Detailed Design of Instagram

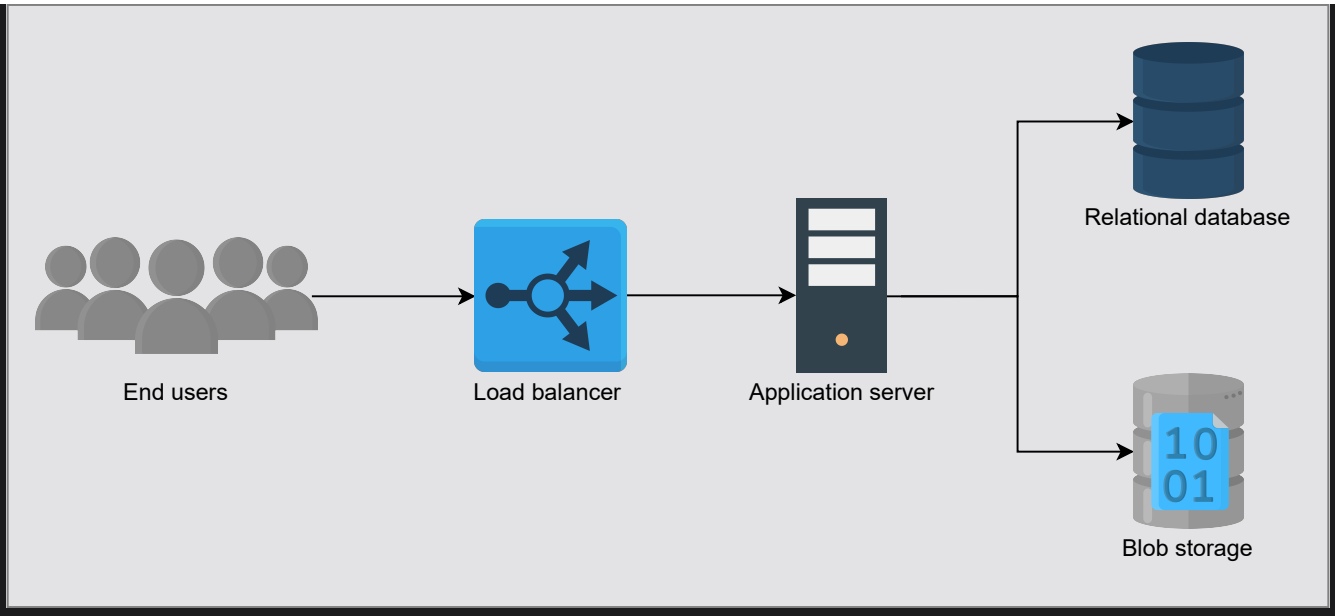
Explore the design of Instagram in detail and understand the interaction of various components.

We'll cover the following

Add more components

Let’s add a few more components to our design:

- **Load balancer**: To balance the load of the requests from the end users.
- **Application servers**: To host our service to the end users.
- **Relational database**: To store our data.
- **Blob storage**: To store the photos and videos uploaded by the users.



Adding components to design

Upload, view, and search a photo

The client requests to upload the photo, load balancer passes the request to any of the application servers, which adds an entry to the database. An update that the photo is stored successfully is sent to the user. If an error is encountered, the user is communicated about it as well.

