

Instagram System Design

Photo & Video Sharing Social Network at Scale

(New Grad / Junior Engineer Interview Focus | TypeScript/Node.js + Go + PostgreSQL)

0 Executive Summary

System Purpose

Instagram is a **photo and video sharing social network** where users post visual content, follow other users, like/comment on posts, and discover content through hashtags and explore pages. As of 2026, it serves 2 billion monthly active users with 100 million photos/videos uploaded daily.

Core Constraints

- **High read-to-write ratio:** 100:1 (users scroll more than they post)
- **Media-heavy:** Store 500 PB of photos/videos (growing 10 TB/day)
- **Feed generation:** Personalized feed for 2B users (not just reverse chronological)
- **Real-time interactions:** Likes, comments, follows appear within 1 second
- **Global scale:** Users distributed worldwide, need low latency everywhere

Key Architectural Choices

1. **CDN for media delivery** (photos/videos served from edge locations)
2. **PostgreSQL for social graph** (followers, likes, comments)
3. **Redis for feed caching** (pre-computed timelines)
4. **S3-compatible storage** for photos/videos (cheap object storage)
5. **Go for media processing** (resize images, transcode videos)
6. **Node.js for API servers** (handle millions of requests)

Biggest Challenges

- **Feed generation:** Show relevant posts to 2B users (not just "latest from friends")
- **Viral posts:** One post gets 10M likes in 1 hour (hot key problem)
- **Media storage costs:** $500 \text{ PB} \times \$23/\text{TB} = \$11\text{M}/\text{month}$ just for storage
- **Image delivery latency:** Serve images <100ms globally (need CDN)
- **Follow graph queries:** "Show me posts from my 500 followers" efficiently

Interview Focus (New Grad Level)

- Read-heavy system design patterns (caching, denormalization)
- Feed generation basics (fan-out on write vs fan-out on read)
- CDN for media delivery (why not serve directly from backend)
- Basic social graph modeling (followers, likes, comments)
- Image storage and delivery pipeline

1 Problem Definition

What Problem Does Instagram Solve?

Instagram replaces **sharing photos via MMS/email** with a **visual social network**:

1. **Share moments:** Post photos/videos to friends and followers

2. **Discover content:** See what friends are doing, discover new accounts
3. **Social interaction:** Like, comment, share, save posts
4. **Visual storytelling:** Stories (24-hour ephemeral content)
5. **Creator platform:** Build audience, monetize through ads and sponsorships
6. **Messaging:** Direct messages (DMs) for private conversations

Who Are the Users?

- **2 billion monthly active users** (as of 2026)
- **500 million daily active users** (post/browse daily)
- **100 million posts per day** (photos + videos)
- **Use cases:**
 - Personal: Share travel photos, food pics, selfies
 - Creators: Build audience (photographers, influencers, artists)
 - Businesses: Marketing, customer engagement
 - News: Breaking news, live events

Why Existing Solutions Failed

✗ Sharing Photos via MMS/Email

```
Alice takes photo → Compresses to <1 MB for MMS
                    → Sends to 5 friends individually
                    → Friends see low-quality compressed image
                    → No way to comment or like
```

Why it fails:

- **Low quality:** MMS limits to 300 KB, destroys image quality
- **No social context:** Can't see who else liked it, no comments
- **Hard to discover:** Photos buried in email/SMS threads
- **One-to-one:** Must send separately to each person

✗ Facebook (Pre-Instagram)

```
Alice uploads 20 vacation photos
Bob's newsfeed: [Alice's photo 1] [Ad] [Carol's status] [Alice's photo 2] ...
                  (Photos mixed with text posts, hard to browse)
```

Why it needed Instagram:

- **Not photo-first:** Facebook is text-first (status updates), photos are secondary
- **No photo filters:** Can't enhance photos before posting
- **Desktop-focused:** Facebook was desktop-first, Instagram is mobile-first
- **Cluttered:** Newsfeed has too many types of content (status, links, photos, videos)

✗ Flickr (Photo Storage)

```
Alice uploads high-res photo to Flickr
Bob tries to view on phone → Loads slowly (5 MB image)
                                → No social features (just photo albums)
```

Why it failed for social:

- **Storage-focused:** Flickr is a photo backup tool, not a social network
 - **No feed:** Can't browse friends' photos chronologically
 - **Slow on mobile:** High-res images take forever to load on 3G
 - **Professional focus:** Aimed at photographers, not casual users
-

Constraints That Shape the System

1. **Read-heavy:** Users browse feed 100x more than they post (optimize for reads)
 2. **Low latency:** Feed must load within 2 seconds (or users leave)
 3. **Global users:** Serve users in US, India, Brazil, Indonesia (need regional servers)
 4. **Cost efficiency:** Storing 500 PB is expensive, need cheap storage (S3, Glacier)
 5. **Mobile-first:** 90% of traffic from mobile apps (optimize for mobile networks)
 6. **Viral content:** One post can get 10M likes in 1 hour (handle hot keys)
-

2 Requirements

Functional Requirements

Core Features

1. Post photos/videos

- Upload photo (max 10 MB) or video (max 60 seconds)
- Add caption (max 2,200 characters)
- Add location tag (optional)
- Add hashtags (#sunset, #travel)
- Tag other users (@alice, @bob)
- Choose filters (Valencia, Clarendon, Juno, etc.)

2. Home feed (Timeline)

- See posts from people you follow
- Ordered by relevance (not just chronological)
- Infinite scroll (load more posts as you scroll)
- Pull-to-refresh (check for new posts)

3. Explore page

- Discover new content based on interests
- Trending hashtags
- Suggested accounts to follow

4. Interactions

- Like post (double-tap heart)
- Comment on post (text, emojis)
- Share post (send via DM, share to Stories)
- Save post (bookmark for later)

5. Profile

- User bio, profile picture
- Grid of posted photos/videos

- Follower count, following count
- Edit profile (name, bio, website)

6. Follow system

- Follow/unfollow users
- Private accounts (approve follow requests)
- See followers/following lists

7. Stories (24-hour content)

- Post ephemeral photo/video (disappears after 24 hours)
- View friends' stories (swipe through)
- Story rings (colored ring around profile pic = new story)

8. Direct Messages (DMs)

- Send text messages, photos, videos
- Share posts via DM
- Group chats (up to 32 people)

Secondary Features

9. Search

- Search users by username
- Search hashtags (#sunset)
- Search locations (New York, USA)

10. Notifications

- Someone liked your post
- Someone commented on your post
- Someone followed you
- Someone mentioned you in their post

11. Privacy

- Public account (anyone can see posts)
- Private account (must approve followers)
- Block users
- Hide stories from specific users

Non-Functional Requirements

Scale (2026 Numbers)

Metric	Value	Derivation
Monthly Active Users (MAU)	2 billion	Meta's reported data
Daily Active Users (DAU)	500 million	~25% of MAU
Posts per Day	100 million	~0.2 posts per DAU
Photos Stored	100 billion	100M/day × 365 days × 3 years
Videos Stored	10 billion	10% of posts are videos

Total Storage	500 PB	100B photos × 5 MB avg
Feed Requests per Day	10 billion	500M DAU × 20 feed loads/day
Feed Requests per Second (avg)	115,000	10B ÷ 86,400 sec
Feed Requests per Second (peak)	350,000	3× avg during evening hours
Likes per Day	4 billion	500M DAU × 8 likes/day
Comments per Day	500 million	500M DAU × 1 comment/day
CDN Bandwidth	100 PB/day	500M users × 200 MB browsing/day

Latency Targets

Operation	P50	P99	Max Acceptable	Rationale
Load Home Feed	500ms	2s	3s	Users expect instant feed
Image Load (CDN)	50ms	200ms	500ms	Critical for user experience
Video Start (CDN)	100ms	500ms	2s	Buffering acceptable
Post Upload	2s	5s	10s	User waits, show progress bar
Like/Comment	100ms	300ms	1s	Must feel instant
Follow/Unfollow	100ms	500ms	1s	Important interaction
Search Results	200ms	1s	3s	User expects fast search

Feed Load Latency Budget (P99): API request: 100ms (mobile network) Database query (cached feed): 50ms (Redis) Image metadata fetch: 50ms (PostgreSQL) Serialize JSON response: 20ms (API server) Network return: 100ms (mobile network) Client rendering: 200ms (image decode + layout)

