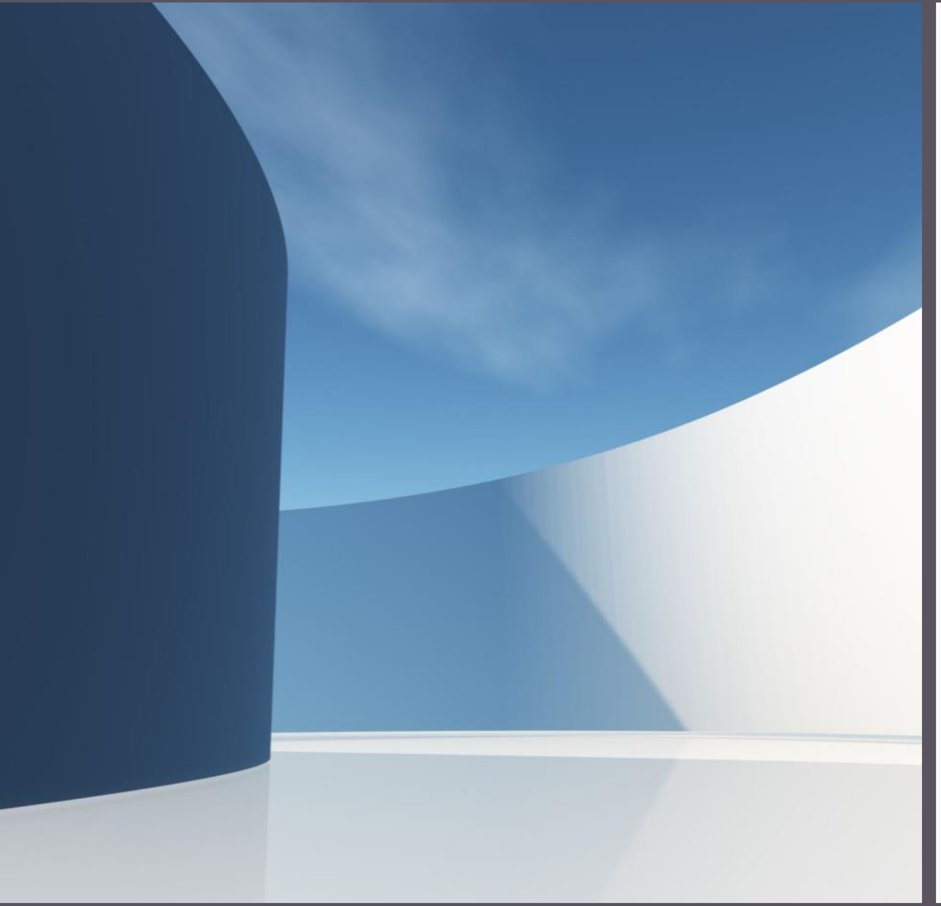


INSTAGRAM CLONE

BSDSF22M003 – EMAN ASIF

BSDSF22M004 – MUSQAN YOUSAF

BSDSF22M008 – MAHAM JAMIL

BSDSF21M059 – MISBAH FATEH

Major Project Requirements

Functional Requirements

- User authentication
- Profile management
- Upload reels & stories
- Like, comment, follow
- Personalized feed
- Media streaming

Non-Functional Requirements

- Low latency (<200ms)
- Horizontal scalability
- Fault tolerance
- High availability
- Eventual consistency
- Real-time interactions

