How LinkedIn Works

> ***Target Audience****: Fullstack engineers (TS/Go, PostgreSQL, Redis)*
> ***Focus****: Production architecture, not interview soundbites*
> ***Scale Context****: ~900M users, ~58M companies, ~15M job posts*

---

1. Problem Definition (What This System Must Achieve)

LinkedIn is a professional networking platform that must solve:

Core functional requirements:

- User profiles with work history, skills, endorsements
- Bidirectional connection requests (not follow model like Twitter)
- News feed showing posts from connections + promoted content
- Job search and recommendations
- Messaging (real-time, persistent)
- Company pages and updates
- Content publishing (articles, posts, comments, reactions)

Non-functional requirements:

- **Latency**: Feed load < 500ms (p95), messaging < 200ms
- **Scale**: 900M users, ~300M MAU, ~60M DAU
- **Consistency**: Connections must be strongly consistent (no phantom connections)
- **Availability**: 99.9%+ uptime (standard SaaS target)
- **Data integrity**: Work history, connections, job applications are critical business data

What makes this hard:

- **Feed personalization** at scale (not just chronological)
- **Graph operations** (2nd/3rd degree connections, mutual connections)
- **Real-time messaging** + persistent storage
- **Job matching** (skills, location, experience, salary expectations)
- **Abuse prevention** (fake profiles, spam, scrapers)
- **B2B features** (recruiter tools, company analytics, advertising)

Unlike Twitter (public content), LinkedIn connections create a **permission boundary** — you can only see content from your network. This makes every feed query a graph traversal problem.

---

2. Naive Design (And Why It Fails)

The Simple Version

```
┌─────────┐
│ Node.js │
│  Server │
└────┬────┘
     │
     ▼
┌─────────┐
│Postgres │
```

```
│   (all)   │
└───────────┘
```

Schema:

```
// Single database
users: { id, name, email, headline, location, ... }
connections: { user_id, connected_user_id, status: 'pending' | 'accepted' }
posts: { id, author_id, content, created_at }
feed: computed on-demand by querying posts from connections
```

Feed generation (naive):

```
async function getFeed(userId: string) {
  // Get all connections
  const connections = await db.query(
    'SELECT connected_user_id FROM connections WHERE user_id = $1 AND status = $2',
    [userId, 'accepted']
  );

  // Get posts from all connections
  const posts = await db.query(
    'SELECT * FROM posts WHERE author_id = ANY($1) ORDER BY created_at DESC LIMIT 
20',
    [connections.map(c => c.connected_user_id)]
  );

  return posts;
}
```

Why This Breaks

1. Feed latency explodes:

- Average user has 500 connections
- Each feed load queries 500 users' posts
- At 1M DAU, that's 500M post scans per day
- With likes/comments, this becomes a join nightmare
- **Result**: p99 latency > 5 seconds

2. Database becomes a bottleneck:

- Connections are read-heavy (every feed load)
- Posts are write-heavy (users publish constantly)
- Same database handles profiles, messages, jobs, events
- **Result**: Lock contention, query queue buildup

3. Personalization is missing:

- No ranking (just chronological)
- No filtering (see all connection spam)
- No algorithmic boost for "important" content

- **Result**: Users see low-quality feed, engagement drops

4. Graph queries kill performance:

- "People You May Know" requires 2nd-degree traversal
- "X mutual connections" is a join on connections table
- At scale, these become $O(N^2)$ operations
- **Result**: These features simply timeout

5. Messaging breaks the model:

- Storing every message in Postgres is expensive
- Real-time delivery requires websockets + pub/sub
- Message search across 100K messages per user is slow
- **Result**: Messaging is either broken or murders the DB

6. Abuse vectors:

- No rate limiting → scraping epidemic
- No connection throttling → spam bots
- No content filtering → adult/violent content spreads
- **Result**: Platform unusability

---

3. High-Level Architecture

Component Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                       CDN (Cloudflare)                            │
│                (profile pics, media, static assets)                │
└─────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                    API Gateway (Node.js)                          │
│          (Auth, rate limiting, request routing)                   │
└─────────────────────────────────────────────────────────────────┘
         │              │                │               │
         │              │                │               │
         ▼              ▼                ▼               ▼
┌──────────┐   ┌──────────┐   ┌──────────────┐   ┌──────────┐
│ Profile  │   │  Feed    │   │  Messaging   │   │  Jobs    │
│ Service  │   │ Service  │   │   Service    │   │ Service  │
│ (Node)   │   │(Node+Go) │   │    (Go)      │   │ (Node)   │
└──────────┘   └──────────┘   └──────────────┘   └──────────┘
     │              │                │                 │
     │              │                │                 │
     ▼              ▼                ▼                 ▼
┌─────────┐  ┌─────────┐     ┌─────────┐      ┌─────────┐
│ Postgres│  │ Postgres│     │Postgres │      │ Postgres│
│ (Users) │  │ (Posts) │     │ (Msgs)  │      │ (Jobs)  │
└─────────┘  └─────────┘     └─────────┘      └─────────┘
     │              │                │                 │
     ▼              ▼                ▼                 ▼
┌─────────────────────────────────────────────────────────────┐
```

