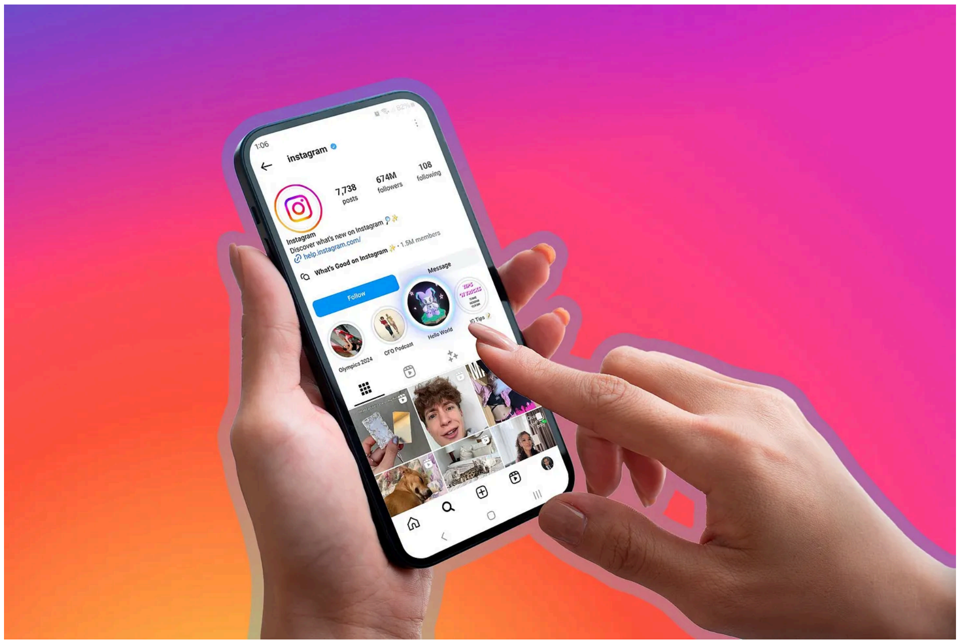
Design Instagram - System Design Interview

ASHISH PRATAP SINGH
MAR 20, 2025 · PAID

65 6 Share

With over **2 billion** monthly active users, **Instagram** is the 3rd most popular social network after Facebook and YouTube.



source: https://www.makeuseof.com/tag/what-are-instagram-highlights/

It enables users to upload photos and videos, interact with content, while handling hundreds of millions of daily visitors, managing petabytes of data, billions of views, while maintaining **low latency** and **high availability**.

Given its scale and complexity, designing Instagram is a popular system design interview question.

While Instagram supports a wide range of features including direct messaging, Ree and Stories—this article will primarily focus on **the core functionality of photo anc video sharing**.

We'll walk through every step of the design—from requirements and high-level architecture to database and API design—before diving deep into core use cases.

1. Requirement Clarification

Before diving into the design, lets outline the functional and non-functional requirements.

Functional Requirements

1. Users can **upload** photos and videos.
2. Users can add **captions** to their posts.
3. Users can **follow/unfollow** other users.
4. Users can **like**, **share**, and **comment** on posts.
5. Support for multiple images/videos in a single post (carousel).
6. Users can view a **personalized feed** consisting of posts from accounts they follo
7. Users can **search** by username and hashtag.

Out of Scope

1. Direct messaging.
2. Short-form video content (Reels).
3. Push Notifications for likes, comments, and follows.

Non Functional Requirements

1. **Low Latency:** The feed should load fast (~100ms).
2. **High Availability:** The system should be available 24/7 with minimal downtime
3. **Eventual Consistency:** A slight delay in users seeing the latest posts from accounts they follow is acceptable.
4. **High Scalability:** Handle millions of concurrent users and billions of posts.
5. **High Durability:** The uploaded photos/videos shouldn't be lost.

