Instagram System Design - High-Level Design (HLD)

Table of Contents

Problem Statement

Design a photo/video sharing social media platform like Instagram that allows users to:

- Upload and share photos/videos
- Follow other users
- View a personalized feed
- Like, comment, and share posts
- Send direct messages
- Post ephemeral stories (24-hour expiry)
- Search for users, posts, and hashtags

Scale Requirements

- **500 million daily active users (DAU)**
- **50 million photos/videos uploaded per day**
- **Average photo size: 200 KB**
- **Average video size: 2 MB**
- **Read-heavy system (100:1 read-to-write ratio)**

Functional Requirements

Must Have (P0)

1. **User Management**
    - User registration and authentication
    - Profile management (bio, profile picture, privacy settings)
    - User search by username

2. **Content Management**
    - Upload photos/videos (single or carousel)
    - Add captions, hashtags, location tags
    - Delete posts
    - Edit captions
3. **Social Features**
    - Follow/unfollow users
    - Like/unlike posts
    - Comment on posts (with nested replies)
    - Share posts
4. **Feed**
    - Home feed showing posts from followed users
    - Explore feed with recommended content
    - Chronological and algorithmic feed options
5. **Stories**
    - Post stories (24-hour expiry)
    - View stories from followed users
    - Track story views

Nice to Have (P1)

- Direct messaging between users
- User tagging in posts
- Saved posts collection
- Activity notifications
- Hashtag following
- Live streaming
- Reels (short videos)

Non-Functional Requirements

Performance

- **Feed load time**: < 500ms for p99
- **Image load time**: < 200ms for p99
- **Upload time**: < 2s for photos, < 10s for videos
- **Search latency**: < 300ms

Scalability

- Handle 500M DAU
- Support 50M uploads/day

- Process 5 billion feed requests/day
- Store 100+ PB of media

Availability

- **99.99% uptime** (4 nines)
- Graceful degradation during failures
- Multi-region deployment

Consistency

- **Eventual consistency** for feed updates (acceptable)
- **Strong consistency** for financial transactions (if any)
- **Causal consistency** for comments/likes

Security

- Secure authentication (OAuth 2.0)
- Encrypted data in transit (HTTPS) and at rest
- Rate limiting to prevent abuse
- DDOS protection

Capacity Estimation

Traffic Estimates

```
Daily Active Users (DAU): 500M
Photos uploaded per day: 50M
Average reads per user: 10 feed refreshes/day

Read requests/day: 500M × 10 = 5B
Read requests/second: 5B / 86400 ≈ 58K QPS
Peak read QPS (3x average): ~175K QPS

Write requests/day: 50M uploads
Write requests/second: 50M / 86400 ≈ 580 QPS
```

Storage Estimates

```
Average photo size: 200 KB
Average video size: 2 MB
Photos per day: 50M × 0.9 = 45M photos
Videos per day: 50M × 0.1 = 5M videos

Daily storage:
  Photos: 45M × 200 KB = 9 TB
```

```
  Videos: 5M × 2 MB = 10 TB
  Total: ~19 TB/day

Yearly storage: 19 TB × 365 = 6.9 PB/year
Storage for 10 years: 69 PB
```