```
│                    Redis Cluster                           │
│  (Session cache, feed cache, connection graph cache)       │
└───────────────────────────────────────────────────────────┘
                              │
                              ▼
                    ┌───────────────────┐
                    │  Kafka / Queues   │
                    │  (Async tasks)    │
                    └───────────────────┘
                              │
                    ┌─────────┴─────────┐
                    ▼                   ▼
            ┌──────────────┐    ┌──────────────┐
            │Feed Builder  │    │ Notification │
            │  Worker (Go) │    │Worker (Node) │
            └──────────────┘    └──────────────┘
```

Service Boundaries

Profile Service (Node.js):

- CRUD operations on user profiles
- Connection requests (create, accept, reject)
- Connection graph queries (mutual connections, 2nd degree)
- Privacy settings

Feed Service (Node.js + Go workers):

- API layer (Node.js): Serve pre-built feeds from cache
- Worker layer (Go): Background feed materialization
- Post creation, likes, comments (write path)
- Feed ranking and filtering

Messaging Service (Go):

- WebSocket gateway for real-time delivery
- Message persistence
- Typing indicators, read receipts
- Message search

Jobs Service (Node.js):

- Job postings CRUD
- Job search and filtering
- Job recommendations
- Application tracking

Why Go for some services:

- Feed workers: High concurrency, CPU-bound ranking
- Messaging: Long-lived WebSocket connections (1M+ concurrent)
- Go's goroutines handle this better than Node's event loop

---

4. Core Data Model

PostgreSQL Schema (Sharded)

Users Database (sharded by user_id):

```
// users table
interface User {
  id: string; // UUID
  email: string; // unique
  password_hash: string;
  name: string;
  headline: string;
  location: string;
  profile_picture_url: string;
  created_at: timestamp;
  updated_at: timestamp;
}
// Index: (email), (created_at)

// connections table (core asset)
interface Connection {
  user_id: string;
  connected_user_id: string;
  status: 'pending' | 'accepted' | 'rejected';
  created_at: timestamp;
  updated_at: timestamp;
}
// Composite PK: (user_id, connected_user_id)
// Index: (user_id, status), (connected_user_id, status)
// Index: (created_at) for analytics
```

Posts Database (sharded by author_id):

```
interface Post {
  id: string; // UUID
  author_id: string;
  content: string; // max 3000 chars
  media_urls: string[]; // S3 keys
  visibility: 'public' | 'connections' | 'private';
  created_at: timestamp;
  updated_at: timestamp;

  // Denormalized counters (eventually consistent)
  like_count: number;
  comment_count: number;
  share_count: number;
}
// Index: (author_id, created_at)
// Index: (created_at) for global trending

interface PostEngagement {
  post_id: string;
```

```
  user_id: string;
  type: 'like' | 'comment' | 'share';
  created_at: timestamp;
}
// Composite PK: (post_id, user_id, type)
// Index: (user_id, created_at) for activity history
```

Messages Database (sharded by conversation_id):

```
interface Conversation {
  id: string; // UUID
  participant_ids: string[]; // sorted array [user1_id, user2_id]
  created_at: timestamp;
  updated_at: timestamp;
}
// Index: (participant_ids) using GIN for array search

interface Message {
  id: string; // UUID
  conversation_id: string;
  sender_id: string;
  content: string;
  created_at: timestamp;
  read_at: timestamp | null;
}
// Index: (conversation_id, created_at)
// Index: (sender_id, created_at)
```

Jobs Database (sharded by company_id):

```
interface Job {
  id: string;
  company_id: string;
  title: string;
  description: string;
  location: string;
  remote: boolean;
  salary_min: number | null;
  salary_max: number | null;
  experience_level: 'entry' | 'mid' | 'senior' | 'lead';
  created_at: timestamp;
  expires_at: timestamp;
}
// Index: (company_id, created_at)
// Index: (location, experience_level, created_at) for search
```

Redis Cache Structure

```
// Session cache (TTL: 7 days)
`session:${userId}` → { userId, email, name, profilePicture }
```

```
// Connection graph cache (TTL: 1 hour)
`connections:${userId}` → Set<userId> (all accepted connections)

// Feed cache (TTL: 10 minutes)
`feed:${userId}` → JSON array of post IDs with scores

// Mutual connections cache (TTL: 1 hour)
`mutual:${userId}:${otherUserId}` → Set<userId>

// Rate limiting
`rate:${userId}:${action}` → counter (TTL: varies)
```

Consistency Guarantees

Strongly consistent:

- Connection status (must be transactional)
- Job applications
- User credentials

Eventually consistent:

- Post like counts (tolerate brief stale reads)
- Feed updates (10-minute lag acceptable)
- Profile view counts

Immutable:

- Messages (never edited, only deleted via soft delete)
- Connection history (for audit)

---

5. Core Workflows

Workflow 1: User Creates a Post

Step-by-step:

1. **Client** sends POST request to API Gateway

   ```
   POST /api/v1/posts
   {
     "content": "Just shipped a new feature!",
     "media": ["image1.jpg"],
     "visibility": "connections"
   }
   ```

2. **API Gateway** validates auth token (checks Redis session cache)

   - If invalid → 401 Unauthorized
   - If valid → extract userId, forward to Feed Service

3. **Feed Service** (Node.js):

```
async function createPost(userId: string, data: PostData) {
  // Upload media to S3 first
  const mediaUrls = await uploadMedia(data.media);

  // Insert post into Postgres
  const post = await db.transaction(async (tx) => {
    const result = await tx.query(
      'INSERT INTO posts (id, author_id, content, media_urls, visibility,
created_at) VALUES ($1, $2, $3, $4, $5, NOW()) RETURNING *',
      [generateUUID(), userId, data.content, mediaUrls, data.visibility]
    );
    return result.rows[0];
  });

  // Publish event to Kafka
  await kafka.publish('post.created', {
    postId: post.id,
    authorId: userId,
    visibility: data.visibility,
    createdAt: post.created_at
  });

  return post;
}
```

4. **Kafka Consumer** (Go worker) picks up `post.created` event

```
func handlePostCreated(event PostCreatedEvent) {
    // Get author's connections from cache/DB
    connections := getConnections(event.AuthorId)

    // Fan-out: Add post to each connection's feed cache
    for _, connId := range connections {
        rdb.ZAdd(ctx, fmt.Sprintf("feed:%s", connId), &redis.Z{
            Score:  float64(event.CreatedAt.Unix()),
            Member: event.PostId,
        })