2. Capacity Estimation

User Base

- **Total Monthly Active Users (MAUs):** 2 billion
- **Daily Active Users (DAUs): → 500 million users/day**

Estimating Read & Write Requests

Post Uploads (Writes)

- 100M media uploads/day
- Each upload generates metadata writes (DB + cache)
- **Total write requests:** 100M uploads + 100M metadata writes = 200M writes/day

Feed Reads

- Assume an average user scrolls through 100 posts per session
- 500 million DAUs × 100 posts viewed = 50B feed requests/day
- Assuming 80% of feed reads are served from cache, backend reads = 10B D reads/day

Estimating Storage Requirements

Assumptions

- 20% of DAUs (100M) upload media every day
- 80% of uploads are photos, 20% are videos
- **Average photo size:** 1MB
- **Average video size:** 10 MB

Daily Storage Calculation

- **Photos:** (100M × 80%) × 1 MB = 80 TB/day
- **Videos:** (100M × 20%) × 10 MB = 200 TB/day
- **Total storage per day:** 280 TB/day

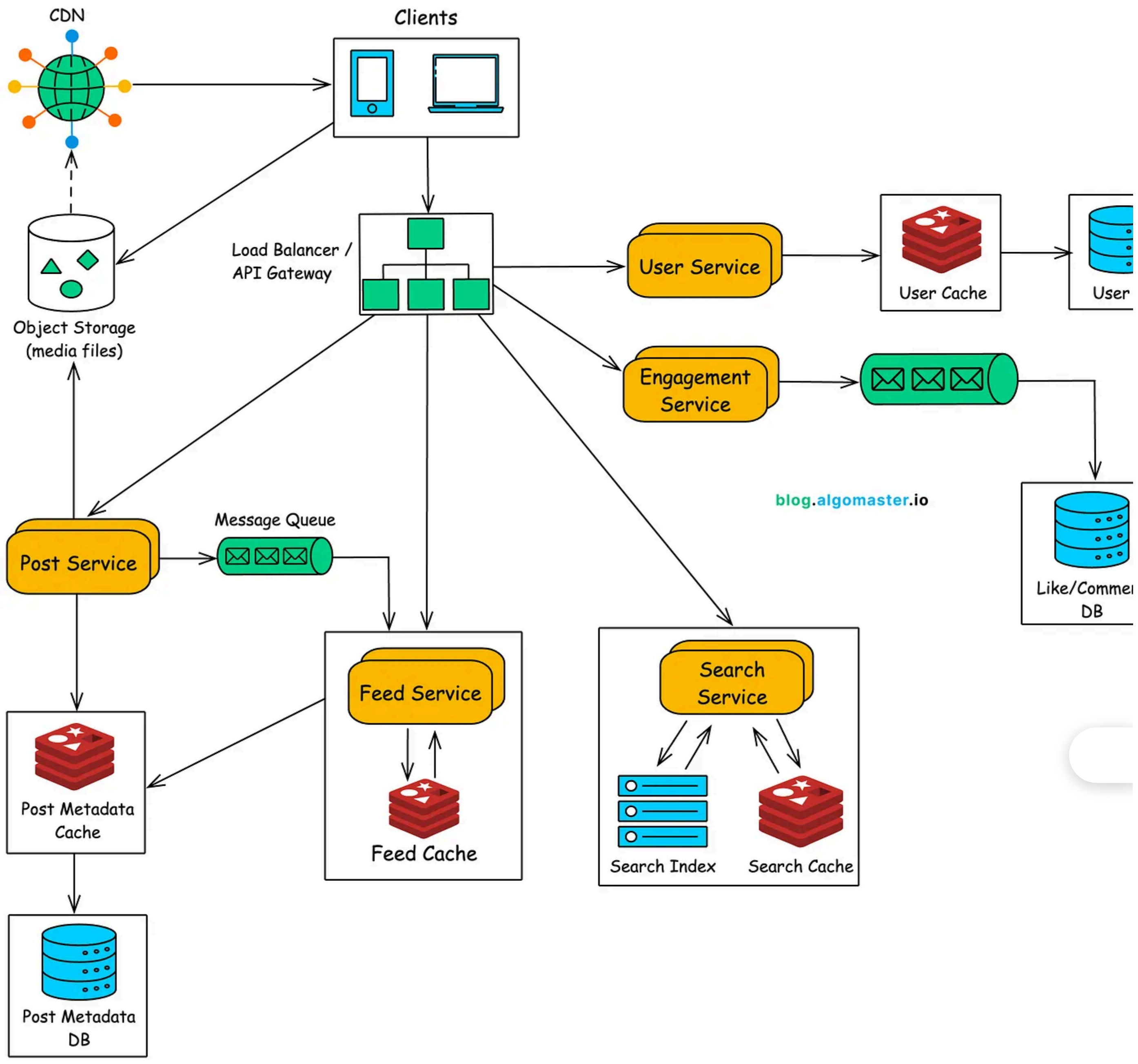
Database Storage

- **Metadata per post:** ~500 bytes (caption, timestamp, author, engagement co
- **Total posts in a year:** 100M × 365 = 36B posts
- **Metadata storage per year:** 90 TB/year

Caching Requirements

- **Hot cache size:** Store recent & popular 1 billion posts
- Assume each cached post takes 2 KB (post data + engagement counts)
- Cache size = 2 TB for active posts (Redis/Memcached)

3. High Level Design



Components:

1. Clients (Web, Mobile)

- Users interact with the platform via web browsers or mobile apps.
- The client applications handle video playback, user interactions (likes, commer and UI rendering.

- They communicate with backend services through an **API Gateway** or **Load Balancer**.

2. Load Balancer / API Gateway

- Acts as the single entry point for all client requests.
