0.20 = 1086TB

1. System API’s

* postMedia(userID, media\_type, list\_of\_hashtags, caption)
* followUser(userID, target\_userID)
* likePost(userID, target\_userID, post\_id)
* searchPhotos(userID, keyword)
* viewNewsfeed(userID, generate\_timeline)

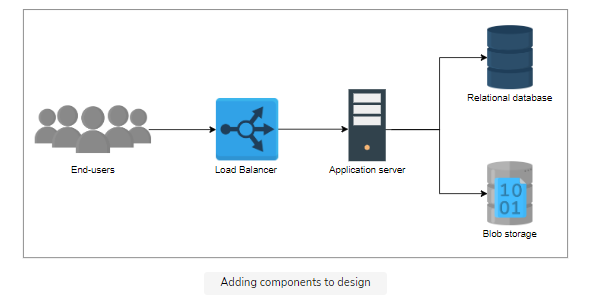
1. Database Design

* Our data is inherently relational, and we need an order for the data (posts should appear in chronological order) and no data loss even in case of failures (data durability).
* Moreover, in our case, we would benefit from relational queries like fetching the followers or images based on a user ID.
* Hence, SQL-based databases fulfill these requirements.
* On Basic level, we need the following tables:
  + Users, Followers, Photos, Videos

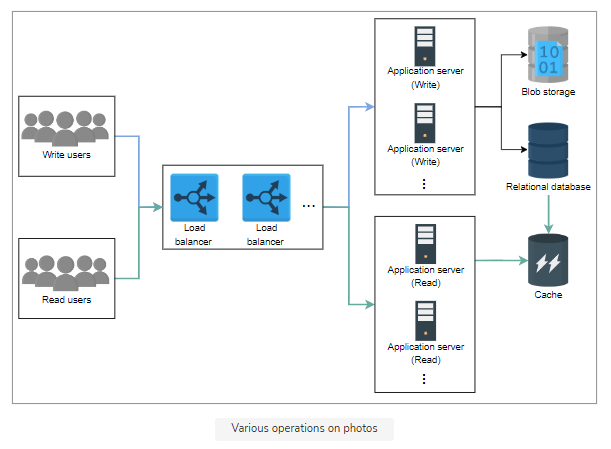


* Most modern services use both SQL and NoSQL stores. Instagram officially uses a combination of SQL (PostgreSQL) and No-SQL (Cassandra) databases. The loosely structured data like timeline generation is usually stored in No-SQL, while relational data is saved in SQL-based storage.

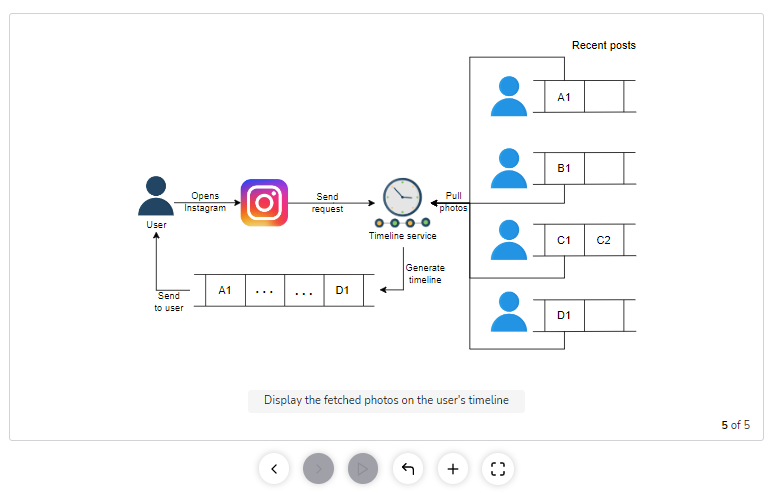
1. Present the building blocks of the Design
2. Load Balancers: To balance the load of the requests from the end-users.
3. Application Servers: To host our service to the end-users.
4. Relational Databases: To store our data.
5. Blob store: We’ll store the photos and videos in a blob store (like S3) and save the path of the photo or video in the table as it is efficient to save larger data in a distributed storage.
6. Task scheduler: Will be used for Story of our Instagram. In the story feature, the users can add a photo that stays available for others to view for 24 hours only. We can do this by maintaining an option in the table where we can store a story’s duration. We can set it to 24 hours, and the task scheduler deletes the entries whose time exceeds the 24 hours limit.
7. Cache:
8. CDN:
9. Propose a Design Diagram and get an agreement