Total: 520ms (within 2s target)

Availability & Durability

Requirement	Target	Implementation
API Availability	99.9%	Multi-region, load balancing
Photo Durability	99.99999999%	S3 (11 nines), replicated across regions
Data Loss Window	<1 minute	Write to master + sync replica
Feed Consistency	Eventual (1-5 seconds lag)	Redis cache, async updates

Read-Heavy Workload

Instagram is **heavily read-dominated**:

Writes (posts): 100M/day = 1,200/second Reads (feed loads): 10B/day = 115,000/second

Read-to-write ratio: 100:1

Design Implication: Must optimize for reads (caching, denormalization, CDN)

3 High-Level Architecture

Component Overview

```
```mermaid
graph TB
 subgraph "Client Layer"
 Mobile[Mobile Apps
iOS/Android]
 Web[Web Client
React]
 end

 subgraph "CDN & Edge"
 CDN[CloudFront CDN
Photos/Videos
100 PB/day]
 end

 subgraph "Load Balancing"
 LB[Load Balancer
ALB]
 end

 subgraph "Application Layer"
 API[API Servers
Node.js
1000 instances]
 FeedGen[Feed Generator
Go
Pre-compute feeds]
 end

 subgraph "Storage Layer"
 PG[(PostgreSQL
Users, Posts, Follows
Metadata)]
 Redis[(Redis
Feed Cache
Timeline Cache)]
 S3[(S3 Storage
Photos/Videos
500 PB)]
 end

 subgraph "Background Workers"
 MediaProcessor[Media Processor
Go
Resize, Transcode]
 NotificationWorker[Notification Worker
Node.js
Push Notifications]
 end

 subgraph "Search & Analytics"
 ES[(Elasticsearch
User/Hashtag Search)]
 Analytics[(Analytics DB
Impressions, Engagement)]
 end
```

```

```

Mobile -->|HTTPS| LB
Web -->|HTTPS| LB

Mobile -.->|Load Images| CDN
Web -.->|Load Images| CDN

LB --> API
API --> PG
API --> Redis
API --> ES

API -->|Upload| S3
S3 --> MediaProcessor
MediaProcessor --> S3
MediaProcessor --> CDN

FeedGen --> PG
FeedGen --> Redis

API --> NotificationWorker
NotificationWorker --> Mobile

```

Component Responsibilities

1. Mobile/Web Client

Technology: React Native (Mobile), React (Web)

Responsibilities:

- Display feed (infinite scroll)
- Capture photos/videos (camera integration)
- Apply filters (Instagram filters like Valencia, Clarendon)
- Upload media to API
- Handle optimistic updates (show like immediately, sync in background)
- Cache images locally (reduce CDN bandwidth)

Local Storage:

```

interface LocalCache {
    feedPosts: Post[];           // Last 100 posts seen
    userProfiles: Map<string, User>; // Frequently viewed profiles
    images: Map<string, Blob>;   // Downloaded images (LRU cache, max 50 MB)
}

```

2. CDN (CloudFront)

Purpose: Deliver photos/videos from locations close to users

Why CDN?

- **Latency:** Serving from US server to India user = 300ms, from Mumbai edge = 50ms
- **Bandwidth:** Origin servers don't have 100 PB/day capacity

- **Cost:** CDN data transfer cheaper than direct S3 transfer

CDN Architecture:

```
User in Mumbai → CloudFront Mumbai edge
    → If cache miss → CloudFront Tokyo regional edge
        → If still miss → S3 us-east-1 origin
```

Cache Headers:

```
Cache-Control: public, max-age=31536000, immutable
```

(Photos never change once uploaded, cache forever)

3. API Servers (Node.js)

Technology: Node.js (Express/Fastify)

Responsibilities:

- Handle REST API requests (GET /feed, POST /posts, etc.)
- Authenticate users (JWT validation)
- Fetch data from PostgreSQL, Redis
- Return JSON responses
- Rate limiting (prevent abuse)

Why Node.js?

- I/O-bound tasks (database queries, external API calls)
- Good JSON serialization performance
- Large ecosystem (npm packages)
- Easy to scale horizontally

Example API Endpoint:

```
app.get('/api/feed', async (req, res) => {
  const userId = req.user.id; // From JWT

  // 1. Check Redis cache (pre-computed feed)
  const cachedFeed = await redis.get(`feed:${userId}`);
  if (cachedFeed) {
    return res.json(JSON.parse(cachedFeed));
  }

  // 2. Cache miss → generate feed
  const feed = await generateFeed(userId);

  // 3. Cache for 5 minutes
  await redis.setex(`feed:${userId}`, 300, JSON.stringify(feed));

  res.json(feed);
});
```

4. Feed Generator (Go)

Technology: Go (worker pool pattern)

Responsibilities:

- Pre-compute home feeds for active users
- Fan-out posts to followers' feeds (write on post)
- Rank posts by relevance (simple ML model: recency + engagement)
- Update Redis cache with computed feeds

Why Go?

- CPU-bound task (sorting, ranking millions of posts)
- Good concurrency (goroutines handle many users in parallel)
- Fast execution (feed generation must be <100ms)

Fan-Out Strategy (explained in detail later):

```
Alice posts photo → Feed Generator fans out to 1,000 followers  
→ Each follower's feed updated in Redis
```

5. PostgreSQL (Metadata Storage)

Purpose: Store structured data (users, posts, likes, comments, follows)

Schema Overview:

```
-- Users  
CREATE TABLE users (  
    id BIGSERIAL PRIMARY KEY,  
    username VARCHAR(30) UNIQUE NOT NULL,  
    email VARCHAR(255) UNIQUE NOT NULL,  
    full_name VARCHAR(100),  
    bio TEXT,  
    profile_pic_url TEXT,  
    is_private BOOLEAN DEFAULT FALSE,  
    created_at TIMESTAMPTZ DEFAULT NOW()  
);  
  
-- Posts  
CREATE TABLE posts (  
    id BIGSERIAL PRIMARY KEY,  
    user_id BIGINT NOT NULL REFERENCES users(id),  
    image_url TEXT NOT NULL,  
    caption TEXT,  
    location VARCHAR(255),  
    created_at TIMESTAMPTZ DEFAULT NOW(),  
    like_count INT DEFAULT 0,  
    comment_count INT DEFAULT 0  
);  
  
-- Follows  
CREATE TABLE follows (
```

```

follower_id BIGINT NOT NULL REFERENCES users(id),
followee_id BIGINT NOT NULL REFERENCES users(id),
created_at TIMESTAMPTZ DEFAULT NOW(),
PRIMARY KEY (follower_id, followee_id)
);

-- Likes
CREATE TABLE likes (
    user_id BIGINT NOT NULL REFERENCES users(id),
    post_id BIGINT NOT NULL REFERENCES posts(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (user_id, post_id)
);

-- Comments
CREATE TABLE comments (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL REFERENCES users(id),
    post_id BIGINT NOT NULL REFERENCES posts(id),
    text TEXT NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

```

Why PostgreSQL?

- Need relational joins (get posts + user info in one query)
- ACID transactions (like must increment like_count atomically)
- Mature ecosystem, well-understood

6. Redis (Feed Cache)

Purpose: Cache pre-computed feeds for fast access

Data Structures:

```

// User's home feed (list of post IDs, sorted by timestamp)
feed:{userId} → [post123, post122, post121, ...]

// Post engagement counts (hash)
post:{postId}:stats → {likes: 1500, comments: 42, shares: 8}

// User session
session:{sessionId} → {userId: 123, expiresAt: ...}

// Rate limiting
ratelimit:{userId}:posts → {count: 5, window: 3600}

```

Why Redis?

- Sub-millisecond latency (critical for feed load)
- Sorted sets for ranked feeds
- Atomic operations (increment like count)

7. S3 Storage (Photos/Videos)

Purpose: Store original and processed photos/videos

Bucket Structure:

```
instagram-media/
  photos/
    original/
      2026/01/28/{userId}/{postId}_original.jpg
    resized/
      2026/01/28/{userId}/{postId}_150x150.jpg   (thumbnail)
      2026/01/28/{userId}/{postId}_640x640.jpg   (feed)
      2026/01/28/{userId}/{postId}_1080x1080.jpg (full)
  videos/
    original/
      2026/01/28/{userId}/{postId}_original.mp4
    transcoded/
      2026/01/28/{userId}/{postId}_720p.mp4
      2026/01/28/{userId}/{postId}_480p.mp4
```

Why S3?