- Distributes incoming traffic across multiple service instances to ensure **high availability** and **scalability**.
- Enforces **rate limiting**, **authentication**, and **authorization** before forwarding requests to downstream services.

3. User Service

- Stores and manages **user authentication, profile data, and social connections** (follow/unfollow).

4. Post Service

- Handles **photo/video uploads** and stores metadata (caption, user info, timest
- Coordinates the upload of media files from users device to **Object Storage** (**e.g. AWS S3**) and updates metadata in a **database**.
- Uses a message queue (e.g., **Kafka**) **to notify the Feed Service** when a new post created.

5. Feed Service

- Precomputes and stores user feeds in a high-performance cache (e.g., Redis, Memcached) to enable fast retrieval.
- Queries the database if a feed is not cached.

6. Engagement Service

- Manages **likes, comments, and shares**.

- Writes engagement data to a high-throughput database asynchronously via a message queue.

7. Search Service

- Allows users to search for other users, hashtags, and posts.
- Uses **Elasticsearch** to index and retrieve data quickly.
- Supports **autocomplete and full-text search** for improved user experience.

8. Message Queue

- Decouples services and ensures event-driven processing.
- Notifies the **Feed Service** of new posts.
- Updates engagement data asynchronously.

9. Object Storage & CDN

- Photos/videos are stored in a distributed object Storage (S3, Google Cloud Storage).
- A CDN (**Cloudflare, AWS CloudFront**) ensures fast delivery globally.

4. Database Design

A large-scale content platform like Instagram requires handling both **structured da** (e.g., user accounts, post metadata) and **unstructured/semistructured data** (e.g., pho videos, search indexes).