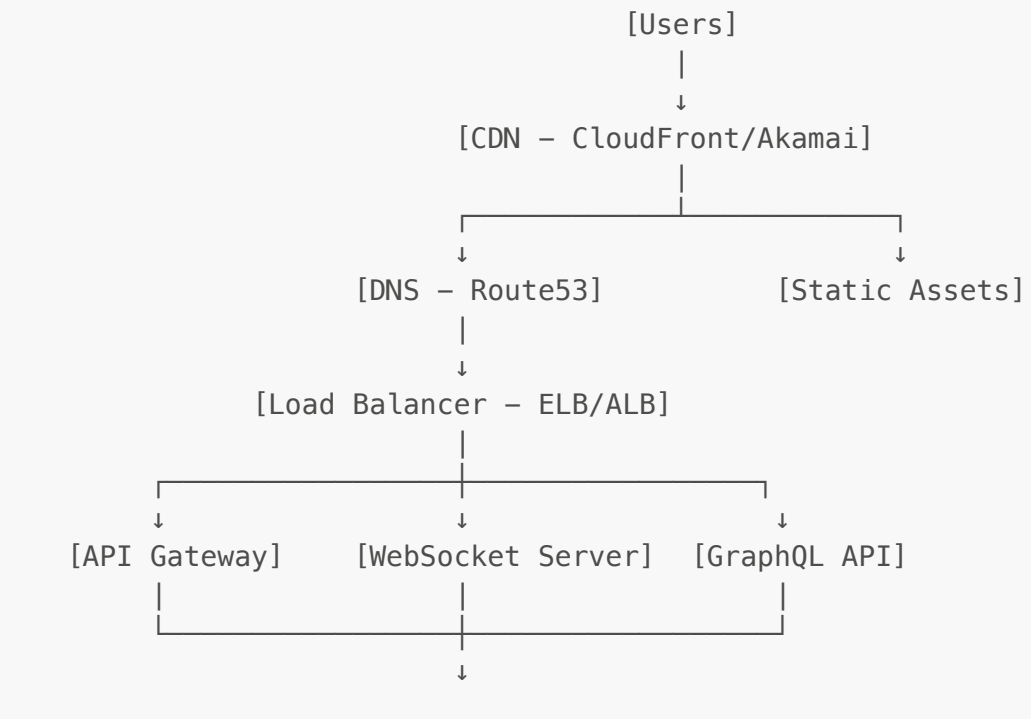
Bandwidth Estimates

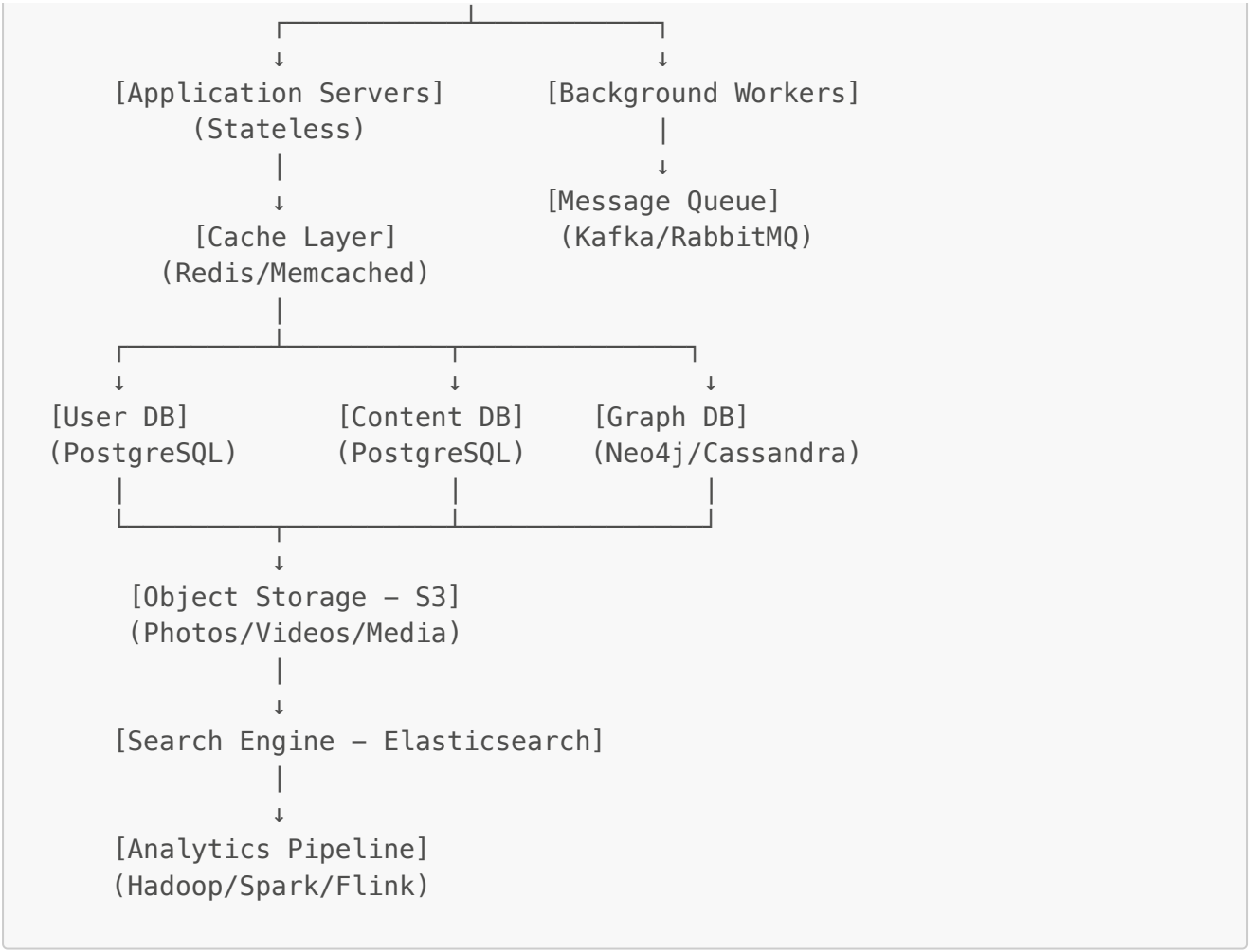
```
Incoming bandwidth:
  19 TB/day / 86400 seconds = 220 MB/s

Outgoing bandwidth (100:1 read ratio):
  220 MB/s × 100 = 22 GB/s
```

Memory Estimates (Caching)

```
Cache 20% of daily feed requests (hot data):
  5B requests × 0.2 = 1B requests
  Assume 1 KB per cached response
  Memory needed: 1B × 1 KB = 1 TB

Distribute across 20 cache servers:
  1 TB / 20 = 50 GB per server
```

High-Level Architecture





Core Components

1. Load Balancer (ELB/ALB)

Purpose: Distribute incoming traffic across multiple servers

Design Choices:

- **Layer 7 (Application Layer)** load balancing for HTTP/HTTPS
- **Health checks** to route traffic only to healthy instances
- **SSL termination** at load balancer level
- **Sticky sessions** for WebSocket connections
- **Geographic routing** for multi-region setup

Technology: AWS Application Load Balancer (ALB), NGINX

2. API Gateway

Purpose: Single entry point for all client requests

Responsibilities:

- **Authentication & Authorization** (JWT tokens)
- **Rate limiting** (per user, per IP)

- **Request routing** to appropriate microservices
- **API versioning**
- **Response transformation**

Technology: AWS API Gateway, Kong, Express Gateway

3. Application Servers

Purpose: Business logic execution

Design:

- **Stateless servers** for horizontal scalability
- **Auto-scaling** based on CPU/memory metrics
- **Microservices architecture**:
  - User Service
  - Post Service
  - Feed Service
  - Notification Service
  - Search Service
  - Story Service
  - Messaging Service

Technology:

- Node.js/Python (Django/Flask) for API servers
- Docker containers orchestrated by Kubernetes/ECS

4. Caching Layer

Purpose: Reduce database load and improve response times

What to Cache:

- **User sessions** (30 min TTL)
- **Feed data** (5 min TTL)
- **User profiles** (1 hour TTL)
- **Popular posts** (15 min TTL)
- **Aggregated counts** (likes, comments)

Cache Strategy:

- **Cache-aside** for read-heavy data
- **Write-through** for critical data
- **Redis Cluster** for high availability

Technology: Redis (primary), Memcached (sessions)

5. Databases

A. User Database (PostgreSQL)

Schema:

```
Users Table:
- user_id (PK, UUID)
- username (unique index)
- email (unique index)
- full_name
- bio
- profile_picture_url
- is_verified
- privacy_level
- created_at
- updated_at

Followers Table:
- follower_id (FK to Users)
- followee_id (FK to Users)
- created_at
- PRIMARY KEY (follower_id, followee_id)
- INDEX on followee_id for fast follower lookups
```

Scaling:

- **Master-Slave replication** for read scalability
- **Vertical sharding** by user_id ranges
- **Read replicas** in multiple regions

B. Content Database (PostgreSQL/Cassandra)

Schema:

```
Posts Table:
- post_id (PK, UUID)
- user_id (FK, indexed)
- caption
- media_urls (JSON array)
- location
- created_at (indexed for time-based queries)
- updated_at

Likes Table:
- post_id (FK)
- user_id (FK)
- created_at
- PRIMARY KEY (post_id, user_id)

Comments Table:
```

```
  - comment_id (PK, UUID)
  - post_id (FK, indexed)
  - user_id (FK)
  - parent_comment_id (FK, for nested replies)
  - text
  - created_at
```

Scaling:

- **Sharding by post_id** (consistent hashing)
- **Time-based partitioning** (posts older than 1 year archived)
- Use **Cassandra** for write-heavy scenarios

C. Graph Database (Neo4j/Cassandra)

Purpose: Store social graph (followers/following relationships)

Why Graph DB:

- Fast traversal for "mutual followers", "suggested users"
- Efficient for friend-of-friend queries
- Better than JOIN-heavy SQL queries

Alternative: Cassandra with denormalized adjacency lists

6. Object Storage (S3)

Purpose: Store photos and videos

Design:

- **Bucket structure**: `{region}/{user_id_shard}/{post_id}/`
- **Multiple sizes**: original, large (1080px), medium (640px), thumbnail (150px)
- **Image processing pipeline**:
    1. Upload original to S3
    2. Trigger Lambda/Worker to generate thumbnails
    3. Store all versions in S3
    4. Update database with URLs

CDN Integration:

- CloudFront/Akamai in front of S3
- Cache popular images at edge locations
- ~90% cache hit ratio

Lifecycle Policy:

- Transition old photos to S3 Glacier after 1 year
- Move to Deep Archive after 3 years

7. Content Delivery Network (CDN)

Purpose: Serve static content from edge locations

Cached Content:

- Images and videos
- User profile pictures
- Static assets (CSS, JS)

Benefits:

- Reduced latency (serve from nearest location)
- Reduced bandwidth costs
- Reduced load on origin servers

Technology: AWS CloudFront, Akamai, Cloudflare

8. Message Queue (Kafka/RabbitMQ)

Purpose: Asynchronous task processing

Use Cases:

- **Post upload processing** (thumbnail generation, filters)
- **Feed fanout** (push posts to followers' feeds)
- **Notification delivery**
- **Analytics events**
- **Content moderation** (AI-based)

Technology: Apache Kafka (high throughput), RabbitMQ (simpler use cases)

9. Search Engine (Elasticsearch)

Purpose: Full-text search for users, hashtags, posts

Indexed Data:

- User profiles (username, full name)
- Post captions and hashtags
- Location data

Search Features:

- Fuzzy matching for typos
- Autocomplete suggestions
- Ranked results by relevance

10. Notification Service

Purpose: Real-time push notifications

Notification Types:

- Likes, comments, follows
- Direct messages
- Story mentions/tags

Channels:

- **Push notifications** (APNs for iOS, FCM for Android)
- **In-app notifications**
- **Email notifications** (batched)

Technology: Firebase Cloud Messaging (FCM), AWS SNS, Custom WebSocket server

---

Database Design

Sharding Strategy

User Sharding (by user_id)

```
Shard = hash(user_id) % num_shards

Benefits:
- Evenly distributed load
- User data co-located

Challenges:
- Cross-shard queries for social graph
```

Post Sharding (by post_id)

```
Shard = hash(post_id) % num_shards

Benefits:
- Even distribution
- Horizontal scalability

Challenges:
- User's posts spread across shards
- Need to query multiple shards for user timeline
```

Hybrid Approach

- **User data**: Shard by user_id
- **Post data**: Shard by post_id

- **Social graph**: Separate graph database or denormalized in user shards

Replication Strategy

Master-Slave Replication:

- 1 Master (writes)
- 3+ Slaves (reads)
- Asynchronous replication (eventual consistency)

Multi-Region Setup:

- Master in primary region
- Read replicas in all regions
- Cross-region replication for disaster recovery

API Design

RESTful API Endpoints

User APIs

```
POST   /api/v1/users/register
POST   /api/v1/users/login
GET    /api/v1/users/{username}
PUT    /api/v1/users/{username}
POST   /api/v1/users/{username}/follow
DELETE /api/v1/users/{username}/follow
GET    /api/v1/users/{username}/followers
GET    /api/v1/users/{username}/following
```

Post APIs

```
POST   /api/v1/posts
GET    /api/v1/posts/{post_id}
DELETE /api/v1/posts/{post_id}
PUT    /api/v1/posts/{post_id}
POST   /api/v1/posts/{post_id}/like
DELETE /api/v1/posts/{post_id}/like
GET    /api/v1/posts/{post_id}/likes
POST   /api/v1/posts/{post_id}/comments
GET    /api/v1/posts/{post_id}/comments
```

Feed APIs

```
GET    /api/v1/feed/home?cursor={cursor}&limit=20
GET    /api/v1/feed/explore?cursor={cursor}&limit=20
GET    /api/v1/feed/user/{username}?cursor={cursor}&limit=20
```

Story APIs

```
POST   /api/v1/stories
GET    /api/v1/stories/feed
POST   /api/v1/stories/{story_id}/view
```

API Response Format

```
{
  "success": true,
  "data": {
    "post_id": "123abc",
    "user": {
      "username": "alice",
      "profile_picture": "https://cdn.instagram.com/..."
    },
    "media": [
      {
        "url": "https://cdn.instagram.com/...",
        "type": "image",
        "width": 1080,
        "height": 1080
      }
    ],
    "caption": "Beautiful sunset! 🌅",
    "likes_count": 1523,
    "comments_count": 89,
    "created_at": "2025-01-08T12:00:00Z"
  },
  "pagination": {
    "next_cursor": "xyz789",
    "has_more": true
  }
}
```

Deep Dives

1. News Feed Generation

Challenge: Generate personalized feed for 500M users efficiently

Approach 1: Fan-out on Write (Push Model)

Process:

1. User posts a photo
2. System immediately pushes to all followers' feeds
3. Followers read pre-computed feeds

Pros:

- Fast read (pre-computed)
- Real-time updates

Cons:

- Slow write for users with millions of followers (celebrity problem)
- Wastes computation for inactive users
- High write amplification

Approach 2: Fan-out on Read (Pull Model)

Process:

1. User posts a photo (stored in database)
2. When follower requests feed, system queries posts from all followed users
3. Merge and rank posts in real-time

Pros:

- Fast write
- No wasted computation
- Always up-to-date

Cons:

- Slow read (compute on every request)
- High read latency for users following many people

Hybrid Approach (Instagram's Solution)

For Normal Users (< 10K followers):

- Fan-out on write
- Pre-compute feeds asynchronously

For Celebrities (> 10K followers):

- Fan-out on read
- Merge celebrity posts at read time

Implementation:

```python
def generate_feed(user_id, limit=20):
    # Get pre-computed feed from cache
    cached_posts = redis.zrevrange(f"feed:{user_id}", 0, limit)

    # Get celebrity posts on-demand
    celebrity_following = get_celebrity_following(user_id)
    celebrity_posts = db.query(
        "SELECT * FROM posts WHERE user_id IN (%s)
         ORDER BY created_at DESC LIMIT 50",
        celebrity_following
    )

    # Merge and rank
    merged = merge_and_rank(cached_posts, celebrity_posts)
    return merged[:limit]
```

Feed Ranking Algorithm:

```
Score = (likes × 1.0) + (comments × 2.0) + (shares × 3.0) +
        (recency_factor × 10.0) + (user_affinity × 5.0)

Where:
- recency_factor: 1.0 for posts < 1 hour, decays exponentially
- user_affinity: engagement history between users
```