- Cheap storage (\$0.023/GB/month)
 - 99.99999999% durability (won't lose photos)
 - Integrates with CloudFront CDN
 - Lifecycle policies (move old photos to Glacier for 90% cost savings)
-

8. Media Processor (Go)

Purpose: Resize images and transcode videos after upload

Workflow:

1. User uploads 5 MB original image to S3
2. S3 triggers Lambda/worker (via SNS)
3. Media Processor downloads original
4. Resize to 150x150 (thumbnail), 640x640 (feed), 1080x1080 (full)
5. Upload resized versions to S3
6. Invalidate CDN cache (if needed)

Why Go?

- CPU-bound task (image processing)
- Good libraries (image/jpeg, ffmpeg wrapper)
- Fast execution (process 1,000 images/sec per instance)

Example Code:

```
func resizeImage(originalPath string, width, height int) (string, error) {
    // Open original image
    file, err := os.Open(originalPath)
    if err != nil {
```

```

        return "", err
    }
    defer file.Close()

    img, _, err := image.Decode(file)
    if err != nil {
        return "", err
    }

    // Resize using Lanczos resampling
    resized := resize.Resize(uint(width), uint(height), img, resize.Lanczos3)

    // Save resized image
    outPath := fmt.Sprintf("%s_%dx%d.jpg", originalPath, width, height)
    out, err := os.Create(outPath)
    if err != nil {
        return "", err
    }
    defer out.Close()

    jpeg.Encode(out, resized, &jpeg.Options{Quality: 90})

    return outPath, nil
}

```

9. Elasticsearch (Search)

Purpose: Search users, hashtags, locations

Indices:

```

// Users index
{
    "username": "alice",
    "full_name": "Alice Smith",
    "bio": "Photographer | Travel",
    "follower_count": 1500
}

// Hashtags index
{
    "hashtag": "sunset",
    "post_count": 5000000,
    "trending_score": 0.95
}

// Locations index
{
    "name": "New York, USA",
    "latitude": 40.7128,
    "longitude": -74.0060,
}

```

```
    "post_count": 10000000
}
```

Why Elasticsearch?

- Full-text search (search "alice" matches "alice_smith", "alice123")
- Fast autocomplete (as you type, show suggestions)
- Aggregations (trending hashtags)

4 API & Interface Design

REST API Endpoints

User Management

```
// Sign up
POST /api/users/signup
Request: {
  "username": "alice",
  "email": "alice@example.com",
  "password": "hashed_password",
  "full_name": "Alice Smith"
}
Response: {
  "userId": 123,
  "token": "jwt_token_here"
}

// Login
POST /api/users/login
Request: {
  "username": "alice",
  "password": "hashed_password"
}
Response: {
  "userId": 123,
  "token": "jwt_token_here"
}

// Get profile
GET /api/users/:userId
Response: {
  "userId": 123,
  "username": "alice",
  "full_name": "Alice Smith",
  "bio": "Photographer",
  "profile_pic_url": "https://cdn.instagram.com/...",
  "follower_count": 1500,
  "following_count": 300,
  "post_count": 250,
  "is_following": true, // Whether current user follows this user
}
```

```
    "is_private": false  
}
```

Post Management

```
// Create post (Step 1: Get upload URL)  
POST /api/posts/upload-url  
Request: {  
  "content_type": "image/jpeg",  
  "file_size": 5242880 // 5 MB  
}  
Response: {  
  "upload_url": "https://s3.amazonaws.com/...", // Pre-signed URL  
  "post_id": "p456",  
  "expires_in": 3600 // 1 hour  
}  
  
// Create post (Step 2: Confirm upload)  
POST /api/posts  
Request: {  
  "post_id": "p456",  
  "caption": "Beautiful sunset! #sunset #nature",  
  "location": "Malibu Beach, CA",  
  "tagged_users": ["user789", "user101"]  
}  
Response: {  
  "post_id": "p456",  
  "image_url": "https://cdn.instagram.com/.../p456_640x640.jpg",  
  "created_at": "2026-01-28T18:00:00Z"  
}  
  
// Get post  
GET /api/posts/:postId  
Response: {  
  "post_id": "p456",  
  "user": {  
    "user_id": 123,  
    "username": "alice",  
    "profile_pic_url": "..."  
  },  
  "image_url": "https://cdn.instagram.com/.../p456_640x640.jpg",  
  "caption": "Beautiful sunset! #sunset #nature",  
  "location": "Malibu Beach, CA",  
  "like_count": 1500,  
  "comment_count": 42,  
  "created_at": "2026-01-28T18:00:00Z",  
  "has_liked": false // Whether current user liked this post  
}  
  
// Delete post
```

```
DELETE /api/posts/:postId
Response: {
  "success": true
}
```

Home Feed

```
// Get home feed
GET /api/feed?cursor=&limit=20
Response: {
  "posts": [
    {
      "post_id": "p456",
      "user": {
        "user_id": 123,
        "username": "alice",
        "profile_pic_url": "..."
      },
      "image_url": "...",
      "caption": "...",
      "like_count": 1500,
      "comment_count": 42,
      "created_at": "2026-01-28T18:00:00Z",
      "has_liked": false
    },
    ...
  ],
  "next_cursor": "p420", // For pagination
  "has_more": true
}

// Cursor-based pagination (not offset-based)
GET /api/feed?cursor=p420&limit=20
```

Why cursor-based pagination?

- Offset-based (`LIMIT 20 OFFSET 100`) is slow for large offsets
- Cursor-based uses post ID, constant-time lookup
- Handles new posts being added while user scrolls

Interactions

```
// Like post
POST /api/posts/:postId/like
Response: {
  "success": true,
  "new_like_count": 1501
}

// Unlike post
```

```

DELETE /api/posts/:postId/like
Response: {
  "success": true,
  "new_like_count": 1500
}

// Comment on post
POST /api/posts/:postId/comments
Request: {
  "text": "Amazing photo! 😍"
}
Response: {
  "comment_id": "c789",
  "user": {
    "user_id": 456,
    "username": "bob"
  },
  "text": "Amazing photo! 😍",
  "created_at": "2026-01-28T18:05:00Z"
}

// Get comments
GET /api/posts/:postId/comments?limit=50
Response: {
  "comments": [
    {
      "comment_id": "c789",
      "user": {
        "user_id": 456,
        "username": "bob",
        "profile_pic_url": "..."
      },
      "text": "Amazing photo! 😍",
      "created_at": "2026-01-28T18:05:00Z"
    },
    ...
  ]
}

```

Follow System

```

// Follow user
POST /api/users/:userId/follow
Response: {
  "success": true,
  "is_following": true
}

// Unfollow user
DELETE /api/users/:userId/follow

```

```
Response: {
  "success": true,
  "is_following": false
}

// Get followers
GET /api/users/:userId/followers?cursor=&limit=50
Response: {
  "users": [
    {
      "user_id": 456,
      "username": "bob",
      "full_name": "Bob Jones",
      "profile_pic_url": "...",
      "is_following": false // Whether current user follows Bob
    },
    ...
  ],
  "next_cursor": "u420",
  "has_more": true
}

// Get following
GET /api/users/:userId/following?cursor=&limit=50
```

Search

```
// Search users
GET /api/search/users?q=alice&limit=20
Response: {
  "users": [
    {
      "user_id": 123,
      "username": "alice",
      "full_name": "Alice Smith",
      "profile_pic_url": "...",
      "follower_count": 1500,
      "is_verified": false
    },
    ...
  ]
}

// Search hashtags
GET /api/search/hashtags?q=sunset&limit=20
Response: {
  "hashtags": [
    {
      "hashtag": "sunset",
      "post_count": 5000000
```

```

},
{
  "hashtag": "sunsetphotography",
  "post_count": 1200000
},
...
]
}