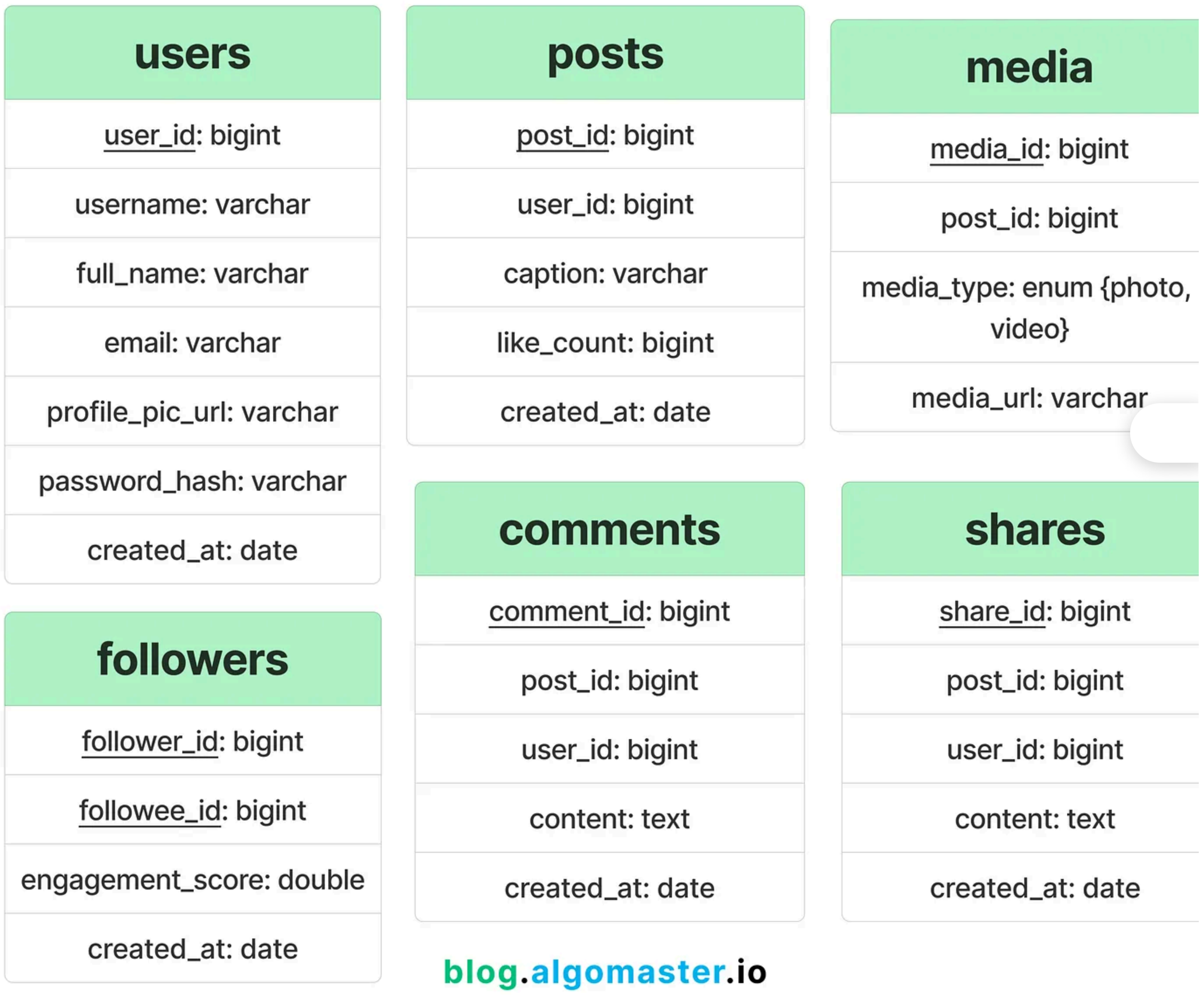
Typically, you'll combine multiple database solutions to handle different workloads

Given the requirements, we will use a **relational database** (**e.g., PostgreSQL, M**
for structured data and a **NoSQL database** (**Cassandra, DynamoDB, or Elasticsear**

for feed storage and search indexing.