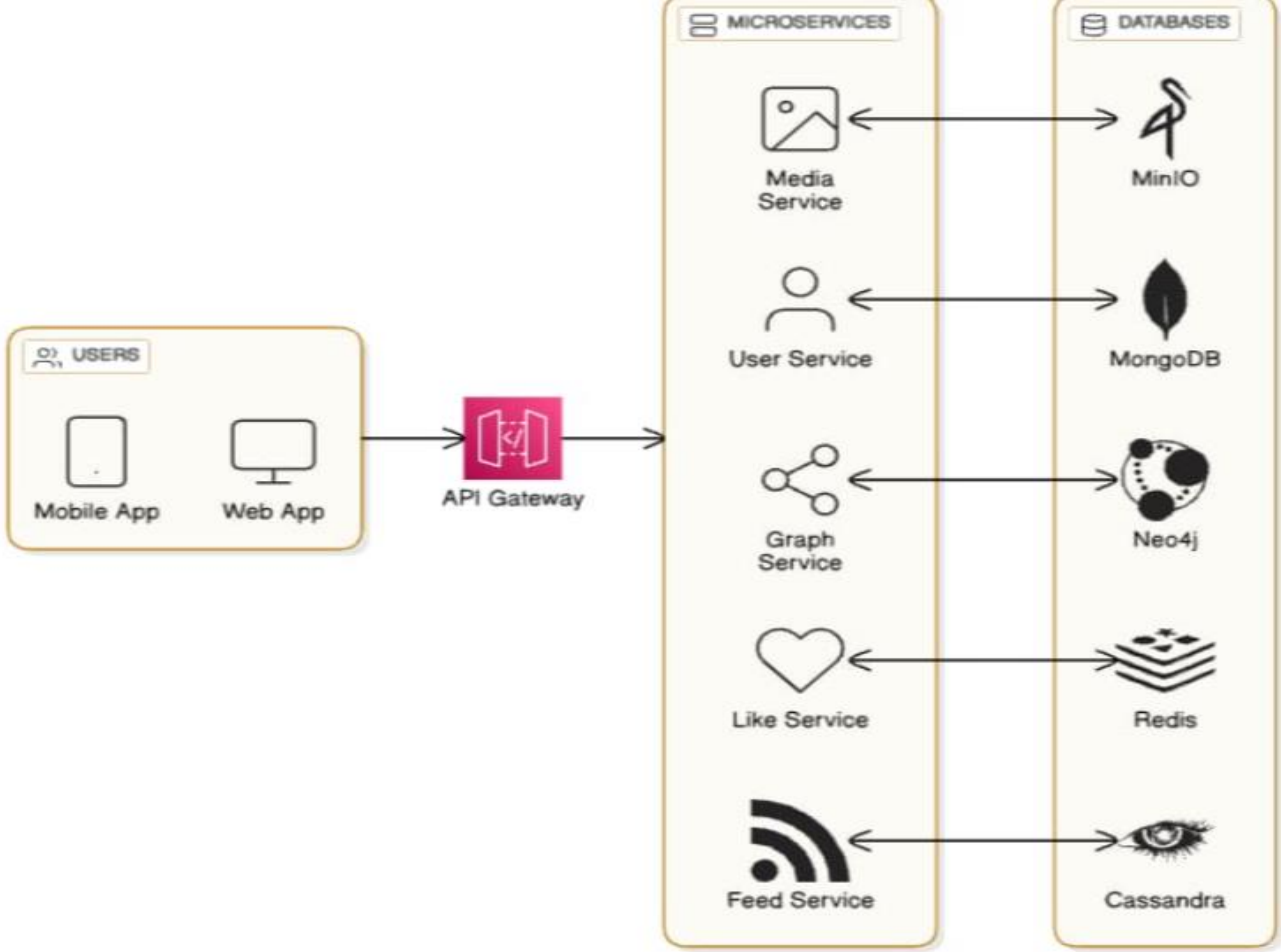
Big Data Design Perspective

Big Data Aspect	Manifestation
Volume	Millions of reels, likes, comments
Velocity	Likes, follows, comments in real time
Variety	Text, video, images, graphs
Veracity	Durable writes + backups
Value	Engagement, recommendations

Architectural Style

- Microservices-oriented
- Polyglot persistence

High Level Architecture



Subsystem	Responsibility
Redis	Real-time counters & fast interactions
MongoDB	User-centric & document-oriented data
Cassandra	High-volume feed
Neo4j	Social relationships & recommendations
MinIO	Media (video/image) storage

Redis – Likes, Counters & Real-Time State

Why Instagram Needs Redis

- Likes must feel instant
- Counters are updated millions of times per second
- Users expect immediate UI feedback

Why This Design Is Smart

- `INCR` and `DECR` are atomic
  No race condition

Big Data Perspective

- Handles high velocity
- Acts as a hot data layer

MongoDB – User Profiles, Comments & Durable Data

Why MongoDB Fits Instagram

- User profiles vary (bio, avatar, settings)
- Comments have nested replies
- Flexible schema needed