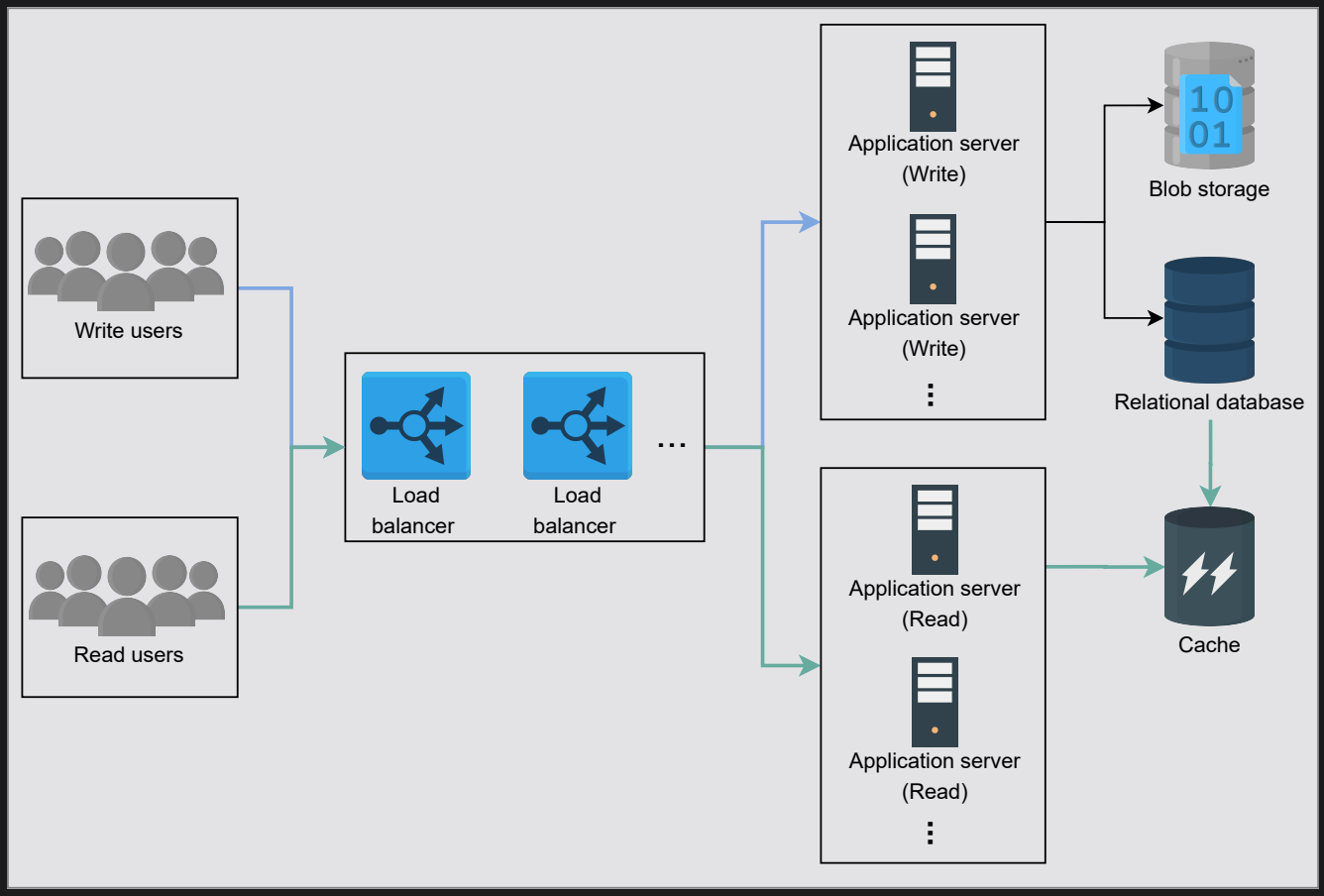
The photo viewing process is also similar to the above-mentioned flow. The client requests to view a photo, and an appropriate photo that matches the request is fetched from the database and shown to the user. The client can also provide a keyword to search for a specific image.

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 fo the user, while the write service helps upload content to the system.

We also need to cache the data to handle millions of reads. It improves the user experience by making the fetching process fast. 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.

The updated design is as follows:



Various operations on photos

Generate a timeline

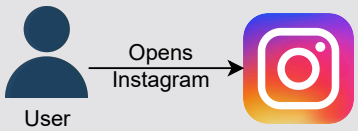
Now our task is to generate a user-specific timeline. Let's explore various approaches and the advantages and disadvantages to opt for the appropriate strategy.

The 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.

We can substantially reduce user-perceived latency by generating the timeline offline. For example, we define a service that fetches the relevant data for the user before, and as the person opens Instagram, it displays the timeline. This decreases the latency rate to show the timeline. Let’s take a look at the slides below to understand the problem and its solution.