4.1 Relational Database for Structured Data

Given the structured nature of user profiles and posts metadata, a relational databa (like **PostgreSQL** or **MySQL**) is often well-suited.



users
user_id: bigint
username: varchar
full_name: varchar
email: varchar
profile_pic_url: varchar
password_hash: varchar
created_at: date

posts
post_id: bigint
user_id: bigint
caption: varchar
like_count: bigint
created_at: date

media
media_id: bigint
post_id: bigint
media_type: enum {photo, video}
media_url: varchar

followers
follower_id: bigint
followee_id: bigint
engagement_score: double
created_at: date

comments
comment_id: bigint
post_id: bigint
user_id: bigint
content: text
created_at: date

shares
share_id: bigint
post_id: bigint
user_id: bigint
content: text
created_at: date

- **Users Table:** Stores user account details.
- **Posts Table:** Stores metadata related to posts.

- **Media Table:** Stores photo/video metadata, but not the actual files.
- **Comments Table:** Stores post comments.
- **Shares Table:** Stores post shares.
- **Followers Table:** Maintains the follow/unfollow relationship. Stores engageme score from followers to help with ranking posts in the feed.

4.2 NoSQL Databases for High-Volume Data

While relational databases are ideal for structured data, they struggle with high-velocity writes and large scale distributed workloads. NoSQL databases like **Cassandra**, **DynamoDB, or Redis** provide horizontal scalability and high availabilit

To reduce **feed generation latency**, a **denormalized feed table** stores precomputed timelines:

```
{
  "user_id": 56789,
  "feed": [
    {"post_id": 111, "user_id": 123, "media_url": "s3://path1", 
"caption": "Hello world"},
    {"post_id": 112, "user_id": 234, "media_url": "s3://path2", 
"caption": "Sunset view"}
  ]
}
```

- Updated **asynchronously** via **Kafka** when a user posts.
- Cached **in Redis for quick retrieval**.

Using Graph Databases for Social Connections

To support complex relationship queries, such as mutual friends, suggested followe and influencer ranking, we can use a **graph database** like Neo4j or Amazon Neptun

They efficiently model follower-following relationships with nodes and edges.

Example Query: "People You May Know"

```
MATCH (me:User {id:12345})-[:FOLLOWS]->(friend)-[:FOLLOWS]->(suggested)
WHERE NOT (me)-[:FOLLOWS]->(suggested)
RETURN suggested LIMIT 5
```

This allows **real-time friend suggestions** without complex SQL joins.

4.3 Search Indexes

To support fast and scalable search queries, we can leverage **Elasticsearch**, a distributed, real-time search engine optimized for full-text searches.

Each user profile and post metadata can be stored as a document in an Elastics index, allowing quick lookups and advanced filtering.

Example: Storing User Data in Elasticsearch

```
{
  "user_id": 12345,
  "username": "john_doe",
  "full_name": "John Doe",
  "bio": "Photographer | Traveler"
}
```

To support trending hashtags and keyword searches, we can store **hashtags** in a separate Elasticsearch index.

Example:

```
{
  "hashtag": "#travel",
  "post_count": 1500000,
  "last_used": "2025-03-20T12:00:00Z"
}
```

4.4 Media Storage

Instagram handles **petabytes of photos/videos**, requiring a **durable and low-latenc storage solution**.

A **distributed object storage system**, such as **Amazon S3**, is well-suited for storing media files. It supports **pre-signed URLs**, enabling users to upload media directly without routing through application servers, reducing load and latency.

To ensure **high durability**, media files are stored in multiple replicas across differer data centers, protecting against data loss.

To further optimize read latency, content can be cached closer to users using a **Content Delivery Network** (CDN) like **Cloudflare or Amazon CloudFront**. This reduces load times and improves the user experience, especially for frequently accessed media.