2. Media Upload & Storage

Upload Flow:

```
1. Client → API Server: Request upload URL
2. API Server → S3: Generate pre-signed URL
3. API Server → Client: Return pre-signed URL
4. Client → S3: Direct upload (bypasses app server)
5. Client → API Server: Confirm upload with S3 key
6. API Server → Queue: Trigger async processing
7. Worker: Generate thumbnails, apply filters
8. Worker → S3: Store processed images
9. Worker → DB: Update post with media URLs
10. Worker → Feed Service: Fanout to followers
```

Image Processing:

- **Original**: Stored as-is
- **Large** (1080px): For desktop web
- **Medium** (640px): For mobile feed
- **Thumbnail** (150px): For grid view

CDN Strategy:

- Push processed images to CDN immediately
- Use CloudFront with multiple origin servers
- Set TTL based on content type:
  - User avatars: 1 day
  - Posts: 30 days
  - Stories: 1 hour (24-hour expiry)

3. Timeline Ranking (Algorithmic Feed)

Factors Considered:

1. **Post Engagement** (40%)
   - Likes, comments, shares, saves
   - Engagement velocity (likes per hour)
2. **User Affinity** (30%)
   - How often user engages with poster
   - DM frequency, tag frequency
   - Profile visits
3. **Recency** (20%)
   - Exponential decay: recent posts weighted higher
   - Time since post creation
4. **Content Type** (10%)
   - User preference (photos vs videos)
   - Completion rate for videos

ML Model:

- Train ranking model on user engagement data
- Features: user demographics, post features, historical interactions
- Update model daily/weekly
- A/B test new models

4. Notifications System

Architecture:

```
[Event] → [Kafka] → [Notification Worker] → [Push Service]
                                          → [WebSocket Server]
                                          → [Email Service]
```

Notification Types:

- **Real-time**: Likes, comments, new followers (WebSocket/Push)
- **Batched**: Daily digest emails
- **Prioritized**: Direct messages (high priority push)

Deduplication:

- Group similar notifications: "Alice and 10 others liked your post"
- Time window: 5 minutes for grouping

User Preferences:

- Allow users to configure notification settings
- Opt-out of specific notification types

5. Search System

Architecture:

```
[User Query] → [API Server] → [Elasticsearch Cluster]
                                        ↓
                                [Index Documents]
                              (Users, Posts, Hashtags)
```

Indexing Strategy:

- **Users**: Index username, full name, bio
- **Hashtags**: Index hashtag name, post count
- **Posts**: Index caption text (optional)

Search Features:

- **Autocomplete**: Prefix matching on usernames
- **Fuzzy search**: Handle typos (edit distance ≤ 2)
- **Ranking**:
  - Verified users ranked higher
  - Users with more followers ranked higher
  - Recent posts ranked higher

Update Strategy:

- Near real-time indexing (< 1 second delay)
- Kafka pipeline: DB change → Kafka → Elasticsearch indexer

6. Story System

Design Challenges:

- 24-hour auto-expiry

- High read volume (viewed multiple times)
- Real-time view tracking

Architecture:

```
Stories Table:
- story_id (PK)
- user_id (FK, indexed)
- media_url
- created_at
- expires_at (indexed for cleanup)

Story_Views Table:
- story_id (FK)
- viewer_id (FK)
- viewed_at
- PRIMARY KEY (story_id, viewer_id)
```

Optimization:

- Cache active stories in Redis (24-hour TTL)
- Use Redis Sorted Set for view count:

  ```
  ZINCRBY story:{story_id}:views 1 {viewer_id}
  ```

- Background job to delete expired stories every hour
- Store media in S3 with 25-hour lifecycle policy

7. Direct Messaging

Architecture:

```
[Client] ←→ [WebSocket Server] ←→ [Message Queue] ←→ [Message Service]
                                                            ↓
                                                    [Message Database]
                                                    (Cassandra/MongoDB)
```

Database Schema (Cassandra):

```
Messages Table:
- conversation_id (partition key)
- message_id (clustering key, timeuuid)
- sender_id
- receiver_id
- content
```

```
- created_at
- read_at

Conversations Table:
- conversation_id (PK)
- participants (Set<user_id>)
- last_message_id
- updated_at
```

Features:

- Read receipts (update read_at timestamp)
- Message delivery status (sent, delivered, read)
- End-to-end encryption (optional)

Scaling:

- Partition by conversation_id
- Use WebSocket for real-time delivery
- Fall back to push notifications if user offline

Scalability & Reliability

Horizontal Scaling

Stateless Components:

- API servers can be added/removed dynamically
- Auto-scaling based on metrics:
  - CPU > 70% → scale up
  - CPU < 30% for 10 min → scale down

Database Scaling:

- **Read replicas** for read-heavy workload
- **Sharding** for write-heavy workload
- **Caching** to reduce database load by 80-90%

High Availability

Multi-Region Deployment:

```
Primary Region: us-east-1 (50% traffic)
Secondary Region: us-west-2 (30% traffic)
Tertiary Region: eu-west-1 (20% traffic)

Benefits:
- Low latency for global users
```

```
- Disaster recovery
- Load distribution
```

Replication:

- Master-Slave for SQL databases
- Multi-master for NoSQL (Cassandra)
- Cross-region S3 replication