```

5 Data Modeling & Storage

PostgreSQL Schema (Detailed)

```

-- Users table
CREATE TABLE users (
  id BIGSERIAL PRIMARY KEY,
  username VARCHAR(30) UNIQUE NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  full_name VARCHAR(100),
  bio TEXT,
  profile_pic_url TEXT,
  website VARCHAR(255),
  is_private BOOLEAN DEFAULT FALSE,
  is_verified BOOLEAN DEFAULT FALSE,
  follower_count INT DEFAULT 0,
  following_count INT DEFAULT 0,
  post_count INT DEFAULT 0,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_users_username ON users(username);
CREATE INDEX idx_users_email ON users(email);

-- Posts table
CREATE TABLE posts (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT NOT NULL REFERENCES users(id),
  image_url TEXT NOT NULL,
  caption TEXT,
  location VARCHAR(255),
  like_count INT DEFAULT 0,
  comment_count INT DEFAULT 0,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_posts_user_created ON posts(user_id, created_at DESC);
CREATE INDEX idx_posts_created ON posts(created_at DESC);

```

```

-- Follows table (social graph)
CREATE TABLE follows (
    follower_id BIGINT NOT NULL REFERENCES users(id),
    followee_id BIGINT NOT NULL REFERENCES users(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (follower_id, followee_id)
);

CREATE INDEX idx_follows_follower ON follows(follower_id);
CREATE INDEX idx_follows_followee ON follows(followee_id);

-- Likes table
CREATE TABLE likes (
    user_id BIGINT NOT NULL REFERENCES users(id),
    post_id BIGINT NOT NULL REFERENCES posts(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (user_id, post_id)
);

CREATE INDEX idx_likes_post ON likes(post_id);
CREATE INDEX idx_likes_user ON likes(user_id);

-- Comments table
CREATE TABLE comments (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL REFERENCES users(id),
    post_id BIGINT NOT NULL REFERENCES posts(id),
    text TEXT NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_comments_post_created ON comments(post_id, created_at DESC);
CREATE INDEX idx_comments_user ON comments(user_id);

-- Hashtags table
CREATE TABLE hashtags (
    id BIGSERIAL PRIMARY KEY,
    tag VARCHAR(100) UNIQUE NOT NULL,
    post_count INT DEFAULT 0
);

CREATE INDEX idx_hashtags_tag ON hashtags(tag);
CREATE INDEX idx_hashtags_post_count ON hashtags(post_count DESC);

-- Post hashtags (many-to-many)
CREATE TABLE post_hashtags (
    post_id BIGINT NOT NULL REFERENCES posts(id),
    hashtag_id BIGINT NOT NULL REFERENCES hashtags(id),
    PRIMARY KEY (post_id, hashtag_id)
);

CREATE INDEX idx_post_hashtags_hashtag ON post_hashtags(hashtag_id);

```

```
-- Notifications table
CREATE TABLE notifications (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL REFERENCES users(id),
    actor_id BIGINT NOT NULL REFERENCES users(id),
    type VARCHAR(20) NOT NULL, -- 'like', 'comment', 'follow', 'mention'
    post_id BIGINT REFERENCES posts(id),
    comment_id BIGINT REFERENCES comments(id),
    is_read BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_notifications_user_created ON notifications(user_id, created_at
DESC) WHERE NOT is_read;
```

Denormalization for Performance

Problem: Counting likes on every request is slow

```
-- Slow query (scans entire likes table)
SELECT COUNT(*) FROM likes WHERE post_id = 123;
```

Solution: Store like_count in posts table

```
-- Fast query (indexed lookup)
SELECT like_count FROM posts WHERE id = 123;

-- Update like_count when user likes
BEGIN;
    INSERT INTO likes (user_id, post_id) VALUES (456, 123);
    UPDATE posts SET like_count = like_count + 1 WHERE id = 123;
COMMIT;
```

Trade-off: Risk of inconsistency (like_count might drift), but acceptable for non-critical data. Periodic reconciliation job fixes drift.

Image URL Structure

Stored in posts table:

```
image_url = 'photos/2026/01/28/user123/p456'
```

Constructed URLs (served via CDN):

```
Thumbnail (150x150):
https://cdn.instagram.com/photos/2026/01/28/user123/p456_150x150.jpg
Feed (640x640):
https://cdn.instagram.com/photos/2026/01/28/user123/p456_640x640.jpg
```

Full (1080×1080):
https://cdn.instagram.com/photos/2026/01/28/user123/p456_1080x1080.jpg

Why store relative path, not full URL?

- Can change CDN provider without updating database
- Can switch between multiple CDNs (A/B testing)
- Smaller storage (20 bytes vs 100 bytes per post)

Storage Cost Analysis

Photo Storage:

100 billion photos total
Average sizes:
- Original: 5 MB
- 1080×1080: 500 KB
- 640×640: 150 KB
- 150×150: 10 KB
Total per photo: ~5.66 MB

Total storage: $100B \times 5.66 \text{ MB} = 566 \text{ PB}$

S3 Standard: $566 \text{ PB} \times \$23/\text{TB/month} = \13M/month
S3 Intelligent-Tiering: Auto-move to cheaper tiers = $\$8\text{M/month}$
S3 + Glacier (>1 year old): $\$4\text{M/month}$

Optimization: Move photos older than 1 year to Glacier (90% cheaper, slower access)

6 Core System Flows

Flow 1: User Uploads Photo

```
sequenceDiagram
    participant U as User (Mobile)
    participant API as API Server
    participant S3
    participant MQ as Message Queue
    participant MP as Media Processor
    participant DB as PostgreSQL
    participant Redis
    participant FeedGen as Feed Generator

    U->>API: POST /api/posts/upload-url<br/>{contentType, fileSize}
    API->>S3: Generate pre-signed URL
    S3-->>API: upload_url (expires 1 hour)
    API-->>U: {upload_url, post_id}

    U->>S3: PUT upload_url<br/>(upload 5 MB image)
    S3-->>U: 200 OK
```

```

U->>API: POST /api/posts<br/>{post_id, caption, location}
API->>DB: INSERT INTO posts
DB-->API: Success

API->>MQ: Queue media processing job<br/>{post_id, s3_key}
API-->U: 201 Created {post_id, image_url}

MQ->>MP: Process image
MP->>S3: Download original
MP->>MP: Resize (150x150, 640x640, 1080x1080)
MP->>S3: Upload resized versions
MP-->>MQ: ACK

API->>FeedGen: Trigger feed fan-out
FeedGen->>DB: Get user's followers (1000 users)
FeedGen->>Redis: Update each follower's feed<br/>ZADD feed:{follower_id}
{timestamp} {post_id}

```

Upload Flow Details:

1. Pre-signed URL: Client uploads directly to S3 (doesn't go through API server)

- Reduces server load (API doesn't handle 5 MB upload)
- Faster upload (client → S3 direct connection)

2. Async Processing: Resizing happens in background

- User doesn't wait for resizing (instant upload confirmation)
- Media Processor scales independently (can process 1,000 images/sec)

3. Feed Fan-Out: Happens asynchronously

- Doesn't block post creation (instant response to user)
- Feed Generator updates followers' feeds in Redis

Flow 2: User Loads Home Feed

```

sequenceDiagram
    participant U as User (Mobile)
    participant API as API Server
    participant Redis
    participant DB as PostgreSQL
    participant CDN

    U->>API: GET /api/feed?limit=20

    API->>Redis: GET feed:{user_id}

    alt Cache Hit
        Redis-->>API: [p456, p455, p454, ...]
        API->>DB: Batch fetch post details<br/>SELECT * FROM posts WHERE id IN (...)

        DB-->>API: Post metadata

```

```

API->>DB: Batch fetch user profiles
DB-->>API: User profiles
API-->>U: JSON response (posts)
else Cache Miss
    API->>DB: Get user's following list
    DB-->>API: [user789, user101, ...]

    API->>DB: Get recent posts from following<br/>(complex query, slow)
    DB-->>API: Posts

    API->>Redis: Cache feed for 5 min<br/>ZADD feed:{user_id} ...
    API-->>U: JSON response (posts)
end

U->>CDN: Load image (p456_640x640.jpg)
CDN-->>U: Image bytes

```

Feed Load Optimization:

1. **Redis Cache:** Pre-computed feed (list of post IDs)
 - Cache hit: ~50ms (Redis lookup + DB batch fetch)
 - Cache miss: ~500ms (generate feed from scratch)
2. **Batch Fetching:** Fetch 20 posts in one query (not 20 separate queries)

```
SELECT * FROM posts WHERE id IN (456, 455, 454, ...);
```

3. **Image Loading:** Images loaded separately from CDN
 - API returns image URLs only (fast JSON response)
 - Client loads images in parallel (feels faster)

Flow 3: Feed Generation (Fan-Out on Write)

Scenario: Alice posts a photo. She has 1,000 followers.

```

graph LR
  A[Alice posts photo] --> FG[Feed Generator]
  FG --> F1[Update Bob's feed<br/>Redis]
  FG --> F2[Update Carol's feed<br/>Redis]
  FG --> F3[Update Dave's feed<br/>Redis]
  FG --> FN[... 1000 followers]

```

Implementation:

```

async function fanOutPost(postId: string, authorId: string) {
  // 1. Get author's followers
  const followers = await db.query(
    'SELECT follower_id FROM follows WHERE followee_id = $1 LIMIT 10000',
    [authorId]
  )
}

```

```

);
// 2. Update each follower's feed in Redis
const pipeline = redis.pipeline();
for (const follower of followers) {
  // Add to feed (sorted set, scored by timestamp)
  pipeline.zadd(`feed:${follower.follower_id}`, Date.now(), postId);

  // Keep only last 1000 posts (trim old posts)
  pipeline.zremrangebyrank(`feed:${follower.follower_id}`, 0, -1001);
}
await pipeline.exec();

console.log(`Fanned out post ${postId} to ${followers.length} followers`);
}

```

Why Fan-Out on Write?

- **Fast reads:** Feed is pre-computed, just read from Redis
- **Slow writes:** Must update 1,000 followers' feeds (but users post infrequently)

Trade-off: Celebrity problem (user with 10M followers → must update 10M feeds)

Flow 4: Celebrity Problem (Fan-Out Hybrid)

Problem: Justin Bieber has 250M followers. Fan-out on write = update 250M feeds (too slow!)

Solution: Hybrid approach

- **Regular users** (<10K followers): Fan-out on write
- **Celebrities** (>10K followers): Fan-out on read

Hybrid Feed Generation:

```

async function generateFeed(userId: string): Promise<Post[]> {
  // 1. Get pre-computed feed from Redis (regular users Alice follows)
  const cachedFeed = await redis.zrevrange(`feed:${userId}`, 0, 100);

  // 2. Get celebrities user follows
  const celebrities = await db.query(`
    SELECT followee_id FROM follows
    WHERE follower_id = $1
    AND followee_id IN (SELECT id FROM users WHERE follower_count > 10000)
  `, [userId]);

  // 3. Fetch recent posts from celebrities (on-the-fly)
  const celebrityPosts = await db.query(`
    SELECT id FROM posts
    WHERE user_id = ANY($1)
    AND created_at > NOW() - INTERVAL '7 days'
    ORDER BY created_at DESC
    LIMIT 50
  `, [celebrities.map(c => c.followee_id)]);

```

```

// 4. Merge cached feed + celebrity posts (sort by timestamp)
const mergedFeed = mergeSorted(cachedFeed, celebrityPosts);

return mergedFeed.slice(0, 20);
}

```

Trade-off: Celebrity posts appear slightly delayed (1-5 seconds) for followers

Flow 5: User Likes Post

```

sequenceDiagram
    participant U as User
    participant API as API Server
    participant DB as PostgreSQL
    participant Redis
    participant Notif as Notification Worker

    U->>API: POST /api/posts/p456/like

    API->>DB: BEGIN TRANSACTION
    API->>DB: INSERT INTO likes (user_id, post_id)
    API->>DB: UPDATE posts SET like_count = like_count + 1
    API->>DB: COMMIT

    API->>Redis: Increment cached like count<br/>HINCRBY post:p456:stats likes 1

    API->>Notif: Queue notification<br/>{type: 'like', post_author_id, actor_id}

    API-->U: 200 OK {new_like_count: 1501}

    Notif->>DB: INSERT INTO notifications
    Notif->>U: Push notification<br/>"Alice liked your photo"

```

Like Flow Optimizations:

1. **Atomic Increment:** `like_count = like_count + 1` in database (prevents race conditions)
2. **Redis Cache:** Update cached like count (next user sees updated count immediately)
3. **Async Notification:** Don't wait for notification to send (faster response)

Idempotency: If user double-clicks like button, don't increment twice

```

INSERT INTO likes (user_id, post_id) VALUES ($1, $2)
ON CONFLICT (user_id, post_id) DO NOTHING; -- Already liked, ignore

```

Flow 6: User Comments on Post

```

sequenceDiagram
    participant U as User

```

```

participant API as API Server
participant DB as PostgreSQL
participant Redis
participant Notif as Notification Worker

U->>API: POST /api/posts/p456/comments<br/>{text: "Amazing!"}

API->>DB: BEGIN TRANSACTION
API->>DB: INSERT INTO comments
API->>DB: UPDATE posts SET comment_count = comment_count + 1
API->>DB: COMMIT

API->>Redis: Invalidate cached comments<br/>DEL post:p456:comments

API->>Notif: Queue notification

API-->U: 201 Created {comment_id, ...}

Notif->>DB: INSERT INTO notifications
Notif->>U: Push notification<br/>"Alice commented on your photo"

```

7 Consistency, Ordering & Concurrency

Eventual Consistency (Feed Updates)

Scenario: Alice posts photo at 6:00 PM. Bob loads feed at 6:00:01 PM (1 second later).

Question: Does Bob see Alice's post?

Answer: Maybe not immediately (eventual consistency)

Why?

- Feed fan-out takes 1-5 seconds (updating 1,000 Redis keys)
- Bob's cached feed might be from 5 minutes ago
- Bob will see post within 5 seconds (acceptable)

Trade-off:

- Strong consistency = Bob always sees post immediately (slow writes, slow reads)
- Eventual consistency = Bob might wait 5 seconds (fast writes, fast reads)

Instagram chooses **eventual consistency** for better performance.

Like Count Consistency (Acceptable Drift)

Scenario: Post has 1,000 likes. 10 users unlike simultaneously.

Potential Issue: Race condition

```

User 1: Read like_count = 1000 → Update to 999
User 2: Read like_count = 1000 → Update to 999 (should be 998!)

```

Solution: Use atomic operations

```
-- Don't do this (race condition)
UPDATE posts SET like_count = 1000 - 1 WHERE id = 123;

-- Do this (atomic)
UPDATE posts SET like_count = like_count - 1 WHERE id = 123;
```

Cached Counts: Redis cache might drift from database

```
// Database: like_count = 1000
// Redis: like_count = 1005 (5 likes cached but not written to DB yet)
```

Mitigation: Periodic reconciliation

```
// Every 5 minutes, sync Redis with database
async function reconcileLikeCounts() {
  const posts = await db.query('SELECT id, like_count FROM posts WHERE updated_at > NOW() - INTERVAL \'5 minutes\'');

  for (const post of posts) {
    await redis.hset(`post:${post.id}:stats`, 'likes', post.like_count);
  }
}
```

Trade-off: Accept temporary drift (showing 1005 instead of 1000) for faster reads.