5. API Design

5.1 Get User Profile

```
GET /api/v1/users/{user_id}
Authorization: Bearer JWT_TOKEN
```

Response:

```
{
  "user_id": 12345,
  "username": "john_doe",
  "full_name": "John Doe",
  "profile_pic": "https://cdn.example.com/profile.jpg",
  "followers_count": 200,
  "following_count": 150
}
```

5.2 Follow a User

```
POST /api/v1/users/{user_id}/follow
Authorization: Bearer JWT_TOKEN
```

5.3 Create a New Post

```
POST /api/v1/posts
Authorization: Bearer JWT_TOKEN
Content-Type: multipart/form-data
```

Form Data:

```
caption: "Sunset at the beach"
media: [image1.jpg, video1.mp4]
```

Response:

```
{
  "post_id": 98765,
  "message": "Post uploaded successfully"
}
```

5.4 Get a Post by ID

```
GET /api/v1/posts/{post_id}
```

Response:

```
{
  "post_id": 98765,
  "user_id": 12345,
  "caption": "Sunset at the beach",
  "media": [
    "https://cdn.example.com/photo1.jpg",
    "https://cdn.example.com/video1.mp4"
  ],
  "likes_count": 500,
  "comments_count": 120
}
```

5.5 Get User Feed

```
GET /api/v1/feed?page=1&limit=10
Authorization: Bearer JWT_TOKEN
```

Response:

```
[
  {
    "post_id": 123,
    "user": {
      "user_id": 56789,
      "username": "travel_lover"
    },
    "caption": "Exploring the mountains!",
    "media": ["https://cdn.example.com/photo123.jpg"],
    "likes_count": 1200,
    "comments_count": 85
  },
  {
    "post_id": 124,
    "user": {
      "user_id": 98765,
      "username": "foodie_dude"
    },
    "caption": "Delicious sushi!",
    "media": ["https://cdn.example.com/photo124.jpg"],
    "likes_count": 980
  }
]
```

5.6 Like a Post

```
POST /api/v1/posts/{post_id}/like
Authorization: Bearer JWT_TOKEN
```

5.7 Comment on a Post

```
POST /api/v1/posts/{post_id}/comment
Authorization: Bearer JWT_TOKEN
Content-Type: application/json
```

```
{
  "content": "Amazing shot!"
}
```

5.8 Get Comments for a Post

```
GET /api/v1/posts/{post_id}/comments?page=1&limit=10
```

Response:

```
[
  {
    "comment_id": 65432,
    "user": {
      "user_id": 12345,
      "username": "john_doe"
    },
    "content": "Amazing shot!",
    "created_at": "2025-03-18T12:34:56Z"
  },
  {
    "comment_id": 65433,
    "user": {
      "user_id": 67890,
      "username": "nature_lover"
    },
    "content": "Breathtaking view!"
  }
]
```

5.9 Search Users

```
GET /api/v1/search/users?q=john
```

Response:

```
[
  {
    "user_id": 12345,
    "username": "john_doe",
    "full_name": "John Doe",
    "profile_pic": "https://cdn.example.com/profile.jpg"
  }
]
```

6. Design Deep Dive

6.1 Photo/Video Upload

1. **User Initiates the Upload**
    a. The user selects one or more photos or videos and enters a caption.
    b. The client (mobile app/web browser) sends an upload request to the API Gateway.
2. **API Gateway Handles the Request**
    a. The API gateway authenticates and validates the request.
    b. Routes the request to the Post Service.
3. **Post Service Generates a Pre-signed URL**
    a. Instead of uploading media directly through the backend, the Post Service generates pre-signed URLs from Object Storage (one per media file).

   b. It sends the pre-signed URLs back to the client.

4. **Client Uploads Media to Object Storage**
   a. The client directly uploads each file in parallel to Object Storage via the pr
   signed URLs.
   b. This reduces backend load and enables faster parallel uploads.
   c. Once all uploads are complete, the client sends a confirmation request to tl
   backend with all media URLs.