Health Checks:

- Load balancer health checks (every 10s)
- Application-level health endpoint: `/health`
- Database connection pool monitoring

Disaster Recovery

Backup Strategy:

- **Database**: Daily full backup, continuous WAL archiving
- **S3**: Cross-region replication enabled
- **Cache**: Rebuilt from database (acceptable loss)

Recovery Time Objective (RTO): < 1 hour
Recovery Point Objective (RPO): < 5 minutes

Monitoring & Alerting

Metrics to Monitor:

- **Application**: Request rate, error rate, latency (p50, p95, p99)
- **Infrastructure**: CPU, memory, disk I/O
- **Business**: DAU, posts/day, engagement rate

Tools:

- **Metrics**: Prometheus, Datadog, CloudWatch
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Tracing**: Jaeger, AWS X-Ray
- **Alerting**: PagerDuty, Opsgenie

Security

Authentication & Authorization

- **JWT tokens** for stateless authentication
- **OAuth 2.0** for third-party integrations
- **Refresh tokens** (7 days) and access tokens (1 hour)
- **Two-factor authentication** (2FA) for enhanced security

Data Security

- **HTTPS** for all communications (TLS 1.3)
- **Encryption at rest** for S3 (AES-256)
- **Database encryption** for sensitive fields (email, phone)
- **Secrets management**: AWS Secrets Manager, HashiCorp Vault

Rate Limiting

```
Per User Limits:
- API requests: 1000/hour
- Post uploads: 50/day
- Comments: 200/hour
- Likes: 500/hour
- Follow actions: 200/hour

Per IP Limits:
- Registration: 10/hour
- Login attempts: 20/hour
```

Implementation: Token bucket algorithm in Redis

Content Moderation

- **Automated** AI-based moderation (NSFW detection)
- **Manual** review queue for flagged content
- **User reporting** system
- **Shadow banning** for policy violations

Trade-offs & Alternatives

1. SQL vs NoSQL

Chose: Hybrid (PostgreSQL + Cassandra)

- PostgreSQL for user data (ACID, complex queries)
- Cassandra for posts (write-heavy, time-series)

Alternative: Pure NoSQL (DynamoDB)

- Simpler architecture
- Better write scalability
- Loss of complex queries and transactions

2. Push vs Pull for Feed

Chose: Hybrid (Push for normal users, Pull for celebrities)

- Balanced write and read performance
- Handles celebrity problem

Alternative: Pure Pull

- Simpler implementation
- Slower reads
- Not suitable for real-time feed

3. Monolith vs Microservices

Chose: Microservices

- Independent scaling
- Team autonomy
- Technology diversity

Alternative: Modular Monolith

- Simpler deployment
- No network overhead
- Faster initial development

4. Synchronous vs Asynchronous Processing

Chose: Async for heavy operations

- Better user experience (immediate response)
- Decoupled components
- Better resource utilization

Alternative: Synchronous

- Simpler code flow
- Immediate feedback
- Potential timeouts under load

Additional Considerations

1. Data Privacy & Compliance

- **GDPR compliance** (Europe)
- **CCPA compliance** (California)
- **Data deletion** on user request
- **Data export** feature (download your data)

2. Analytics & Business Intelligence

```
[User Actions] → [Kafka] → [Stream Processor] → [Data Warehouse]
                              (Flink/Spark)      (Redshift/BigQuery)
                                                        ↓
                                                   [BI Tools]
                                                   (Tableau/Looker)
```

Metrics Tracked:

- User engagement (DAU, WAU, MAU)
- Content metrics (posts/day, engagement rate)
- Platform health (error rates, latency)
- Business metrics (revenue, growth rate)

3. A/B Testing Framework

- Feature flags for gradual rollouts
- User segmentation (random, geographic, demographic)
- Metric tracking per experiment
- Statistical significance testing

4. Content Recommendation

ML Pipeline:

- Collaborative filtering (users like you liked...)
- Content-based filtering (similar to posts you liked)
- Deep learning models (computer vision for similar images)
- Real-time personalization

Technology Stack Summary

Layer	Technology	Purpose
CDN	CloudFront/Akamai	Static content delivery
Load Balancer	AWS ALB, NGINX	Traffic distribution
API Gateway	AWS API Gateway, Kong	Request routing, auth
App Servers	Node.js, Python (Django)	Business logic
Caching	Redis Cluster, Memcached	Performance
SQL Database	PostgreSQL	User data, transactions
NoSQL Database	Cassandra, DynamoDB	Posts, messages
Graph Database	Neo4j	Social graph
Object Storage	Amazon S3	Media files

Layer	Technology	Purpose
Message Queue	Apache Kafka, RabbitMQ	Async processing
Search	Elasticsearch	Full-text search
Real-time	WebSocket, Socket.io	Live updates
Monitoring	Prometheus, Datadog	Metrics & alerts
Logging	ELK Stack	Centralized logs
Container	Docker, Kubernetes	Orchestration

Scalability Metrics

Instagram's Actual Numbers (2011 - Early Days)