A user opens Instagram

1 of 5

Point to Ponder

Question

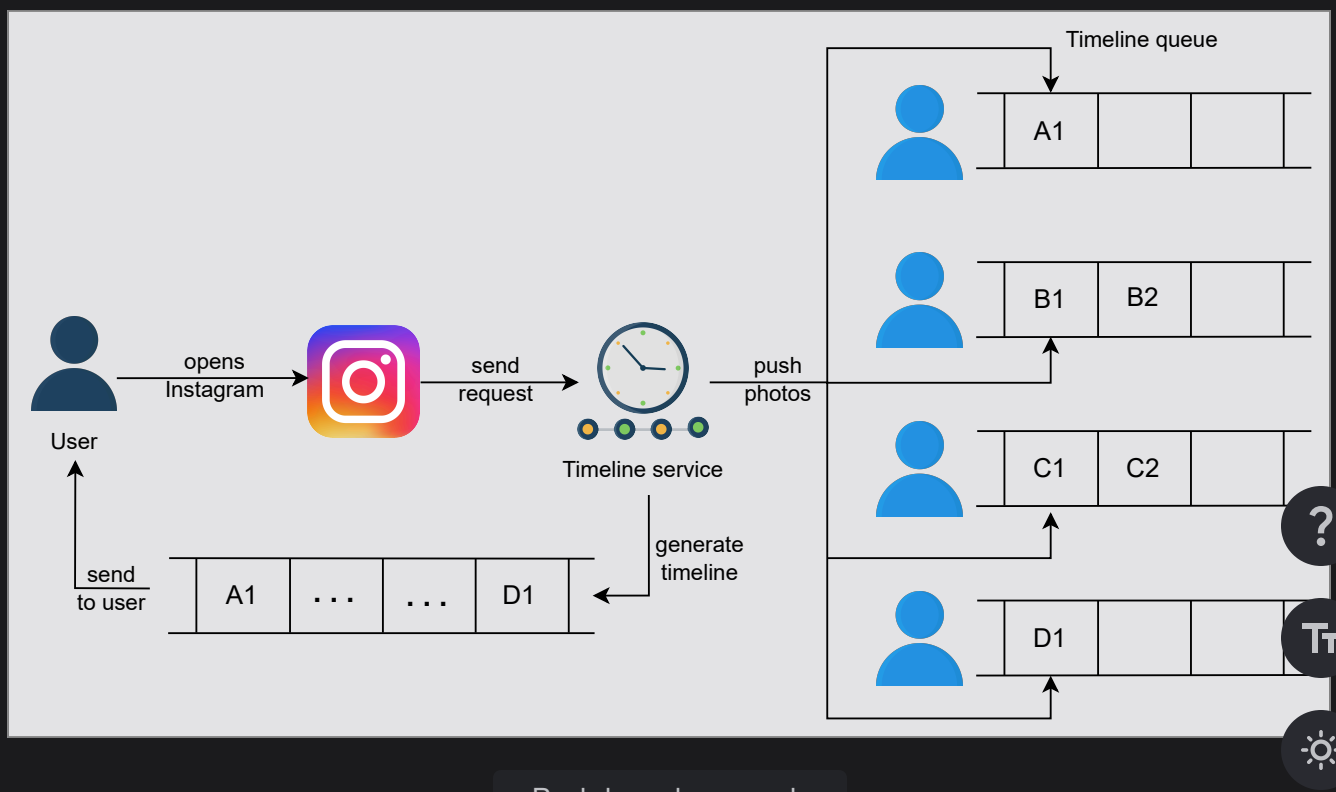
What are the shortcomings of the pull approach?

Show Answer

The 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.

Now we only need to fetch the data that is pushed towards that particular user to generate the timeline. The push approach has stopped a lot of requests that return empty results when followed users have no post in a specified time.



Push-based approach

Point to Ponder

Question

What are the shortcomings of the push approach?