5. **Post Service Saves Metadata in the Database**
   a. The Post Service stores post metadata (caption, timestamp, user ID) in the Posts table and stores each media file separately in the Media Table.

6. **Kafka Publishes a "New Post" Event**
   a. The Post Service sends an event to Kafka, notifying the Feed Service.

6.2 Newsfeed Generation

Since users follow both normal users and celebrities, the system must mix posts efficiently.

Fan-out-on-write (Push Model) for Normal Users

For **normal users** with a manageable number of followers, we use **fan-out-on-write** meaning posts are **pushed** to followers' feeds at the time of posting.

How It Works

1. User A posts a new photo/video.
2. The Post Service sends an event to Kafka, notifying the Feed Service.
3. The Feed Service identifies the users followers (e.g., 500 followers).

4. The post is immediately inserted into each follower's timeline, stored in Redis cache).
5. When followers open their feeds, posts are instantly available, ensuring low-latency reads.

Example: LPUSH - Add Post to Followers' Feeds

- User `12345` (John Doe) posts a new photo
- He has 500 followers
- The Feed Service pushes this post to all 500 followers' feeds

```
LPUSH feed:56789 "{'post_id': 98765, 'author': 'john_doe', 'media_url':
'https://cdn.instagram.com/photo98765.jpg', 'caption': 'Sunset at the
beach!', 'timestamp': '2025-03-20T14:30:00Z'}"

LPUSH feed:67890 "{'post_id': 98765, 'author': 'john_doe', 'media_
'https://cdn.instagram.com/photo98765.jpg', 'caption': 'Sunset at the
beach!', 'timestamp': '2025-03-20T14:30:00Z'}"

LPUSH feed:78901 "{'post_id': 98765, 'author': 'john_doe', 'media_url':
'https://cdn.instagram.com/photo98765.jpg', 'caption': 'Sunset at the
beach!', 'timestamp': '2025-03-20T14:30:00Z'}"

...
```

Here, John's post is pushed to the feeds of followers `56789`, `67890`, and `78901`, alc with 497 other followers.

Example: Fetching a User's Feed (LRANGE - Get Recent Posts)

```
LRANGE feed:56789 0 9  # Fetch the latest 10 posts from user 56789's
feed
```

Benefits:

- Super-fast reads since followers' feeds are pre-loaded.
- Works efficiently for small and medium-sized accounts.

Challenges:

- Becomes inefficient for users with millions of followers (e.g., celebrities).
- Writing a post requires copying it to potentially millions of timelines, leading t high write amplification.

Fan-out-on-read (Pull Model) for Celebrities

For **celebrities and influencers**, where a single post may need to reach **millions of followers**, preloading into every follower's feed is impractical.

Instead, a **fan-out-on-read** (**pull model**) is used.

How It Works

1. When a user requests their newsfeed, the Feed Service dynamically retrieves:
   - Normal users' posts from Redis (precomputed feeds).
   - Celebrity posts from a hot cache (Redis) or a persistent store (PostgreSQL).
2. The system merges both types of posts in real-time before serving the feed.

Benefits:

- Avoids massive write operations, keeping the system scalable.
- Ensures fresh data when users request feeds.

Challenges:

- Slightly higher read latency than the push model.

- Requires caching optimization to reduce database lookups.

6.3 Search

Indexing New Content

1. **A New Post/User is Created**
   a. A user uploads a post or creates an account.
   b. The Post/User stores metadata in the database.
   c. The Post/User Service publishes an event to Kafka.
2. **Search Service Updates Elasticsearch Index**
   a. The Search Service consumes Kafka events and adds new users, posts, or hashtags to Elasticsearch.