- **14 million users**
- **150 million photos**
- **Terabytes of data**
- **3 engineers**
- **100+ EC2 instances**
- **25+ Django app servers**
- **12 PostgreSQL instances**
- **Several Redis instances**

Modern Scale Estimates (2025)

- **2+ billion monthly active users**
- **95 million posts per day**
- **500 million stories per day**
- **100+ petabytes of storage**
- **Millions of requests per second**

Interview Talking Points

Key Design Decisions

1. **Why hybrid feed model?** Balance between performance and freshness
2. **Why Redis for feed cache?** Fast, supports sorted sets, pub/sub
3. **Why S3 for media?** Durable (99.999999999%), scalable, cheap
4. **Why CDN?** Global distribution, reduced latency, bandwidth savings
5. **Why Cassandra for posts?** Write-heavy workload, excellent for time-series
6. **Why message queue?** Decouple services, handle traffic spikes, async processing
7. **Why microservices?** Independent scaling, team autonomy, fault isolation

Potential Bottlenecks & Solutions

1. **Database becomes bottleneck** → Add sharding, read replicas, caching
2. **Feed generation is slow** → Use hybrid push-pull, pre-compute feeds
3. **Image storage grows** → Use S3 lifecycle policies, compress images
4. **Celebrity posts overload** → Fan-out on read for high-follower users
5. **Search is slow** → Use Elasticsearch with proper indexing

---

System Design Patterns Used

1. **CQRS (Command Query Responsibility Segregation)**

- Separate read and write models
- Write to master DB, read from replicas/cache
- Optimize each for its specific use case

2. **Event Sourcing**

- Store all changes as events in Kafka
- Rebuild state from event log if needed
- Useful for audit trail and analytics

3. **Circuit Breaker**

- Prevent cascading failures
- Fail fast when service is down
- Implement using Hystrix or Resilience4j

4. **Bulkhead Pattern**

- Isolate resources for different features
- Prevent one feature from consuming all resources
- Separate thread pools for critical vs non-critical operations

5. **Cache-Aside Pattern**

- Application manages cache explicitly
- Check cache first, then database
- Update cache after database write

Detailed Component Interactions

Upload Photo Flow

```
1. User selects photo in mobile app
2. App calls POST /api/v1/posts/upload
3. API Gateway authenticates request
4. Post Service generates pre-signed S3 URL
5. App uploads directly to S3
```

```
6. App calls POST /api/v1/posts/confirm
7. Post Service:
   - Creates post entry in database
   - Publishes event to Kafka
8. Image Processing Worker:
   - Generates thumbnails
   - Applies filters
   - Stores processed images in S3
   - Updates database with URLs
9. Feed Fanout Worker:
   - Retrieves follower list (< 10K check)
   - Pushes post to followers' Redis feeds
   - For celebrity: skip fanout
10. Notification Worker:
    - Sends push to followers
11. Search Indexer:
    - Updates Elasticsearch index
12. Analytics Worker:
    - Logs event to data warehouse
```

View Feed Flow

```
1. User opens app/refreshes feed
2. App calls GET /api/v1/feed/home
3. API Gateway authenticates & rate limits
4. Feed Service:
   - Checks Redis cache for user's feed
   - If cache miss: generate feed
     a. Get pre-computed posts (normal users)
     b. Get celebrity posts (on-demand)
     c. Merge and rank
   - Return top 20 posts
5. For each post:
   - Get media URLs from CDN
   - Get like/comment counts from cache
6. App renders feed
7. For videos: lazy load on scroll
```

Like Post Flow

```
1. User taps like button
2. App calls POST /api/v1/posts/{post_id}/like
3. Post Service:
   - Writes to Likes table
   - Increments like count in Redis
   - Publishes event to Kafka
4. Notification Worker:
   - Sends notification to post author
```

```
5. Analytics Worker:
   - Updates engagement metrics
6. Feed Ranker:
   - Updates post ranking score
```

Performance Optimizations

1. Database Query Optimization

```sql
-- Bad: N+1 query problem
SELECT * FROM posts WHERE user_id IN (following_list);
-- Then for each post, query likes/comments

-- Good: Use JOIN with aggregation
SELECT p.*,
       COUNT(DISTINCT l.user_id) as like_count,
       COUNT(DISTINCT c.comment_id) as comment_count
FROM posts p
LEFT JOIN likes l ON p.post_id = l.post_id
LEFT JOIN comments c ON p.post_id = c.post_id
WHERE p.user_id IN (following_list)
GROUP BY p.post_id
ORDER BY p.created_at DESC
LIMIT 20;
```

2. Connection Pooling

- Use connection pools for database connections
- Avoid creating new connection per request
- PgBouncer for PostgreSQL
- Typical pool size: 20-50 connections per app server

3. Lazy Loading

- Don't load all media at once
- Load images as user scrolls
- Placeholder while loading (blur hash)

4. Compression

- Compress API responses (gzip)
- Use WebP format for images (30% smaller than JPEG)
- Video compression: H.264/H.265