Follow/Unfollow Race Condition

Scenario: User clicks "Follow" button twice quickly (network lag, didn't see first response)

Without Idempotency:

```
-- First request
INSERT INTO follows (follower_id, followee_id) VALUES (123, 456);
UPDATE users SET follower_count = follower_count + 1 WHERE id = 456;

-- Second request (duplicate)
INSERT INTO follows (follower_id, followee_id) VALUES (123, 456); -- ERROR:
Duplicate key
UPDATE users SET follower_count = follower_count + 1 WHERE id = 456; -- Incorrect
count!
```

With Idempotency:

```
-- First request
INSERT INTO follows (follower_id, followee_id) VALUES (123, 456)
ON CONFLICT (follower_id, followee_id) DO NOTHING;

-- Only update follower_count if INSERT succeeded
UPDATE users SET follower_count = follower_count + 1
```

```

WHERE id = 456 AND NOT EXISTS (
    SELECT 1 FROM follows WHERE follower_id = 123 AND followee_id = 456
);

-- Second request (duplicate)
INSERT INTO follows (follower_id, followee_id) VALUES (123, 456)
ON CONFLICT DO NOTHING; -- No-op, follower_count not incremented

```

8 Caching Strategy

Multi-Layer Caching

```

Layer 1: Client Cache (Mobile App)
    ↓
Layer 2: CDN (CloudFront)
    ↓
Layer 3: Application Cache (Redis)
    ↓
Layer 4: Database (PostgreSQL)

```

Layer 1: Client-Side Cache

What to Cache:

- Feed posts (last 100 seen)
- User profiles (frequently viewed)
- Images (LRU cache, max 50 MB)

Implementation (React Native):

```

import AsyncStorage from '@react-native-async-storage/async-storage';

class FeedCache {
    async cacheFeed(posts: Post[]) {
        await AsyncStorage.setItem('feed_cache', JSON.stringify(posts));
        await AsyncStorage.setItem('feed_cache_timestamp', Date.now().toString());
    }

    async getFeed(): Promise<Post[] | null> {
        const cached = await AsyncStorage.getItem('feed_cache');
        const timestamp = await AsyncStorage.getItem('feed_cache_timestamp');

        // Cache valid for 5 minutes
        if (cached && timestamp && Date.now() - parseInt(timestamp) < 300000) {
            return JSON.parse(cached);
        }

        return null;
    }
}

```

Benefit: Instant feed load on app open (from local cache, then refresh in background)

Layer 2: CDN Cache (CloudFront)

What to Cache: All images and videos

Cache Headers:

```
Cache-Control: public, max-age=31536000, immutable
```

Why `immutable` ?

- Images never change after upload (no need to revalidate)
- Browser/CDN can cache forever

CDN Hit Rate: 95% (most images served from CDN, not S3 origin)

Cost Savings:

```
Without CDN: 100 PB/day from S3 = $9,000/day bandwidth  
With CDN (95% hit): 5 PB/day from S3 = $450/day bandwidth  
Savings: $8,550/day = $3.1M/year
```

Layer 3: Application Cache (Redis)

Feed Cache

```
// Cache structure: Sorted set (score = timestamp)  
feed:{userId} → ZSET {  
  1738100000: "p456",  
  1738099000: "p455",  
  1738098000: "p454",  
  ...  
}  
  
// Get feed (last 20 posts)  
const feed = await redis.zrevrange('feed:${userId}', 0, 19);  
  
// TTL: No expiration (updated on new posts)
```

Post Metadata Cache

```
// Cache post like/comment counts (frequently accessed)  
post:{postId}:stats → HASH {  
  likes: 1500,  
  comments: 42,  
  shares: 8  
}
```

```
// TTL: 5 minutes (stale counts acceptable)
await redis.expire(`post:${postId}:stats`, 300);
```

User Profile Cache

```
// Cache user profile (bio, follower count, etc.)
user:{userId}:profile → JSON string

// TTL: 1 hour
await redis.setex(`user:${userId}:profile`, 3600, JSON.stringify(profile));

// Invalidation: On profile update
async function updateProfile(userId: string, updates: Partial<User>) {
  await db.query('UPDATE users SET ... WHERE id = $1', [userId]);
  await redis.del(`user:${userId}:profile`); // Invalidate cache
}
```

Cache Invalidation Strategies

Write-Through (For Likes)

```
async function likePost(userId: string, postId: string) {
  // 1. Write to database
  await db.query('INSERT INTO likes (user_id, post_id) VALUES ($1, $2)', [userId, postId]);
  await db.query('UPDATE posts SET like_count = like_count + 1 WHERE id = $1', [postId]);

  // 2. Update cache immediately
  await redis.hincrby(`post:${postId}:stats`, 'likes', 1);
}
```

Benefit: Cache always up-to-date (no stale data)

Cache-Aside (For Feed)

```
async function getFeed(userId: string): Promise<Post[]> {
  // 1. Check cache
  const cached = await redis.zrevrange(`feed:${userId}`, 0, 19);
  if (cached.length > 0) {
    return fetchPostDetails(cached);
  }

  // 2. Cache miss → generate feed
  const feed = await generateFeed(userId);

  // 3. Populate cache
  const pipeline = redis.pipeline();
```

```

for (const post of feed) {
  pipeline.zadd(`feed:${userId}`, post.created_at, post.id);
}

await pipeline.exec();

return feed;
}

```

9 Scaling Strategy

Horizontal Scaling (Stateless Services)

All services are **stateless** (no local state, can scale horizontally):

| Component | Current Scale | Scaling Strategy | Bottleneck |
|-------------------------|-------------------|----------------------------------|--------------------------|
| API Servers | 1,000 instances | Add more instances, load balance | Database read throughput |
| Feed Generators | 200 instances | Add more instances, queue-based | Redis write throughput |
| Media Processors | 500 instances | Add more instances, queue-based | CPU (image processing) |
| PostgreSQL | 100 read replicas | Add more replicas | Master write throughput |
| Redis | 50 shards | Add more shards | Memory |

Database Scaling (Read Replicas)

Problem: Feed queries overload master database (100K queries/sec)

Solution: Read replicas (route reads to replicas, writes to master)

```

// Write to master
await masterDB.query('INSERT INTO posts (user_id, image_url, caption) VALUES ($1,
$2, $3)',
[userId, imageUrl, caption]);

// Read from replica (with slight lag, acceptable)
const posts = await replicaDB.query('SELECT * FROM posts WHERE user_id = $1 ORDER BY
created_at DESC LIMIT 20',
[userId]);

```

Replication Lag: 50-200ms (acceptable for non-critical reads)

When to read from master?

- After creating post (user expects to see their own post immediately)
- After following user (user expects to see their posts in feed)

Sharding (Future, if Needed)

Current: Single PostgreSQL database (with read replicas)

Problem (future): 2B users, 100B posts → single DB can't handle

Solution: Shard by user_id

```
Shard 0: Users 0-199,999,999      (200M users)
Shard 1: Users 200M-399,999,999   (200M users)
Shard 2: Users 400M-599,999,999   (200M users)
...
...
```

Routing:

```
function getDBShard(userId: number): Database {
  const shardId = Math.floor(userId / 200_000_000);
  return dbShards[shardId];
}

// Query user's posts (single shard)
const shard = getDBShard(userId);
const posts = await shard.query('SELECT * FROM posts WHERE user_id = $1', [userId]);
```

Cross-Shard Queries (feed generation):

```
Alice (shard 0) follows Bob (shard 1) and Carol (shard 2)
To generate Alice's feed, must query shard 1 and shard 2 (expensive!)
```

Mitigation: Pre-compute feeds (fan-out on write), avoid cross-shard queries at read time.

CDN Scaling (Auto-Scaling by AWS)

CDN scales automatically (managed by AWS CloudFront):

- 410+ edge locations worldwide
- Auto-scales based on traffic
- No configuration needed

Cost: Pay-per-use (\$0.085/GB for first 10 TB/month, cheaper for higher volumes)

10 Fault Tolerance & Reliability

Failure Scenarios

Failure 1: API Server Crashes

Impact: Users connected to that server get errors

Detection: Load balancer health check fails

Recovery:

1. Load balancer removes unhealthy server from pool

2. New requests route to healthy servers
3. Auto-scaling group launches new instance (2 minutes)

Data Loss: None (servers are stateless)

RTO: 2 minutes (time to launch new instance)

Failure 2: PostgreSQL Master Fails

Impact: Can't write new posts, likes, comments

Detection: Health check fails, replica stops receiving replication stream

Mitigation: Automatic failover

1. Patroni detects master failure (30-second timeout)
2. Promotes read replica to master (20 seconds)
3. DNS updated to point to new master (10 seconds)
4. Total failover time: 60 seconds

Data Loss: None (synchronous replication ensures replica is up-to-date)

RTO: 1 minute

Failure 3: Redis Cluster Node Fails

Impact: Cached feeds on that node lost, must regenerate

Mitigation: Redis Sentinel auto-failover

1. Sentinel detects node failure (30 seconds)
2. Promotes replica to master (10 seconds)
3. Total failover time: 40 seconds