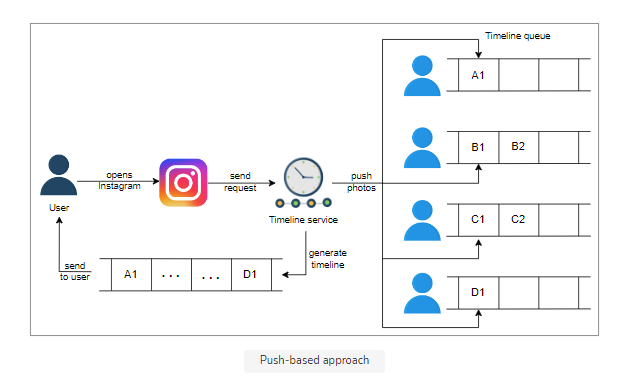
* **Separate Read/ Write Servers**:
  + The read requests are more than write requests and it takes time to upload the content in the system. It is efficient if we separate the write (uploads) and read services.
  + The multiple services operated by many servers handle the relevant requests. The read service performs the tasks of fetching the required content for the user, while the write service helps upload content to the system.
* **Cache Reads:** 
  + We also need to cache the data to handle millions of reads. It improves the user experience by making the fetching process fast.
* **Lazy Loading:**
  + We’ll also opt for lazy loading, which minimizes the client’s waiting time. It allows us to load the content when the user scrolls and therefore save the bandwidth and focus on loading the content the user is currently viewing. It improves the latency to view or search a particular photo or video on Instagram.



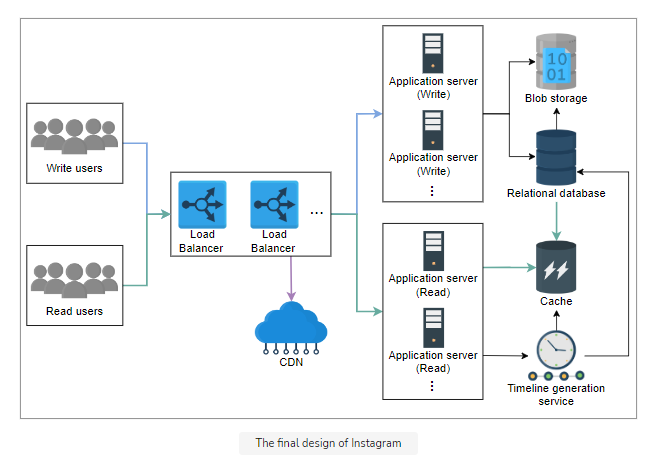
* **Generate Timeline:**
  + **Pull Approach:**
    - When a user opens their Instagram, we send a request for timeline generation. First, we fetch the list of people the user follows, get the photos they recently posted, store them in queues, and display them to the user.
    - But this approach is slow to respond as we generate a timeline every time the user opens Instagram.
    - Shortcomings of the pull Approach: Instagram is a read-heavy system. Many people don’t post any photos and instead just view others’ posts. So, the calls we make to fetch the recent posts from every follower will usually return nothing. Hence, we should keep in mind that it is not a write-heavy system and should develop a possible solution to cater to it.



* + **Push Approach:**
    - In a **push approach**, every user is responsible for pushing the content they posted to the people’s timelines who are following them. In the previous approach, we pulled the post from each follower, but in the current approach, we push the post to each follower.
    - Consider an account that belongs to a celebrity, like Cristiano Ronaldo, who has over 400 million followers. So if he posts a photo or a video, we will push the links of the photo/video to 400 million+ users, which is inefficient.



* + **Hybrid approach:**
    - Let’s split our users into two categories:
      * Push-based users: The users who have a followers count of hundreds or thousands.
      * Pull-based users: The users who are celebrities and have followers count of a hundred thousand or millions.
    - The timeline service pulls the data from pull-based followers and adds it to the user’s timeline. The push-based users push their posts to the timeline service of their followers so the timeline service can add to the user’s timeline.
    - We have used the method which generates the timeline, but where do we store the timeline? We store a user’s timeline against a userID in a key-value store. Upon request, we fetch the data from the key-value store and show it to the user. The key is userID, while the value is timeline content (links to photos and videos). Because the storage size of the value is often limited to a few MegaBytes, we can store the timeline data in a blob and put the link to the blob in the value of the key as we approach the size limit.
* **Task Scheduler:**
  + In the story feature, the users can add a photo that stays available for others to view for 24 hours only. We can do this by maintaining an option in the table where we can store a story’s duration. We can set it to 24 hours, and the [task scheduler](https://www.educative.io/collection/page/10370001/4941429335392256/6152021643624448) deletes the entries whose time exceeds the 24 hours limit.
* **CDN (Content Delivery Network):**
  + We’ll also use CDN (content delivery network) in our design. We can keep images and videos of celebrities in CDN which make it easier for the followers to fetch them. The load balancer first routes the read request to the nearest CDN, if the requested content is not available there, then it forwards the request to the particular read application server (see the “[load balancing chapter](https://www.educative.io/collection/page/10370001/4941429335392256/4521972679049216)” for the details). The CDN helps our system to be available to millions of concurrent users and minimizes latency.