5. Database Indexing

```
-- Critical indexes for fast queries
CREATE INDEX idx_posts_user_created ON posts(user_id, created_at DESC);
CREATE INDEX idx_posts_created ON posts(created_at DESC);
CREATE INDEX idx_likes_post ON likes(post_id);
CREATE INDEX idx_followers_followee ON followers(followee_id);
CREATE INDEX idx_comments_post ON comments(post_id, created_at);
```

Failure Scenarios & Mitigation

Scenario 1: Database Master Failure

Impact: Cannot write new posts/likes/follows
Mitigation:

- Automatic failover to slave (promote to master)
- Use Patroni/repmgr for auto-failover
- Keep write queue in Kafka during failover
- Expected downtime: < 30 seconds

Scenario 2: Redis Cache Failure

Impact: Increased database load, slower feeds
Mitigation:

- Redis Cluster with multiple replicas
- Graceful degradation: serve from database
- Rate limit during cache failure
- Expected impact: 2-3x slower responses

Scenario 3: S3/CDN Unavailability

Impact: Cannot serve images
Mitigation:

- Multi-CDN strategy (CloudFront + Akamai)
- S3 has 99.99% availability SLA
- Show cached thumbnails from app
- Placeholder images as fallback

Scenario 4: Message Queue Backlog

Impact: Delayed notifications, slow feed updates
Mitigation:

- Scale up consumer workers
- Prioritize critical events (DMs over likes)
- Drop non-critical events if backlog > threshold

- Alert ops team

Scenario 5: Celebrity Post Storm

Impact: Feed fanout overwhelms system
Mitigation:

- Rate limit fanout workers
- Fan-out on read for celebrities
- Cache celebrity's recent posts
- Progressive fanout over time

Cost Optimization

Storage Costs

```
Current: 69 PB over 10 years

Optimization strategies:
1. Compress images (WebP): Save 30% → 48 PB
2. Lifecycle policies (Glacier): Save 50% on old data
3. Deduplication: Save 10-15% (similar photos)

Total savings: ~40% cost reduction
```

Compute Costs

```
Optimization strategies:
1. Auto-scaling: Scale down during off-peak hours
2. Spot instances: Save 60-80% for non-critical workloads
3. Reserved instances: Save 30-50% for baseline capacity
4. Right-sizing: Match instance type to workload
```

Bandwidth Costs

```
Optimization strategies:
1. CDN caching: 90% cache hit ratio
2. Image compression: Reduce bandwidth by 30%
3. Smart prefetching: Only fetch visible content
4. Regional data centers: Keep data close to users
```

Future Enhancements

Phase 1 (Months 1-6)

- [ ] Basic posting and feed
- [ ] User profiles and following
- [ ] Like and comment functionality
- [ ] Basic search

Phase 2 (Months 7-12)

- [ ] Stories feature
- [ ] Direct messaging
- [ ] Push notifications
- [ ] Algorithmic feed ranking

Phase 3 (Year 2)

- [ ] Video support (Reels)
- [ ] Live streaming
- [ ] Shopping features
- [ ] AR filters
- [ ] Advanced ML recommendations

References & Further Reading

System Design Resources

1. **System Design Primer** - https://github.com/donnemartin/system-design-primer
2. **Instagram Architecture (2011)** - http://highscalability.com/blog/2011/12/6/instagram-architecture-14-million-users-terabytes-of-photos.html
3. **Facebook News Feed** - https://www.facebook.com/notes/facebook-engineering/
4. **Twitter Timeline** - https://blog.twitter.com/engineering

Instagram Engineering Blog Posts

1. **Sharding & IDs at Instagram** - How they generate unique IDs at scale
2. **Storing Millions of Key-Value Pairs in Redis** - Feed caching strategy
3. **What Powers Instagram** - Original tech stack overview

Related Papers

1. **Cassandra: A Decentralized Structured Storage System**
2. **Dynamo: Amazon's Highly Available Key-value Store**
3. **MapReduce: Simplified Data Processing on Large Clusters**
4. **The Google File System**

Books

1. **Designing Data-Intensive Applications** by Martin Kleppmann

2. **System Design Interview** by Alex Xu
3. **Building Microservices** by Sam Newman
4. **Site Reliability Engineering** by Google

Appendix

Latency Numbers

```
L1 cache reference:              0.5 ns
L2 cache reference:              7 ns
Main memory reference:           100 ns
SSD random read:                 150 µs
Send 1 MB over network:          10 ms
Disk seek:                       10 ms
Read 1 MB from SSD:              1 ms
Read 1 MB from HDD:              30 ms
Round trip in same datacenter:   0.5 ms
Round trip CA to Netherlands:    150 ms
```

Powers of Two

```
8 bits = 1 byte
1024 bytes = 1 KB
1024 KB = 1 MB
1024 MB = 1 GB
1024 GB = 1 TB
1024 TB = 1 PB
```

Common QPS Estimates

```
Small service: 100–1K QPS
Medium service: 1K–10K QPS
Large service: 10K–100K QPS
Very large service: 100K–1M+ QPS
```

Document Version: 1.0
Last Updated: January 8, 2025
Author: System Design Interview Prep
Status: Complete & Interview-Ready ✅