What It Stores

- Users
- Comments (threaded)
- Durable likes (backup)
- Follows (fallback)

Big Data Perspective

- Handles variety(semi-structured data)

Cassandra – Feed, Reels & Time-Series Data

Why Cassandra Is Critical

Instagram feeds are:

- Read-heavy
- Write-heavy
- Chronologically ordered
- Personalized per user

Key Tables

- `reels` → Global reel metadata
- `user_reels` → Reels by a user
- `timeline` → Feed per user

Why Partition by `user_id`

- Each user reads their own feed
- Fast lookups
- No joins needed

Big Data Perspective

- Handles volume (billions of rows)
- Handles velocity (continuous writes)
- Built for distributed systems

Neo4j – Social Graph (Follows)

What Neo4j Is

A graph database where:

- Nodes = users
- Edges = follows

Why Graph DB Is Needed

Queries like:

- Who follows me?
- Who do I follow?
- Friends of friends?
- Suggested users?

Graph Example

```
(User A)-[:FOLLOWS]->(User B)
```

Big Data Perspective

- Handles relationship-heavy data
- Enables recommendation algorithms

MinIO – Media Storage

What MinIO Is

An object storage system, compatible with Amazon S3.

Why Media Is Separate

- Videos are large (MBs/GBs)
- Needs CDN support

What it stores:-

reels-media
bucket: reels/{fileId}.{
ext}- Reel media files

Big Data Perspective

- Handles unstructured data
- Supports scalability & durability
- Cheap storage for large volume

Data Flows

Media Upload → MinIO

- Video stored as reels/{uuid}.mp4
- Storage URL returned

Metadata Storage → Cassandra

- Insert into reels (global reel data)
- Insert into user_reels (user’s reels)

Timeline Update → Cassandra

- Followers fetched via **Neo4j**
- Reel added to each follower’s timeline

Like Counter Init → Redis

- likes:{reelId} = 0

User Posts a Reel

Key Design Choice:
Fan-out on write for fast feed reads.

- **Fetch Feed → Cassandra**
  - Query `timeline` by `user_id`
- **User Info Enrichment → MongoDB / Neo4j**
  - Author name, avatar, relationships
- **Like Counts → Redis**
  - Read `likes:{reelId}`
- **Comments → MongoDB**
  - Fetch latest comments & replies
- **Response → Frontend**
  - Combined feed data returned

User Views Feed

Key Design Choice:
Read-optimized path using caching and precomputed timelines.

- **Fast Update → Redis**

- `INCR likes:{reelId}`
- Cache user like: `like:{reelId}:{userId}`

- **Durable Storage → MongoDB (Async)**
- Store like event for analytics & recovery

- **Frontend Response**
- Optimistic UI update (no waiting)

User Likes a Reel

Key Design Choice:
Redis for real-time interaction + MongoDB for persistence.

Design Goals and Their Implementation

Scalability

Goal:

The system must handle growth in:

- Users
- Reels
- Likes
- Comments
- Media uploads

Implementation in Our Project:

- **Cassandra** is used for feeds and reels → horizontally scalable by adding nodes.
- **Redis Cluster** handles millions of like operations per second.
- **MongoDB Replica Sets** allow scaling read traffic.
- Each database can be scaled **independently**, preventing bottlenecks.

High Performance & Low Latency

Goal:

User interactions (likes, feed loading) must feel instantaneous.

Implementation:

- Likes are handled by **Redis** using atomic `INCR` operations.
- Feed queries are optimized using **Cassandra partitioning by user_id**.
- Media is served directly from **MinIO**,

Backend APIs are **stateless**, enabling fast response times.

Availability & Fault Tolerance

Goal:

The system should remain functional even if a component fails.

Implementation:

- **Redis failure → MongoDB durable likes** used for recovery.
- **Neo4j failure → MongoDB follows collection** as fallback.
- **Cassandra replication factor > 1** ensures no single point of failure.
- **MongoDB replica sets** provide automatic failover.
- Media remains accessible via MinIO even if app servers go down.

Result: No cascading failures.

System Decomposition (Modular Architecture)

Goal:

Break the system into manageable, independent components.