        // Trim to keep only top 1000 posts
        rdb.ZRemRangeByRank(ctx, fmt.Sprintf("feed:%s", connId), 0, -1001)
    }
}
```

5. **Client** receives response with post ID
    - Optimistic UI update (show post immediately)
    - If error, rollback UI change

Failure handling:

- If S3 upload fails → retry 3x, then fail request

- If DB insert fails → rollback, return error
- If Kafka publish fails → log error, post still created (feed will be stale)
- If feed fanout fails → retry via dead letter queue

Workflow 2: User Loads Feed

Step-by-step:

1. **Client** sends GET request

   ```
   GET /api/v1/feed?limit=20&offset=0
   ```

2. **API Gateway** validates session, forwards to Feed Service

3. **Feed Service** checks Redis cache first:

   ```
   async function getFeed(userId: string, limit: number, offset: number) {
     // Try cache first (hot path)
     const cachedPostIds = await redis.zrevrange(
       `feed:${userId}`,
       offset,
       offset + limit - 1,
       'WITHSCORES'
     );

     if (cachedPostIds.length > 0) {
       // Hydrate posts from Postgres
       const posts = await db.query(
         'SELECT * FROM posts WHERE id = ANY($1)',
         [cachedPostIds.map(p => p.value)]
       );

       // Enrich with engagement data
       const enrichedPosts = await enrichWithEngagement(posts, userId);
       return enrichedPosts;
     }

     // Cache miss: Build feed on-demand (cold path)
     return await buildFeedFromScratch(userId, limit, offset);
   }

   async function buildFeedFromScratch(userId: string, limit: number, offset:
   number) {
     // Get connections
     const connections = await getConnections(userId);

     // Get recent posts from connections
     const posts = await db.query(
       `SELECT * FROM posts
        WHERE author_id = ANY($1)
        AND created_at > NOW() - INTERVAL '7 days'
        ORDER BY created_at DESC
   ```

```
     LIMIT $2 OFFSET $3`,
    [connections, limit, offset]
  );

  // Cache the result
  const postScores = posts.map(p => ({
    score: p.created_at.getTime(),
    member: p.id
  }));
  await redis.zadd(`feed:${userId}`, ...postScores);
  await redis.expire(`feed:${userId}`, 600); // 10 min TTL

  return posts;
}
```

4. **Enrichment step** (parallel):

```
async function enrichWithEngagement(posts: Post[], viewerId: string) {
  // Parallel queries
  const [likes, comments, userEngagement] = await Promise.all([
    // Get total engagement counts (cached)
    redis.mget(posts.map(p => `post:${p.id}:likes`)),
    redis.mget(posts.map(p => `post:${p.id}:comments`)),

    // Check if viewer has liked/commented
    db.query(
      'SELECT post_id, type FROM post_engagement WHERE user_id = $1 AND
post_id = ANY($2)',
      [viewerId, posts.map(p => p.id)]
    )
  ]);

  return posts.map((post, i) => ({
    ...post,
    likeCount: parseInt(likes[i]) || post.like_count,
    commentCount: parseInt(comments[i]) || post.comment_count,
    viewerHasLiked: userEngagement.some(e => e.post_id === post.id && e.type
=== 'like')
  }));
}
```

5. **Client** renders feed

Performance targets:

- Cache hit (hot path): 50-100ms
- Cache miss (cold path): 300-500ms
- Hydration queries: < 50ms each

Workflow 3: User Sends a Message

Step-by-step:

1. **Client** establishes WebSocket connection to Messaging Service

```
// Client-side (React)
const ws = new WebSocket('wss://messaging.linkedin.com/ws');

ws.onopen = () => {
  // Authenticate
  ws.send(JSON.stringify({ type: 'auth', token: accessToken }));
};

// Send message
ws.send(JSON.stringify({
  type: 'message',
  to: recipientUserId,
  content: 'Hey, are you open to new opportunities?'
}));
```

2. **Messaging Service** (Go) handles WebSocket message:

```
type MessageService struct {
    // In-memory map: userId -> WebSocket connection
    connections sync.Map
    db          *sql.DB
    redis       *redis.Client
}

func (s *MessageService) HandleMessage(userId string, msg IncomingMessage)
error {
    // 1. Validate recipient exists and is connected
    recipientConn, hasConnection := s.checkConnection(userId, msg.To)
    if !hasConnection {
        return errors.New("not connected to recipient")
    }

    // 2. Persist message to Postgres
    msgId := uuid.New().String()
    conversationId := getOrCreateConversation(userId, msg.To)

    _, err := s.db.Exec(
        "INSERT INTO messages (id, conversation_id, sender_id, content,
created_at) VALUES ($1, $2, $3, $4, NOW())",
        msgId, conversationId, userId, msg.Content,
    )
    if err != nil {
        return err
    }

    // 3. Try to deliver in real-time
    if recipientWs, ok := s.connections.Load(msg.To); ok {
        recipientWs.(*WebSocket).Send(OutgoingMessage{
            Id:             msgId,
```

```
            ConversationId: conversationId,
            From:           userId,
            Content:        msg.Content,
            CreatedAt:      time.Now(),
        })
    } else {
        // Recipient offline: Queue push notification
        s.queuePushNotification(msg.To, userId, msg.Content)
    }

    return nil
}
```

3. **Recipient** receives message via WebSocket (if online)
   - Client updates conversation UI immediately
   - Sends read receipt back
4. **If recipient offline**:
   - Message stored in Postgres
   - Push notification sent to mobile device
   - On next login, client fetches unread messages

Failure handling:

- If Postgres write fails → retry 3x, then show error to sender
- If WebSocket delivery fails → message is already persisted, recipient gets it on next poll
- If connection drops mid-send → client retries with idempotency key

---

Workflow 4: Connection Request (Accept/Reject)

Step-by-step:

1. **User A** sends connection request to **User B**