Search Request

1. **User Initiates a Search Request**
   a. The user types a query in the search bar (e.g., `"john_doe"` or `"#travel"`)
   b. The client (mobile/web) sends a request. The request is routed via the API Gateway to the Search Service.
2. **Search Service Queries Elasticsearch**
   a. The Search Service first checks Redis Cache for recent searches. If not four queries Elasticsearch for relevant results.
   b. Elasticsearch performs full-text search, prefix matching and ranking based engagement/popularity.
3. **Elasticsearch Returns Results**
   a. Elasticsearch returns ranked results matching the query.
   b. The Search Service formats the response.

```
{
  "users": [
    { "user_id": 12345, "username": "john_doe", "full_name":
"John Doe", "profile_pic": "https://cdn.example.com/john.jpg" },
    { "user_id": 67890, "username": "johnny_depp", "full_name":
"Johnny Depp", "profile_pic":
"https://cdn.example.com/johnny.jpg" }
  ],
  "hashtags": [
    { "tag": "travel", "post_count": 10M },
    { "tag": "travelphotography", "post_count": 5M }
  ]
}
```

4. **Search Results are Cached in Redis**
   a. The Search Service caches frequent queries in Redis for faster lookups.
   b. Next time a user searches for the same query, the result is served from Redi instantly.

6.4 Like, Comments and Shares

The **Engagement Service** processes like, comment and share requests.

It sends a Kafka event to update the DB asynchronously.

Like event:

```
{
  "event": "POST_LIKED",
  "user_id": 12345,
  "post_id": 67890
}
```

Share event:

```
{
  "event": "POST_SHARED",
  "user_id": 12345,
  "post_id": 67890
}
```

Comment event:

```
{
  "event": "POST_COMMENTED",
  "user_id": 12345,
  "post_id": 67890,
  "comment_id": 99999,
  "content": "Amazing shot!"
}
```

To optimize the latency for popular posts, we can cache like / share count and to comments.

7. Addressing Scalability, Availability anc Durability

7.1 Scalability

Scalability ensures the system can handle increasing load without degrading performance.

Horizontal Scaling (Scale Out)

- Use distributed databases (Cassandra, DynamoDB) to distribute data across no

- Deploy multiple instances of services behind a load balancer to handle user requests.

Sharding

- Implement sharding to split large datasets.
- User Data → Shard by `user_id mod N`
- Posts → Shard by `post_id mod N`
- Followers Table → Shard by `follower_id mod N`

Microservices Architecture

- Break the system into independent services (e.g., Feed Service, Post Service, Us Service) to improve maintainability and scalability.
- Use message queues (Kafka, RabbitMQ) to handle high-throughput operations asynchronously (e.g., processing notifications, updates, and feed generation).

7.2 Availability

Availability ensures that Instagram remains accessible 24/7, even in the face of failu Given its global user base, the platform must achieve atleast 99.99% uptime.

Redundancy & Replication

- Maintain replicated databases across multiple regions (e.g., PostgreSQL replica Cassandra multi-region clusters).
- Deploy multiple application servers across different availability zones (AZs).

Failover Mechanisms

- Use automatic failover in databases (e.g., leader-follower setup in PostgreS multi-leader Cassandra clusters).
- Implement circuit breakers to gracefully degrade service if a dependency fails.

7.3 Durability

Durability ensures that data—especially user-generated content (photos, videos, comments, likes)—is never lost, even in case of system failures.

Distributed Object Storage

- Store media in Amazon S3 / Google Cloud Storage, which replicates data acros multiple locations to prevent loss.

Database Replication & Backups

- Use multi-region replication (Cassandra, DynamoDB, PostgreSQL replicas) for disaster recovery.
- Perform regular backups to prevent accidental data loss.

Write-Ahead Logging (WAL) & Event Sourcing

- Implement WAL in databases to ensure changes are recorded before comn
- Use event sourcing to log user actions (e.g., new posts, likes) and rebuild state i necessary.

Thank you for reading!

If you found it valuable, hit a like ❤️ and if you have any questions or suggestions, leave a comment.

I hope you have a lovely day!

See you soon,

Ashish

65 Likes · 6 Restacks

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...