Data Loss: Acceptable (feeds are ephemeral, can regenerate from database)

Impact: Temporary slowdown (feed generation from DB instead of cache)

Failure 4: CDN Region Outage

Impact: Users in that region see slow image loading (fallback to origin)

Mitigation: CloudFront automatically routes to nearest healthy region

Recovery: Automatic (no manual intervention)

Data Durability (S3)

Photos/Videos: 99.99999999% durability (11 nines)

What this means: If you store 10 million photos, expect to lose 1 photo every 10,000 years

Implementation: S3 automatically replicates across 3+ availability zones

Graceful Degradation

Feature Flags: Disable non-critical features during incident

```
const features = {
  feed: await redis.get('feature:feed') !== '0', // Critical, always on
  explore: await redis.get('feature:explore') !== '0', // Can disable
  stories: await redis.get('feature:stories') !== '0', // Can disable
  search: await redis.get('feature:search') !== '0', // Can disable
};

if (!features.explore) {
  return res.status(503).json({ error: 'Explore page temporarily unavailable' });
}
```

During PostgreSQL overload:

- Disable explore page (complex queries)
- Disable search (Elasticsearch queries)
- Keep feed working (Redis cache)

1 | 1 Observability & Operations

Key Metrics

Application Metrics

```
// Feed load latency
histogram('feed.load.latency_ms', latencyMs);

// Feed cache hit rate
counter('feed.cache.hits', 1);
counter('feed.cache.misses', 1);

// Post upload rate
counter('posts.uploaded', 1, { user_type: 'regular' | 'verified' });

// API request rate
counter('api.requests', 1, { endpoint: '/api/feed', status: 200 });
```

Infrastructure Metrics

```
// Database connections
gauge('postgres.connections', connectionCount);

// Redis memory usage
gauge('redis.memory.used_mb', usedMemoryMB);

// API server CPU
gauge('api.cpu.percent', cpuPercent);
```

```
// CDN bandwidth  
counter('cdn.bytes_served', bytesServed);
```

Alerts

```
alerts:  
  - name: HighFeedLatency  
    condition: feed.load.latency_ms.p99 > 3s  
    severity: warning  
    action: Check database/Redis performance  
  
  - name: LowCacheHitRate  
    condition: feed.cache.hits / (hits + misses) < 0.8  
    severity: warning  
    action: Investigate cache invalidation, increase cache size  
  
  - name: HighErrorRate  
    condition: api.errors / api.requests > 0.05  
    severity: critical  
    action: Page on-call engineer  
  
  - name: DatabaseDown  
    condition: postgres.health == 0  
    severity: critical  
    action: Trigger automatic failover
```

Debugging Common Issues

Issue 1: "Feed is loading slowly"

Debug Steps:

1. Check Redis cache hit rate: `redis-cli INFO stats | grep cache_hit_rate`
2. Check PostgreSQL slow queries: `SELECT * FROM pg_stat_statements ORDER BY mean_time DESC LIMIT 10;`
3. Check API server CPU: `top` or Datadog dashboard
4. Check network latency: `ping api.instagram.com`

Common Causes:

- Redis cache miss (feed not pre-computed)
- Slow database query (missing index)
- API server overloaded (scale up)

Issue 2: "Images not loading"

Debug Steps:

1. Check CDN status: `curl -I https://cdn.instagram.com/path/to/image.jpg`
2. Check S3 origin: `aws s3 ls s3://instagram-media/path/to/image.jpg`
3. Check CDN cache hit rate: CloudFront dashboard

4. Check image URL format: Ensure correct size variant (640x640)

Common Causes:

- Image not processed yet (media processor lag)
- CDN cache miss (first load after upload)
- S3 bucket misconfigured (permissions)

1 2 Security & Abuse Prevention

Authentication

JWT-based authentication:

```
// Login
POST /api/users/login
Response: {
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expires_in": 604800 // 7 days
}

// Verify token on every request
const decoded = jwt.verify(token, SECRET_KEY);
req.user = { id: decoded.userId, username: decoded.username };
```

Rate Limiting

Per-User Limits:

```
// Limit: 100 posts per day
async function checkPostRateLimit(userId: string): Promise<boolean> {
  const key = `ratelimit:${userId}:posts:${currentDay()}`;
  const count = await redis.incr(key);

  if (count === 1) {
    await redis.expire(key, 86400); // 24 hours
  }

  return count <= 100;
}

// Limit: 1000 likes per hour
async function checkLikeRateLimit(userId: string): Promise<boolean> {
  const key = `ratelimit:${userId}:likes:${currentHour()}`;
  const count = await redis.incr(key);

  if (count === 1) await redis.expire(key, 3600);

  return count <= 1000;
}
```

Content Moderation

Automated Moderation (ML models):

```
async function moderateImage(imageUrl: string): Promise<boolean> {
  // Call ML API (AWS Rekognition, Google Vision, etc.)
  const result = await mlAPI.detect({
    image: imageUrl,
    categories: ['explicit', 'violence', 'hate_symbols']
  });

  if (result.explicit > 0.9 || result.violence > 0.8) {
    // Reject image
    return false;
  }

  return true;
}
```

User Reporting:

```
CREATE TABLE reports (
  id BIGSERIAL PRIMARY KEY,
  reporter_id BIGINT NOT NULL REFERENCES users(id),
  reported_post_id BIGINT REFERENCES posts(id),
  reported_user_id BIGINT REFERENCES users(id),
  reason VARCHAR(50) NOT NULL, -- 'spam', 'inappropriate', 'harassment'
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

Privacy Controls

Private Accounts:

```
async function canViewProfile(viewerId: string, profileId: string): Promise<boolean> {
  const profile = await db.query('SELECT is_private FROM users WHERE id = $1',
  [profileId]);

  if (!profile.is_private) {
    return true; // Public account
  }

  // Check if viewer follows this user
  const isFollowing = await db.query(
    'SELECT 1 FROM follows WHERE follower_id = $1 AND followee_id = $2',
    [viewerId, profileId]
  );
}
```

```
    return isFollowing !== null;
}
```

Blocked Users:

```
CREATE TABLE blocks (
  blocker_id BIGINT NOT NULL REFERENCES users(id),
  blocked_id BIGINT NOT NULL REFERENCES users(id),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  PRIMARY KEY (blocker_id, blocked_id)
);

-- Before showing feed, filter out blocked users
SELECT p.* FROM posts p
WHERE p.user_id NOT IN (
  SELECT blocked_id FROM blocks WHERE blocker_id = $1
);
```

1 | 3 Selective Low-Level Design

Feed Ranking Algorithm (Simplified)

Goal: Show most relevant posts first (not just reverse chronological)

Factors:

1. **Recency:** Newer posts ranked higher
2. **Engagement:** Posts with more likes/comments ranked higher
3. **Relationship:** Posts from close friends ranked higher

Simple Scoring Formula:

```
function scorePost(post: Post, viewerId: string): number {
  // Recency score (decay over time)
  const ageInHours = (Date.now() - post.created_at) / 3600000;
  const recencyScore = Math.exp(-ageInHours / 24); // Exponential decay

  // Engagement score (normalized)
  const engagementScore = Math.log(1 + post.like_count + post.comment_count * 5);

  // Relationship score (do you interact with this author often?)
  const relationshipScore = getRelationshipScore(viewerId, post.user_id);

  // Weighted sum
  return (
    0.5 * recencyScore +
    0.3 * engagementScore +
    0.2 * relationshipScore
  );
}
```

```

async function getRelationshipScore(viewerId: string, authorId: string): Promise<number> {
  // Count interactions in last 30 days
  const interactions = await redis.get(`interactions:${viewerId}:${authorId}`);
  return Math.min(interactions / 10, 1.0); // Normalize to [0, 1]
}

```

Real Instagram: Uses complex ML model (deep learning), but concept is similar.

Image Upload & Processing Pipeline

```

class ImageUploadPipeline {
  async uploadImage(userId: string, imageFile: File): Promise<string> {
    // 1. Generate post ID
    const postId = generateId();

    // 2. Generate pre-signed S3 URL
    const uploadUrl = await s3.getSignedUrl('putObject', {
      Bucket: 'instagram-media',
      Key: `photos/original/${currentDate()}/${userId}/${postId}_original.jpg`,
      Expires: 3600,
      ContentType: 'image/jpeg'
    });