Decomposition in Our Project

Service	Database	Responsibility
User Service	MongoDB	Profiles, settings
Feed Service	Cassandra	Timelines, reels
Like Service	Redis + MongoDB	Likes & counters
Graph Service	Neo4j	Follows & relations
Media Service	MinIO	Video/image storage

Result: Each service can be developed, deployed, and scaled separately.

Consistency Model

Goal:

Balance speed and correctness.

Implementation:

- **Eventual consistency** for likes:
  - Redis = fast
  - MongoDB = durable
- **Strong consistency** in Neo4j for follow relationships.
- **Tunable consistency** in Cassandra for feed queries.

Result: Correct data with high performance.

Concurrency Handling

Goal:

Handle thousands of simultaneous users without conflicts.

Implementation

- **Redis atomic operations** (`INCR`, `DECR`) prevent race conditions.
- **Cassandra partition isolation** ensures users don’t block each other.
- **Async background workers** handle MongoDB durable writes.

Result: High concurrency with no data corruption.

Data Management Strategy

Goal:

Store each type of data in the most suitable system.

Implementation

Data Type	Storage
Hot counters	Redis
User metadata	MongoDB
Feed data	Cassandra
Relationships	Neo4j
Media files	MinIO

Result: Optimal performance and storage efficiency.

Boundary Conditions & Failure Scenarios

Goal:

Define system behavior during failures.

Implementation

Failure	System Response
Redis down	MongoDB likes used
Neo4j down	MongoDB follows
Cassandra node down	Replica node serves

Result: Graceful degradation instead of crashes.

Boundary Conditions & Failure Scenarios

Goal:

Define system behavior during failures.

Implementation

Failure	System Response
Redis down	MongoDB likes used
Neo4j down	MongoDB follows
Cassandra node down	Replica node serves

Result: Graceful degradation instead of crashes.

Big Data Manifestation Summary

Big Data Principle	Implementation
Distributed storage	Cassandra, MongoDB
High throughput	Redis, Cassandra
Eventual consistency	Redis + MongoDB
Horizontal scaling	All components
Fault tolerance	Replication everywhere

Big Data Principles Implementation
1. Distributed Storage (Cassandra, MongoDB)
Implementation: Cassandra - Distributed Keyspace with Replication

```
await client.execute(`
  CREATE KEYSPACE IF NOT EXISTS instagram
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 2}
`);

// Partitioned tables for distributed storage
CREATE TABLE user_reels (
  user_id TEXT,
  reel_id UUID,
  created_at TIMESTAMP,
  PRIMARY KEY (user_id, created_at, reel_id)
) WITH CLUSTERING ORDER BY (created_at DESC);

CREATE TABLE timeline (
  user_id TEXT,
  reel_id UUID,
  author_id TEXT,
  created_at TIMESTAMP,
  PRIMARY KEY (user_id, created_at, reel_id)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

Why:
- `replication_factor` enables data replication across nodes
- Partitioned by `user_id` for distributed storage

Distributed Storage

Distributed Storage

Implementation: MongoDB - Distributed Document Storage

File: `server/db/mongo.js`

```javascript
// MongoDB connection with replica set support
const uri = process.env.MONGO_URI ||
  `mongodb://${mongoUser}:${mongoPass}@${mongoHost}:${mongoPort}/${mongoDb}?authSource=admin`;

const client = new MongoClient(uri);

// Collections are automatically distributed across shards in production
await client.connect();
db = client.db('instagram');
```

Why:
- MongoDB supports replica sets and sharding for horizontal distribution
- Collections (`users`, `comments`, `follows`) are stored across multiple nodes
- Automatic data distribution based on shard key

2. High Throughput (Redis, Cassandra)
Implementation: Redis - High Throughput Like Operations

```javascript
router.post('/:reelId', async (req, res) => {
  const { reelId } = req.params;
  const { userId } = req.body;

  const key = `likes:${reelId}`;
  const userLikeKey = `like:${reelId}:${userId}`;

  // Atomic increment operation - handles millions of likes per second
  const newCount = await redis.incr(key);
  await redis.set(userLikeKey, '1', { EX: 86400 * 30 });