```
POST /api/v1/connections/request
{ "userId": "user-b-id" }
```

2. **Profile Service** creates pending connection:

```
async function sendConnectionRequest(fromUserId: string, toUserId: string) {
  // Check if already connected
  const existing = await db.query(
    'SELECT * FROM connections WHERE (user_id = $1 AND connected_user_id = $2)
OR (user_id = $2 AND connected_user_id = $1)',
    [fromUserId, toUserId]
  );

  if (existing.rows.length > 0) {
    throw new Error('Connection already exists or pending');
  }
```

```
  // Create pending connection (bidirectional)
  await db.transaction(async (tx) => {
    await tx.query(
      'INSERT INTO connections (user_id, connected_user_id, status,
created_at) VALUES ($1, $2, $3, NOW())',
      [fromUserId, toUserId, 'pending']
    );
    await tx.query(
      'INSERT INTO connections (user_id, connected_user_id, status,
created_at) VALUES ($1, $2, $3, NOW())',
      [toUserId, fromUserId, 'pending']
    );
  });

  // Send notification
  await notificationService.send(toUserId, {
    type: 'connection_request',
    from: fromUserId
  });
}
```

3. **User B** accepts the request

```
POST /api/v1/connections/accept
{ "userId": "user-a-id" }
```

4. **Profile Service** updates status (strongly consistent):

```
async function acceptConnection(userId: string, requesterId: string) {
  await db.transaction(async (tx) => {
    // Update both sides to 'accepted' (MUST be atomic)
    await tx.query(
      'UPDATE connections SET status = $1, updated_at = NOW() WHERE user_id =
$2 AND connected_user_id = $3',
      ['accepted', userId, requesterId]
    );
    await tx.query(
      'UPDATE connections SET status = $1, updated_at = NOW() WHERE user_id =
$2 AND connected_user_id = $3',
      ['accepted', requesterId, userId]
    );
  });

  // Invalidate connection cache for both users
  await Promise.all([
    redis.del(`connections:${userId}`),
    redis.del(`connections:${requesterId}`)
  ]);

  // Update graph cache asynchronously
  await kafka.publish('connection.accepted', {
```

```
    user1: userId,
    user2: requesterId
  });
}
```

5. **Graph update worker** refreshes mutual connections, 2nd-degree suggestions

Why strongly consistent:

- Connection state must never be inconsistent (A connected to B, but B not to A)
- Transaction ensures atomic update
- Cache invalidation happens after DB commit

---

6. Core System Flows (Async Processing)

Background Job: Feed Materialization

Why needed:

- Pre-computing feeds reduces latency
- Active users get fresh feeds without cold path

Implementation (Go worker):

```go
func feedMaterializationWorker(userId string) {
    // Runs every 10 minutes for active users

    // Get user's connections
    connections := getConnections(userId)

    // Get posts from last 7 days
    posts := db.Query(`
        SELECT id, author_id, created_at, like_count, comment_count
        FROM posts
        WHERE author_id = ANY($1)
        AND created_at > NOW() - INTERVAL '7 days'
        ORDER BY created_at DESC
        LIMIT 1000
    `, connections)

    // Apply ranking algorithm
    rankedPosts := rankPosts(posts, userId)

    // Store in Redis sorted set
    for _, post := range rankedPosts {
        rdb.ZAdd(ctx, fmt.Sprintf("feed:%s", userId), &redis.Z{
            Score:  post.Score,
            Member: post.Id,
        })
    }

    rdb.Expire(ctx, fmt.Sprintf("feed:%s", userId), 10*time.Minute)
```

```go
}

// Simple ranking: recency + engagement
func rankPosts(posts []Post, viewerId string) []RankedPost {
    ranked := make([]RankedPost, len(posts))

    for i, post := range posts {
        // Base score: timestamp (newer = higher)
        score := float64(post.CreatedAt.Unix())

        // Boost for engagement
        score += float64(post.LikeCount) * 100
        score += float64(post.CommentCount) * 200

        // Boost if author is frequently engaged with
        engagementFreq := getEngagementFrequency(viewerId, post.AuthorId)
        score += engagementFreq * 10000

        ranked[i] = RankedPost{
            Post:  post,
            Score: score,
        }
    }

    sort.Slice(ranked, func(i, j int) bool {
        return ranked[i].Score > ranked[j].Score
    })

    return ranked
}
```

Triggering:

- Scheduled cron job for DAU users (10-min intervals)
- On-demand for MAU users (on login)

---

7. API Design

REST Endpoints

Authentication:

```
POST /api/v1/auth/login
POST /api/v1/auth/logout
POST /api/v1/auth/refresh
```

Profile:

```
GET    /api/v1/users/:userId
PATCH  /api/v1/users/:userId
GET    /api/v1/users/:userId/connections
```

```
POST   /api/v1/connections/request
POST   /api/v1/connections/accept
POST   /api/v1/connections/reject
```

Feed:

```
GET    /api/v1/feed?limit=20&offset=0
POST   /api/v1/posts
GET    /api/v1/posts/:postId
DELETE /api/v1/posts/:postId
POST   /api/v1/posts/:postId/like
DELETE /api/v1/posts/:postId/like
POST   /api/v1/posts/:postId/comments
```

Messaging:

```
GET    /api/v1/conversations
GET    /api/v1/conversations/:conversationId/messages
WebSocket: wss://messaging.linkedin.com/ws
```

Jobs:

```
GET    /api/v1/jobs?location=&experienceLevel=&remote=
GET    /api/v1/jobs/:jobId
POST   /api/v1/jobs/:jobId/apply
```

Idempotency

All write operations use idempotency keys:

```
POST /api/v1/posts
Headers: {
  "Idempotency-Key": "uuid-generated-by-client"
}
```

Implementation:

```
async function createPost(userId: string, data: PostData, idempotencyKey: string) {
  // Check if already processed
  const cached = await redis.get(`idempotency:${idempotencyKey}`);
  if (cached) {
    return JSON.parse(cached); // Return cached result
  }

  // Process request
  const result = await actuallyCreatePost(userId, data);

  // Cache result for 24 hours
  await redis.setex(`idempotency:${idempotencyKey}`, 86400, JSON.stringify(result));
```

```
  return result;
}
```

---

8. Consistency, Ordering & Concurrency

Consistency Model by Feature

Strong consistency (ACID transactions):

- Connection state changes (accept/reject)
- Job applications
- Payment transactions (recruiter subscriptions)

Sequential consistency:

- Messages within a conversation (total order per conversation)
- Comments on a post (order must be stable)

Eventually consistent:

- Feed content
- Like/comment counters
- Profile view counts
- "Who viewed your profile" analytics

Connection Acceptance Race Condition

Problem:

```
Time  User A (accepts B)          User B (accepts A)
t0    Read: status = 'pending'    Read: status = 'pending'
t1    Write: status = 'accepted'  Write: status = 'accepted'
t2    ✓ Both think they accepted
```

Solution: Optimistic locking

```
async function acceptConnection(userId: string, requesterId: string) {
  const result = await db.transaction(async (tx) => {
    // Lock both rows with SELECT FOR UPDATE
    const rows = await tx.query(
      `SELECT * FROM connections
       WHERE (user_id = $1 AND connected_user_id = $2)
       OR (user_id = $2 AND connected_user_id = $1)
       FOR UPDATE`,
      [userId, requesterId]
    );

    if (rows.length !== 2) {
      throw new Error('Connection not found');
    }

    // Check current status
```

```
    const currentStatus = rows[0].status;
    if (currentStatus === 'accepted') {
      return { alreadyAccepted: true };
    }

    if (currentStatus !== 'pending') {
      throw new Error('Invalid state transition');
    }

    // Update both rows
    await tx.query(
      `UPDATE connections
       SET status = 'accepted', updated_at = NOW()
       WHERE (user_id = $1 AND connected_user_id = $2)
       OR (user_id = $2 AND connected_user_id = $1)`,
      [userId, requesterId]
    );

    return { success: true };
  });

  return result;
}
```

Post Engagement Counters

Problem:

- 1M users liking a viral post = 1M DB writes
- Contention on single row (post.like_count)

Solution: Write-behind caching

```
// Immediate response (optimistic)
async function likePost(userId: string, postId: string) {
  // Check if already liked (idempotency)
  const existing = await redis.sismember(`post:${postId}:likers`, userId);
  if (existing) return { alreadyLiked: true };

  // Add to Redis set (fast)
  await redis.sadd(`post:${postId}:likers`, userId);

  // Increment counter in Redis
  const newCount = await redis.incr(`post:${postId}:likes`);

  // Queue DB write (async)
  await kafka.publish('post.liked', { postId, userId });

  return { likeCount: newCount };
}

// Background worker flushes to Postgres
```

```
async function flushLikesToDB() {
  // Runs every 30 seconds
  const batch = await kafka.consumeBatch('post.liked', 1000);

  for (const event of batch) {
    await db.query(
      'INSERT INTO post_engagement (post_id, user_id, type, created_at) VALUES ($1,
$2, $3, NOW()) ON CONFLICT DO NOTHING',
      [event.postId, event.userId, 'like']
    );

    // Update denormalized counter (eventually consistent)
    await db.query(
      'UPDATE posts SET like_count = (SELECT COUNT(*) FROM post_engagement WHERE
post_id = $1 AND type = $2) WHERE id = $1',
      [event.postId, 'like']
    );
  }
}
```

Trade-off:

- Redis can lose data (not durable)
- But likes are not critical data
- Acceptable to lose <0.1% of likes in Redis crash

Message Ordering Guarantees

Requirement:

- Messages in a conversation must appear in send order
- Even if delivered out-of-order (network delays)

Solution: Lamport timestamps

```
type Message struct {
    Id             string
    ConversationId string
    SenderId       string
    Content        string
    CreatedAt      time.Time
    SequenceNum    int64 // Monotonically increasing per conversation
}

func (s *MessageService) SendMessage(senderId, recipientId, content string)
(*Message, error) {
    conversationId := getConversationId(senderId, recipientId)

    // Get next sequence number (atomic)
    seqNum, err := s.redis.Incr(ctx, fmt.Sprintf("conv:%s:seq",
conversationId)).Result()
    if err != nil {
        return nil, err
```

```
    }

    msg := &Message{
        Id:             uuid.New().String(),
        ConversationId: conversationId,
        SenderId:       senderId,
        Content:        content,
        CreatedAt:      time.Now(),
        SequenceNum:    seqNum,
    }

    // Persist to DB
    _, err = s.db.Exec(
        "INSERT INTO messages (id, conversation_id, sender_id, content, created_at,
sequence_num) VALUES ($1, $2, $3, $4, $5, $6)",
        msg.Id, msg.ConversationId, msg.SenderId, msg.Content, msg.CreatedAt,
msg.SequenceNum,
    )

    return msg, err
}

// Client sorts by sequence_num
SELECT * FROM messages
WHERE conversation_id = $1
ORDER BY sequence_num ASC
```

---

9. Caching Strategy

Layers