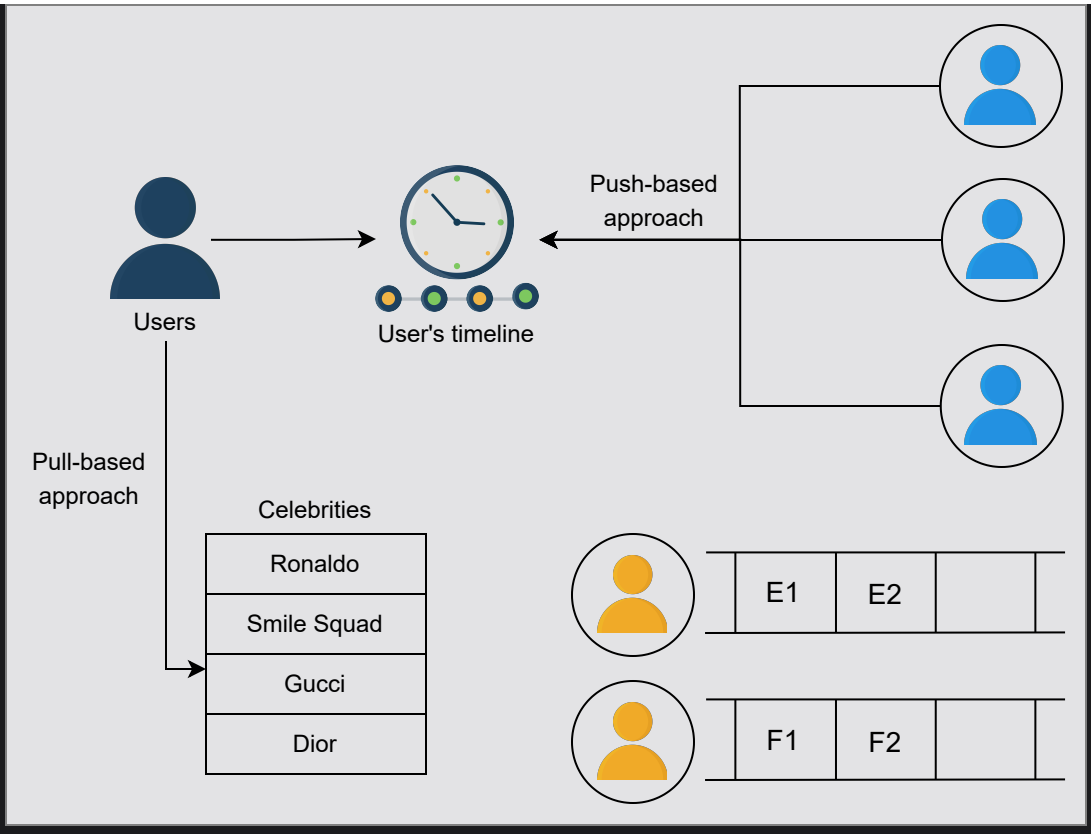
Show Answer

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.