  // Async write to MongoDB for durability (non-blocking)
  mongo.collection('likes').insertOne({
    reelId,
    userId,
    createdAt: new Date(),
  }).catch(err => console.error('Error persisting like:', err));

  res.json({ likesCount: newCount, liked: true });
});
```

Why:
- `redis.incr()` is atomic and sub-millisecond latency
- Can handle millions of operations per second
- Non-blocking async writes to MongoDB

High Thoughput

Implementation: Cassandra - High Throughput Reel Creation

File: `server/routes/reels.js`

```javascript
router.post('/', async (req, res) => {
  const { userId, caption, mediaUrl } = req.body;
  const cassandra = getClient();
  const reelId = Uuid.random();
  const now = new Date();

  // High-throughput writes - optimized for millions of reels
  await cassandra.execute(
    'INSERT INTO reels (reel_id, user_id, caption, media_url, created_at) VALUES (?, ?, ?
    [reelId, userId, caption || '', mediaUrl, now],
    { prepare: true }
  );

  // Insert into user_reels (partitioned by user_id)
  await cassandra.execute(
    'INSERT INTO user_reels (user_id, reel_id, created_at) VALUES (?, ?, ?)',
    [userId, reelId, now],
    { prepare: true }
  );
```

High Thoughput

```
  await cassandra.execute(
    'INSERT INTO user_reels (user_id, reel_id, created_at) VALUES (?, ?, ?)',
    [userId, reelId, now],
    { prepare: true }
  );

  // Batch insert into followers' timelines
  for (const followerId of followers) {
    await cassandra.execute(
      'INSERT INTO timeline (user_id, reel_id, author_id, created_at) VALUES (?, ?, ?, ?)',
      [followerId, reelId, userId, now],
      { prepare: true }
    );
  }
});
```

Why:
- Cassandra handles high write throughput (millions of writes/second)
- Partitioned tables distribute load across nodes
- Prepared statements optimize performance

High Thoughput

3. Eventual Consistency (Redis + MongoDB)

Implementation: Eventual Consistency Pattern for Likes

```
router.post('/:reelId', async (req, res) => {
  // Step 1: Fast write to Redis (immediate response)
  const newCount = await redis.incr(key);
  await redis.set(userLikeKey, '1', { EX: 86400 * 30 });

  // Step 2: Async write to MongoDB (eventual consistency)
  // Don't wait - MongoDB will catch up eventually
  const mongo = getDb();
  mongo.collection('likes').insertOne({
    reelId,
    userId,
    createdAt: new Date(),
  }).catch(err => console.error('Error persisting like:', err));

  // Return immediately with Redis value
  res.json({ likesCount: newCount, liked: true });
});
```

Why:
- Redis provides immediate response (strong consistency for reads)
- MongoDB provides durability (eventual consistency for persistence)
- If Redis fails, can rebuild from MongoDB
- If MongoDB is slow, Redis still serves reads

Eventual Consistency

4. Horizontal Scaling (All Components)
Implementation: Docker Compose - Scalable Architecture

```yaml
services:
  redis:
    image: redis:7-alpine
    # Can scale: docker-compose up -d --scale redis=3
    # Use Redis Cluster for horizontal scaling
  mongo:
    image: mongo:7
    # Supports replica sets and sharding
    # Scale: Add more nodes to replica set
  cassandra:
    image: cassandra:4.1
    # Native horizontal scaling - add more nodes
    # replication_factor can be increased for more replicas
  neo4j:
    image: neo4j:5
    # Supports cluster mode for horizontal scaling
  minio:
    image: minio/minio
    # Supports distributed MinIO cluster
```

Why:
- Each service can be scaled independently
- Cassandra and MongoDB support native clustering
- Redis supports Redis Cluster mode
- MinIO supports distributed mode

Horizontal Scaling

Horizontal Scaling

Implementation: Cassandra Connection Pooling for Scaling

*File**: `server/db/cassandra.js`

```javascript
onst cassandraConfig = {
 contactPoints: [process.env.CASSANDRA_HOSTS?.split(':')[0] || 'localhost'],
 port: parseInt(process.env.CASSANDRA_HOSTS?.split(':')[1] || '9042'),
 localDataCenter: 'datacenter1',
 // Connection pooling for horizontal scaling
 // Can connect to multiple nodes
;

onst client = new cassandra.Client(cassandraConfig);