```
┌──────────────────────────────────────┐
│  CDN (Cloudflare)                    │
│  - Profile pictures                  │
│  - Static assets (JS, CSS)           │
│  - Public company logos              │
│  TTL: 7 days                         │
└──────────────────────────────────────┘
              │
              ▼
┌──────────────────────────────────────┐
│  Redis (Application Cache)           │
│  - Session data                      │
│  - Feed cache                        │
│  - Connection graph                  │
│  - Mutual connections                │
│  - Hot post engagement               │
│  TTL: 10 min - 7 days                │
└──────────────────────────────────────┘
              │
```

```
                ▼
┌───────────────────────────────────────┐
│   Postgres (Source of Truth)          │
└───────────────────────────────────────┘
```

What Gets Cached

Session data (Redis):

```
Key: `session:${token}`
Value: { userId, email, name, profilePicture }
TTL: 7 days
Eviction: LRU

// On every request
async function validateSession(token: string) {
  const cached = await redis.get(`session:${token}`);
  if (cached) return JSON.parse(cached);

  // Cache miss: Load from DB
  const session = await db.query('SELECT * FROM sessions WHERE token = $1',
[token]);
  if (!session) return null;

  await redis.setex(`session:${token}`, 604800, JSON.stringify(session));
  return session;
}
```

Connection graph (Redis):

```
Key: `connections:${userId}`
Value: Set of connected user IDs
TTL: 1 hour
Invalidation: On connection accept/reject

async function getConnections(userId: string): Promise<string[]> {
  const cached = await redis.smembers(`connections:${userId}`);
  if (cached.length > 0) return cached;

  const connections = await db.query(
    'SELECT connected_user_id FROM connections WHERE user_id = $1 AND status = $2',
    [userId, 'accepted']
  );

  const userIds = connections.map(c => c.connected_user_id);
  if (userIds.length > 0) {
    await redis.sadd(`connections:${userId}`, ...userIds);
    await redis.expire(`connections:${userId}`, 3600);
  }
```

```
  return userIds;
}
```

Feed cache (Redis sorted set):

```
Key: `feed:${userId}`
Value: Sorted set of post IDs (score = ranking score)
TTL: 10 minutes
Invalidation: Time-based (no manual invalidation)

async function getCachedFeed(userId: string, limit: number, offset: number) {
  const postIds = await redis.zrevrange(
    `feed:${userId}`,
    offset,
    offset + limit - 1
  );

  if (postIds.length === 0) return null; // Cache miss

  // Hydrate from DB
  const posts = await db.query(
    'SELECT * FROM posts WHERE id = ANY($1)',
    [postIds]
  );

  return posts;
}
```

Cache Invalidation Patterns

1. Time-based (feed):

- No manual invalidation
- TTL = 10 minutes
- Acceptable staleness

2. Event-based (connections):

- Invalidate on write
- Ensures eventual consistency

```
async function acceptConnection(userId: string, requesterId: string) {
  await db.transaction(/* ... */);

  // Invalidate both users' connection cache
  await Promise.all([
    redis.del(`connections:${userId}`),
    redis.del(`connections:${requesterId}`),
    redis.del(`mutual:${userId}:*`), // Wildcard delete
    redis.del(`mutual:${requesterId}:*`)
```

```
  ]);
}
```

3. Write-through (sessions):

- Update cache and DB together
- Strong consistency for auth

Thundering Herd Prevention

Problem:

- Feed cache expires for 10K users at same time
- All hit DB simultaneously
- DB overload

Solution: Probabilistic early expiration

```
async function getCachedFeed(userId: string, limit: number) {
  const ttl = await redis.ttl(`feed:${userId}`);

  // Recompute early with probability inversely proportional to TTL
  // When TTL = 60s, 10% chance to recompute
  // When TTL = 10s, 50% chance to recompute
  const shouldRecomputeEarly = Math.random() < (1 - ttl / 600);

  if (shouldRecomputeEarly) {
    // Trigger async recomputation
    kafka.publish('feed.recompute', { userId });
  }

  const cached = await redis.zrevrange(`feed:${userId}`, 0, limit - 1);
  return cached.length > 0 ? cached : null;
}
```

---

10. Scaling Strategy

Horizontal Scaling

API Gateway (Node.js):

- Stateless servers (session in Redis)
- Load balanced via NGINX/ALB
- Auto-scale based on CPU (target 60%)
- Typically run 50-100 instances

Feed Service:

- Stateless API layer
- Horizontal scaling (100+ instances)
- Background workers (Go): 20-30 instances, CPU-heavy

Messaging Service (Go):

- Stateful (WebSocket connections)
- Sticky sessions via consistent hashing
- Each instance handles 50K concurrent connections
- 20 instances = 1M concurrent users

Database Sharding

Users + Connections (sharded by user_id):

```
Shard 1: user_id hash % 16 = 0
Shard 2: user_id hash % 16 = 1
...
Shard 16: user_id hash % 16 = 15
```

Routing logic:

```typescript
function getUserShard(userId: string): number {
  const hash = xxhash(userId);
  return hash % 16;
}

async function getUser(userId: string) {
  const shard = getUserShard(userId);
  const db = getDBConnection(`users_shard_${shard}`);
  return db.query('SELECT * FROM users WHERE id = $1', [userId]);
}
```

Cross-shard queries (mutual connections):

```typescript
// Problem: User A (shard 3) wants mutual connections with User B (shard 7)
async function getMutualConnections(userId1: string, userId2: string) {
  // Check cache first
  const cacheKey = `mutual:${userId1}:${userId2}`;
  const cached = await redis.smembers(cacheKey);
  if (cached.length > 0) return cached;

  // Fetch from both shards
  const [connections1, connections2] = await Promise.all([
    getConnections(userId1), // Hits shard 3
    getConnections(userId2)  // Hits shard 7
  ]);