    // 3. Return URL to client (client uploads directly to S3)
    return { postId, uploadUrl };
  }

  async processImage(postId: string, s3Key: string) {
    // 1. Download original from S3
    const original = await s3.getObject({ Bucket: 'instagram-media', Key: s3Key });

    // 2. Resize to multiple sizes
    const sizes = [
      { width: 150, height: 150, name: 'thumbnail' },
      { width: 640, height: 640, name: 'feed' },
      { width: 1080, height: 1080, name: 'full' }
    ];

    for (const size of sizes) {
      const resized = await resizeImage(original, size.width, size.height);

      const resizedKey = s3Key.replace('_original.jpg', `_${size.name}.jpg`);
      await s3.putObject({
        Bucket: 'instagram-media',
        Key: resizedKey,
        Body: resized,
        ContentType: 'image/jpeg'
      });
    }
  }
}

```

```

    // 3. Invalidate CDN cache
    await cloudfront.createInvalidation({
      Paths: [`/photos/*/\{postId\}_*.jpg`]
    });
  }
}

```

Optimistic UI Updates

Goal: Make app feel fast (don't wait for server response)

Like Button Example:

```

class LikeButton {
  async handleLike(postId: string) {
    // 1. Optimistic update (show liked immediately)
    this.setState({ liked: true, likeCount: this.state.likeCount + 1 });

    try {
      // 2. Send to server
      const result = await api.post(`/posts/${postId}/like`);

      // 3. Update with server response (in case of drift)
      this.setState({ likeCount: result.new_like_count });
    } catch (err) {
      // 4. Rollback on error
      this.setState({ liked: false, likeCount: this.state.likeCount - 1 });
      alert('Failed to like post');
    }
  }
}

```

Trade-off: User might see incorrect count briefly (e.g., shows 1501, server returns 1505)

1 | 4 Trade-offs & Alternatives

Chosen: Fan-Out on Write (Hybrid)

Why Chosen:

- Fast reads (feed pre-computed in Redis)
- Acceptable writes (most users have <1,000 followers)

Alternative: Fan-Out on Read

| Approach | Read Latency | Write Latency | Celebrity Problem |
|------------------|-----------------------|------------------------|-------------------------|
| Fan-Out on Write | 50ms (Redis) | 1-5s (update 1K feeds) | Hard (update 10M feeds) |
| Fan-Out on Read | 500ms (query + merge) | 10ms (insert post) | Easy (query on-the-fly) |
| Hybrid | 100ms (Redis + query) | 50ms (small fan-out) | Manageable |

Why Hybrid?

- Regular users (<10K followers): Fan-out on write (fast reads)
 - Celebrities (>10K followers): Fan-out on read (fast writes)
-

Chosen: CDN for Images

Why Chosen:

- 95% cache hit rate (most images served from edge)
- Low latency (50ms from nearest edge)
- Cost-effective (\$3M/year savings vs direct S3)

Alternative: Serve images directly from S3

| Approach | Latency (US) | Latency (India) | Cost |
|------------------|--------------|-----------------|-----------|
| Direct S3 | 50ms | 300ms | \$9M/year |
| CDN | 50ms | 50ms | \$6M/year |

Trade-off: CDN adds complexity (cache invalidation), but worth it for global low latency.

Chosen: Eventual Consistency for Feeds

Why Chosen:

- Fast reads and writes
- Users tolerate 1-5 second lag (not critical)

Alternative: Strong consistency (feed always up-to-date)

Trade-off: Strong consistency requires expensive distributed transactions, adds 500ms+ latency.

Chosen: PostgreSQL over NoSQL

Why Chosen:

- Need relational joins (posts + users + likes)
- Need ACID transactions (like_count must be consistent)
- Well-understood, mature ecosystem

Alternative: DynamoDB, Cassandra

| Database | Pros | Cons |
|-------------------|------------------------|-------------------------------|
| PostgreSQL | ACID, joins, mature | Limited write throughput |
| DynamoDB | Auto-scaling, managed | No joins, expensive at scale |
| Cassandra | Linear scaling, writes | No ACID, eventual consistency |

When to consider NoSQL: If writes become bottleneck (10M posts/sec), shard PostgreSQL or migrate to Cassandra.

1 | 5 Interviewer Discussion Notes

Common Follow-Up Questions (New Grad Level)

Q1: "How do you handle a celebrity with 100M followers posting?"

Answer:

- Use **hybrid fan-out** (fan-out on read for celebrities)
- Don't update 100M feeds immediately (too expensive)
- When followers load feed, query celebrity posts on-the-fly and merge

Code (shown earlier in Flow 4)

Q2: "What if two users like the same post at the exact same time?"

Answer:

- Use **atomic increment** in database: `UPDATE posts SET like_count = like_count + 1`
- PostgreSQL handles concurrency internally (row-level locking)
- Both likes counted correctly

Q3: "How do you prevent users from uploading inappropriate images?"

Answer:

- **Automated moderation:** ML model (AWS Rekognition, Google Vision) scans images
- Reject images with high confidence of inappropriate content
- **User reporting:** Users can report posts, human moderators review

Q4: "What happens if PostgreSQL master fails?"

Answer:

- **Automatic failover:** Patroni promotes read replica to master (1 minute)
- **No data loss:** Synchronous replication ensures replica is up-to-date
- **Minimal downtime:** Applications reconnect automatically

Q5: "How do you generate a personalized feed?"

Answer:

- **Simple ranking:** Score posts by recency + engagement + relationship
- **ML model** (advanced): Deep learning model predicts which posts user will like
- **A/B testing:** Test different algorithms, measure engagement

Learning Resources for New Grads

System Design:

- Book: "Designing Data-Intensive Applications" (Chapter 1-3: Fundamentals)
- Course: "System Design Primer" (GitHub repo, excellent resource)

Instagram-Specific:

- Blog: "Instagram Engineering Blog" (engineering.instagram.com)

- Video: "How Instagram Scaled" (YouTube, InfoQ talks)

Practice:

- Build a mini Instagram (photo upload + feed + likes)
 - Use: Node.js + PostgreSQL + Redis + S3
 - Focus on feed generation algorithm
-

What Makes This Design "Junior-Level Appropriate"?

1. **Focus on fundamentals:** CDN, caching, read-heavy optimization (not advanced ML)
 2. **Clear trade-offs:** Fan-out on write vs read, eventual consistency
 3. **Practical scale:** 2B users (realistic), not Google-scale (too abstract)
 4. **Simple algorithms:** Exponential decay, log scaling (not deep learning)
 5. **Standard tech stack:** PostgreSQL, Redis, S3 (not Cassandra, Kafka, custom solutions)
-

Summary (For Interview Recap)

System: Instagram (photo sharing, 2B MAU, 100M posts/day)

Key Decisions:

1. **CDN for images** (95% hit rate, 50ms latency globally)
2. **Hybrid fan-out** (write for regular users, read for celebrities)
3. **Redis for feed caching** (50ms feed load)
4. **PostgreSQL + read replicas** (100 replicas, 50ms lag)
5. **Eventual consistency** (1-5 second lag acceptable)

Scale Numbers:

- 350K feed requests/sec (peak)
- 100 PB/day CDN bandwidth
- 500 PB total storage (photos + videos)
- 100ms P50 latency (feed load)

Hardest Problems:

- Feed generation at scale (2B personalized feeds)
- Celebrity problem (100M followers fan-out)
- Storage costs (500 PB = \$11M/month)
- Global low latency (CDN + regional databases)

Trade-offs Made:

- Hybrid fan-out (speed) over pure fan-out on write (simplicity)
 - Eventual consistency (speed) over strong consistency (correctness)
 - CDN (complexity + cost) over direct S3 (simplicity)
 - Denormalized counts (speed) over always-accurate counts (consistency)
-

End of Document (4,987 lines)

This design represents a **read-heavy social network** optimized for browsing, with practical considerations for storage costs, global latency, and handling viral content. Perfect for new grad interviews focusing on feed generation, caching, and scalability fundamentals!

Prepared for new grad / junior engineer interviews (0-2 YOE).