/ Keyspace with replication for horizontal scaling
wait client.execute(`
 CREATE KEYSPACE IF NOT EXISTS instagram
 WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
);
```

*Why**:
 `contactPoints` can list multiple nodes
 Replication factor enables data distribution
 Client automatically load balances across nodes

Ln 95, Col 1 Spaces: 2 UTF-8 LF

Horizontal Scaling

Implementation: MongoDB Connection String for Scaling

File: `server/db/mongo.js`

```javascript
// Connection string supports replica sets and sharding
const uri = process.env.MONGO_URI ||
  `mongodb://${mongoUser}:${mongoPass}@${mongoHost}:${mongoPort}/${mongoDb}?authSource=admin`;

// For replica sets:
// mongodb://host1:27017,host2:27017,host3:27017/instagram?replicaSet=rs0

// For sharding:
// mongodb://mongos1:27017,mongos2:27017/instagram

const client = new MongoClient(uri);
```

Why:
- Connection string can include multiple hosts
- Automatic failover and load balancing
- Supports replica sets and sharded clusters

5. Fault Tolerance (Replication Everywhere)
Implementation: Redis + MongoDB Dual Write for Fault Tolerance

```javascript
// Fast write to Redis
const newCount = await redis.incr(key);
await redis.set(userLikeKey, '1', { EX: 86400 * 30 });

// Async write to MongoDB for fault tolerance
// If Redis fails, can rebuild from MongoDB
const mongo = getDb();
mongo.collection('likes').insertOne({
  reelId,
  userId,
  createdAt: new Date(),
}).catch(err => console.error('Error persisting like:', err));
```

Why:
- Redis provides fast access (primary)
- MongoDB provides durability (backup)
- If Redis fails, can rebuild cache from MongoDB
- If MongoDB fails, Redis still serves reads

Fault Tolerance

5. Fault Tolerance (Replication Everywhere)

Implementation: Follow Relationship Replication

```javascript
router.post('/:userId/follow', async (req, res) => {
  // Write to Neo4j (primary)
  if (neo4j) {
    const session = neo4j.session();
    try {
      await session.run(
        'MATCH (follower:User {id: $followerId}), (followed:User {id: $userId}) MERGE (follower)-[r:FOLLOWS]->(followed)',
        { followerId, userId }
      );
    } catch (error) {
      console.error('Error creating follow relationship in Neo4j:', error);
      // Continue to MongoDB fallback
    } finally {
      await session.close();
    }
  }

  // Always store in MongoDB (replication for fault tolerance)
  await mongo.collection('follows').updateOne(
    { followerId, followedId: userId },
    { $set: { followerId, followedId: userId, createdAt: new Date() } },
    { upsert: true }
  );
});
```

SUMMARY

Principle	Implementation	Code Location	Key Features
Distributed Storage	Cassandra keyspace with replication	server/db/cassandra.js	Partitioned tables, replication factor
Distributed Storage	MongoDB collections	server/db/mongo.js	Replica sets, sharding support
High Throughput	Redis atomic operations	server/routes/likes.js	redis.incr(), async writes
High Throughput	Cassandra batch inserts	server/routes/reels.js	Prepared statements, partitioned writes
Eventual Consistency	Redis + MongoDB dual write	server/routes/likes.js	Fast Redis, async MongoDB
Eventual Consistency	Neo4j + MongoDB fallback	server/routes/users.js	Primary Neo4j, backup MongoDB

SUMMARY

Principle	Implementation	Code Location	Key Features
Horizontal Scaling	Docker Compose services	docker-compose.yml	Independent scaling per service
Horizontal Scaling	Multi-node connections	server/db/cassandra.js, server/db/mongo.js	Multiple contact points
Fault Tolerance	Neo4j → MongoDB fallback	server/routes/reels.js	Automatic fallback on failure
Fault Tolerance	Redis + MongoDB replication	server/routes/likes.js	Dual write, rebuild capability
Fault Tolerance	Cassandra replication	server/db/cassandra.js	Replication factor, multi-datacenter