  // Compute intersection in application layer
  const mutual = connections1.filter(id => connections2.includes(id));

  await redis.sadd(cacheKey, ...mutual);
  await redis.expire(cacheKey, 3600);

  return mutual;
}
```

Why sharding needed:

- 900M users → 1 DB can't handle write load
- Connections table → 50B+ rows (avg 500 connections/user)

Read Replicas

Setup:

```
┌──────────────┐
│   Primary    │ (writes)
└──────┬───────┘
       │
       ├─────────────┬─────────────┐
       ▼             ▼             ▼
  Replica 1     Replica 2     Replica 3
  (reads)       (reads)       (reads)
```

Read routing:

```
// Write operations → primary
await primaryDB.query('INSERT INTO posts ...');

// Read operations → replicas (round-robin)
await replicaDB.query('SELECT * FROM posts WHERE id = $1', [postId]);

// Strong consistency reads → primary
await primaryDB.query('SELECT * FROM connections WHERE user_id = $1 FOR UPDATE',
[userId]);
```

Replication lag handling:

- Typical lag: 100-500ms
- For feed/posts: Acceptable
- For connections: Not acceptable → read from primary

Hot Key Problem

Scenario: Viral post from Satya Nadella

- 1M likes in 10 minutes
- All requests hit same post cache key

Solution 1: Local in-memory cache

```
// LRU cache per API server instance
const localCache = new LRUCache({ max: 10000, ttl: 60000 });

async function getPost(postId: string) {
  // Check local cache first
  const local = localCache.get(postId);
  if (local) return local;

  // Check Redis
```

```
  const cached = await redis.get(`post:${postId}`);
  if (cached) {
    const post = JSON.parse(cached);
    localCache.set(postId, post);
    return post;
  }

  // Load from DB
  const post = await db.query('SELECT * FROM posts WHERE id = $1', [postId]);
  await redis.setex(`post:${postId}`, 300, JSON.stringify(post));
  localCache.set(postId, post);

  return post;
}
```

Solution 2: Replicate hot keys

```
// Write to multiple Redis keys
async function cacheHotPost(post: Post) {
  // Replicate to 10 keys
  const promises = [];
  for (let i = 0; i < 10; i++) {
    promises.push(redis.setex(`post:${post.id}:${i}`, 300, JSON.stringify(post)));
  }
  await Promise.all(promises);
}

// Read from random replica
async function getHotPost(postId: string) {
  const replica = Math.floor(Math.random() * 10);
  return redis.get(`post:${postId}:${replica}`);
}
```

---

11. Fault Tolerance & Reliability

Failure Modes

1. Database failure (shard goes down):

- **Impact**: All users on that shard can't read/write
- **Detection**: Health check fails (every 10s)
- **Mitigation**:
  - Promote read replica to primary (automatic failover via Patroni)
  - RPO: ~1 minute (replication lag)
  - RTO: ~2 minutes (detection + failover)
- **Prevention**: Run replicas in different AZs

2. Redis cluster failure:

- **Impact**:
  - Feed cache miss → high DB load

- Session cache miss → users logged out
- **Detection**: Connection timeout
- **Mitigation**:
  - Fall back to DB reads (slower but works)
  - Rate limit to prevent DB overload
  - Redis Cluster mode with 3 masters + 3 replicas
- **Degradation**: Feed latency 200ms → 2s

3. Kafka partition failure:

- **Impact**: Feed fanout stops, notifications delayed
- **Detection**: Consumer lag metric spikes
- **Mitigation**:
  - Other partitions continue working
  - Failed jobs go to dead letter queue
  - Manual replay after recovery
- **Prevention**: Replication factor = 3

4. Message delivery failure:

- **Scenario**: User sends message, WebSocket disconnects before ACK
- **Solution**: Client-side retry with idempotency

```
// Client generates message ID
const msgId = generateUUID();

function sendMessage(content: string) {
  ws.send(JSON.stringify({
    id: msgId, // Idempotency key
    type: 'message',
    content: content
  }));

  // Wait for ACK
  const ackTimeout = setTimeout(() => {
    // Retry if no ACK in 5s
    sendMessage(content);
  }, 5000);

  ws.on('ack', (ackMsgId) => {
    if (ackMsgId === msgId) {
      clearTimeout(ackTimeout);
    }
  });
}
```

5. Cascading failure (retry storm):

- **Scenario**: DB slow → API times out → clients retry → DB slower
- **Solution**: Circuit breaker

```
class CircuitBreaker {
  private state: 'closed' | 'open' | 'half-open' = 'closed';
  private failureCount = 0;
  private lastFailTime = 0;

  async execute<T>(fn: () => Promise<T>): Promise<T> {
    if (this.state === 'open') {
      // Check if should try again
      if (Date.now() - this.lastFailTime > 30000) {
        this.state = 'half-open';
      } else {
        throw new Error('Circuit breaker open');
      }
    }

    try {
      const result = await fn();

      if (this.state === 'half-open') {
        this.state = 'closed';
        this.failureCount = 0;
      }

      return result;
    } catch (err) {
      this.failureCount++;
      this.lastFailTime = Date.now();

      if (this.failureCount >= 5) {
        this.state = 'open';
      }

      throw err;
    }
  }
}

// Usage
const breaker = new CircuitBreaker();

async function getPost(postId: string) {
  return breaker.execute(() => db.query('SELECT * FROM posts WHERE id = $1',
[postId]));
}
```

Partial Failure Handling

Scenario: User creates post, fanout fails

```
async function createPost(userId: string, content: string) {
  let postId: string;
```

```
  try {
    // CRITICAL: Persist post first
    const result = await db.query(
      'INSERT INTO posts (id, author_id, content, created_at) VALUES ($1, $2, $3,
NOW()) RETURNING id',
      [generateUUID(), userId, content]
    );
    postId = result.rows[0].id;
  } catch (err) {
    // DB write failed → propagate error to client
    throw err;
  }

  try {
    // NON-CRITICAL: Fanout to feed (best effort)
    await kafka.publish('post.created', { postId, userId });
  } catch (err) {
    // Kafka failed → log but don't fail request
    logger.error('Feed fanout failed', { postId, error: err });