Hybrid approach

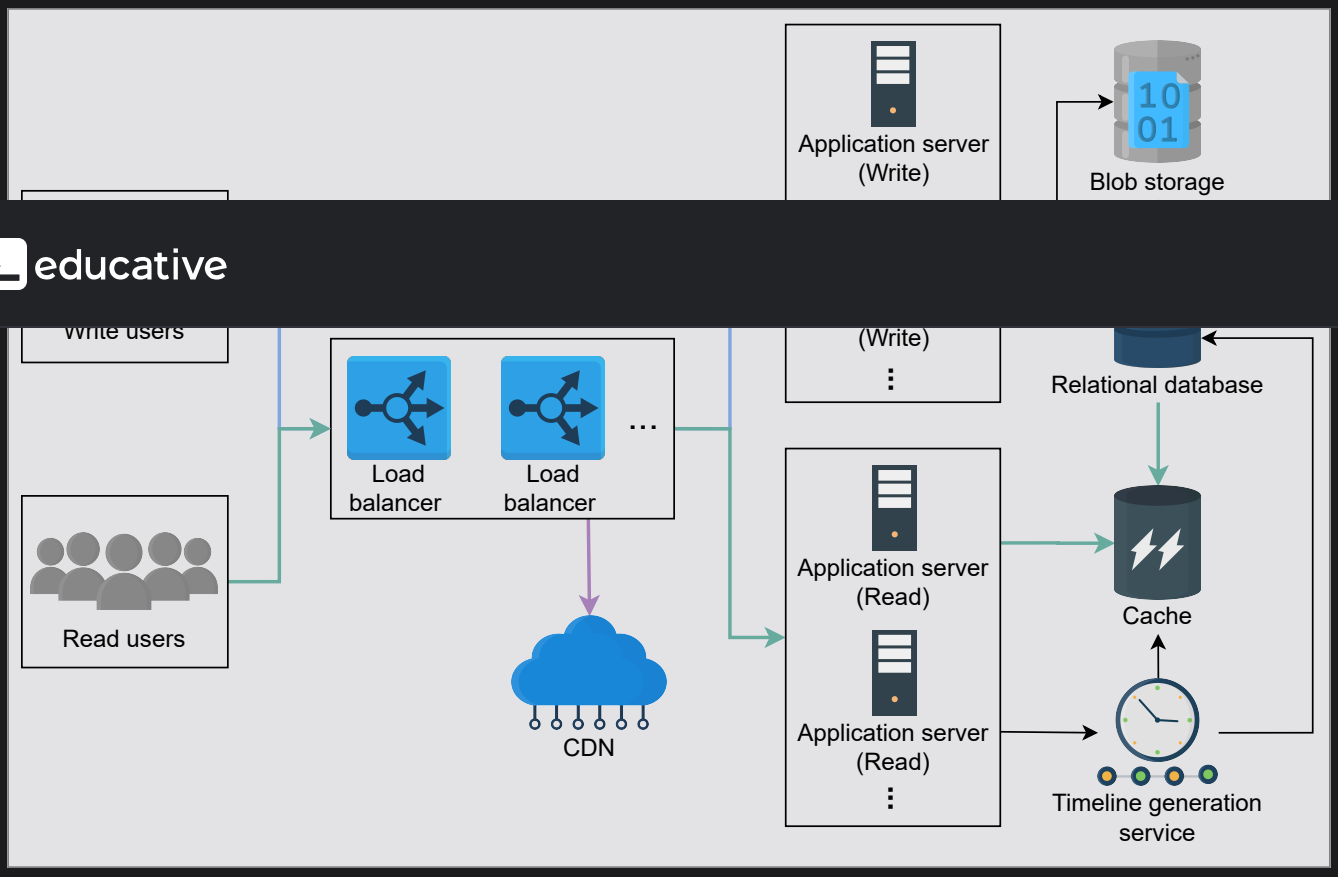
We have used the method which generates the timeline, but where do we store the timeline? We store a user's timeline against a `userID` in a key-value store. Upon request, we fetch the data from the key-value store and show it to the user. The key is `userID`, while the value is timeline content (links 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.

We can add a new feature called story to 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.

Finalized design

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" for the details). The CDN helps our system to be available to millions of concurrent users and minimizes latency.

The final design is given below:



The final design of Instagram

Point to Ponder

Question

How can we count millions of interactions (like or view) on a celebrity post?

Show Answer

Ensure non-functional requirements

We evaluate the Instagram design with respect to its non-functional requirements:

- **Scalability**: 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.
- **Latency**: The use of cache and CDNs have reduced the content fetching time.
- **Availability**: We have made the system available to the users by using the storage and databases that are replicated across the globe.
- **Durability:** We have persistent storage that maintains the backup of the data so any uploaded content (photos and videos) never gets lost.
- **Consistency**: We have used storage like blob stores and databases to keep our data consistent globally.
- **Reliability**: Our databases handle replication and redundancy, so our system stays reliable and data is not lost. The load balancing layer routes requests around failed servers.

Conclusion