8. Workflow

* Not discussed in this design

1. Specific Design Components:

* Timeline generation service can be considered as design specific component. Details of which is described above.
* How can we count millions of interactions (like or view) on a celebrity post?
  + - We can use [sharded counters](https://www.educative.io/collection/page/10370001/4941429335392256/6126387596886016) to count the number of multiple interactions for a particular user. Each counter has a number of shards distributed across various edge servers to reduce the load on the application server and latency. Users nearest to the edge server get the updated count frequently on a specific post compared to those in distant regions.

1. Design Evaluation:
2. Availability
   * **Replication of Components**: Most of our building blocks, like databases, caches, and application servers have built-in replication that ensures availability and fault tolerance
   * **~~Backups to Cloud Storage (Amazon S3)~~**~~: To handle disasters, we can perform frequent backups of the storage and application servers, preferably twice a day, as we can’t risk losing URLs data. We can use the Amazon S3 storage service for backups, as it facilitates cross-zonal replicating and restoration as well.~~
   * **~~Global Servers Load Balancing (GSLB)~~**~~: Our design uses global server load balancing (GSLB) to handle our system traffic. It ensures intelligent request distribution among different global servers, especially in the case of on-site failures.~~
   * **~~Rate Limiter~~**~~: We also apply a limit on the requests from clients to secure the intrinsic points of failures. To protect the system against DoS attacks, we use rate limiters between the client and web servers to limit each user’s resource allocation. This will ensure a good and smooth traffic influx and mitigate the exploitation of system resources.~~
3. Scalability

* **~~Horizontal database sharding~~**~~: Our design is scalable because our data can easily be distributed among horizontally sharded databases. We can employ a consistent hashing scheme to balance the load between the application and database layers.~~
* **~~Choice of NoSQL database (MongoDB) – supports Horizontal Scaling~~**~~: Scaling a traditional relational database horizontally is a daunting process and poses challenges to meeting our scalability requirements. We want to scale and automatically distribute our system’s data across multiple servers. For this requirement, a NoSQL database would best serve our purpose.~~
* **Addition of Servers and Databases:** We can add more servers to application service layers to make the scalability better and handle numerous requests from the clients. We can also increase the number of databases to store the growing users’ data.

1. Latency
   * **~~Choice of Database (MongoDB)~~** ~~- Our system is redirection-heavy. Writing on the database is minimal compared to reading. We deliberately chose MongoDB because of its low latency and high throughput in reading-intensive tasks.~~
   * **Distributed Cache** - The deployment of a distributed cache in our design also ensures that the system redirects the user with the minimum delay possible.
   * **Content Delivery Networks (CDNs)**
2. Durability
   * We have persistent storage that maintains the backup of the data so any uploaded content (photos and videos) never gets lost.
3. Consistency
   * We have used storage like blob stores and databases to keep our data consistent globally.
4. Reliability
   * Our databases handle [replication](https://www.educative.io/collection/page/10370001/4941429335392256/5241733675220992) and redundancy, so our system stays reliable and data is not lost. The load balancing layer routes requests around failed servers.

11. Additional Details:

1. Load Balancers:
   * We could use a simple Round Robin approach that distributes incoming requests equally among backend servers.
   * A problem with Round Robin LB is that we do not consider the server load. As a result, if a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries the backend server about its load and adjusts traffic based on that.
2. Caching:

* **Which Cache**: We can use any off-the-shelf solution like Memcached, which can store full URLs with their respective hashes.
* **How much cache memory should we have?** We can start with 20% of daily traffic and, based on clients’ usage patterns, we can adjust how many cache servers we need.
* **Which cache eviction policy would best fit our needs?** When the cache is full, and we want to replace a link with a newer/hotter URL, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system.