    // Enqueue for retry
    await retryQueue.add({ postId, userId, attempt: 1 });
  }

  return { postId };
}
```

Key principle:

- **Persist critical data first** (post exists in DB)
- **Background work is best-effort** (feed fanout can retry)
- **Never fail user request for non-critical work**

RPO/RTO Targets

Component	RPO	RTO	Strategy
User profiles	0	5 min	Synchronous replication
Connections	0	5 min	Synchronous replication
Posts	1 min	5 min	Async replication (acceptable to lose last minute)
Messages	0	2 min	Synchronous replication
Feed cache	N/A	10 min	Rebuild from scratch
Likes/comments	5 min	10 min	Write-behind cache

---

12. Observability & Operations

Metrics (Critical)

API Gateway:

```
// Request rate
api_requests_total{endpoint="/api/v1/feed", status="200"}
api_requests_total{endpoint="/api/v1/feed", status="500"}

// Latency (histogram)
api_request_duration_seconds{endpoint="/api/v1/feed", quantile="0.95"}

// Active connections
api_active_connections

// Rate limit violations
api_rate_limit_exceeded{user_id="...", endpoint="..."}
```

Database:

```
db_connections_active
db_connections_idle
db_query_duration_seconds{query="SELECT FROM posts"}
db_replication_lag_seconds
db_deadlock_count
db_lock_wait_time_seconds
```

Redis:

```
redis_hits_total
redis_misses_total
redis_memory_used_bytes
redis_evicted_keys_total
redis_connected_clients
```

Kafka:

```
kafka_consumer_lag{topic="post.created", partition="0"}
kafka_producer_errors_total
kafka_message_throughput
```

Logs (Structured JSON)

```
logger.info({
  event: 'post_created',
  userId: 'user-123',
  postId: 'post-456',
  latency_ms: 45,
  cache_hit: false,
  timestamp: Date.now()
});

logger.error({
  event: 'db_query_timeout',
```

```
  query: 'SELECT * FROM posts WHERE author_id = $1',
  params: ['user-123'],
  error: err.message,
  stack: err.stack,
  duration_ms: 5000,
  timestamp: Date.now()
});
```

Alerts (What Wakes You Up at 3 AM)

🔴 **Critical (page immediately):**

- API error rate > 5% for 5 minutes
- Database primary down
- Redis cluster down
- Message delivery failure rate > 10%
- Active connections dropped by 50% suddenly

🟡 **Warning (investigate next day):**

- API p95 latency > 1s for 10 minutes
- Kafka consumer lag > 100K messages
- Cache hit rate < 80%
- Database replication lag > 5 seconds

🟢 **Informational:**

- New deployment completed
- Auto-scaling triggered
- Slow query detected (> 1s)

Debugging Workflow ("My feed is broken!")

Step 1: Check user's feed cache

```
redis-cli> ZRANGE feed:user-123 0 20 WITHSCORES
```

Step 2: Check user's connections

```
redis-cli> SMEMBERS connections:user-123
```

Step 3: Check recent posts from connections

```
SELECT p.id, p.author_id, p.created_at, p.content
FROM posts p
WHERE p.author_id IN (
  SELECT connected_user_id FROM connections WHERE user_id = 'user-123' AND status =
'accepted'
)
AND p.created_at > NOW() - INTERVAL '7 days'
ORDER BY p.created_at DESC
LIMIT 20;
```

Step 4: Check feed fanout events

```
# Check Kafka consumer lag
kafka-consumer-groups --describe --group feed-builder

# Check dead letter queue
SELECT * FROM dlq_feed_fanout WHERE user_id = 'user-123' ORDER BY created_at DESC
LIMIT 10;
```

Step 5: Manually trigger feed rebuild

```
curl -X POST https://api.linkedin.com/internal/feed/rebuild \
  -H "X-Admin-Token: ..." \
  -d '{"userId": "user-123"}'
```

Distributed Tracing

```
// Using OpenTelemetry
const tracer = opentelemetry.trace.getTracer('linkedin-api');

async function getFeed(userId: string, limit: number) {
  const span = tracer.startSpan('get_feed', {
    attributes: { userId, limit }
  });

  try {
    // Trace cache lookup
    const cacheSpan = tracer.startSpan('redis_get_feed', { parent: span });
    const cached = await redis.zrevrange(`feed:${userId}`, 0, limit - 1);
    cacheSpan.end();

    if (cached.length === 0) {
      // Trace DB query
      const dbSpan = tracer.startSpan('db_build_feed', { parent: span });
      const posts = await buildFeedFromScratch(userId, limit);
      dbSpan.end();
      return posts;
    }

    // Trace hydration
    const hydrateSpan = tracer.startSpan('hydrate_posts', { parent: span });
    const posts = await hydratePosts(cached);
    hydrateSpan.end();

    return posts;
  } finally {
    span.end();
  }
}
```

Trace output:

```
get_feed (450ms)
├─ redis_get_feed (15ms) ✓
├─ hydrate_posts (280ms)
│  ├─ db_query_posts (120ms)
│  ├─ redis_get_engagement (80ms)
│  └─ enrich_metadata (80ms)
└─ [end]
```

---

13. Security & Abuse Prevention

Authentication Flow

Login:

```
POST /api/v1/auth/login
{
  "email": "user@example.com",
  "password": "********"
}

// Response
{
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "refreshToken": "refresh_abc123...",
  "expiresIn": 3600
}
```

Implementation:

```
async function login(email: string, password: string) {
  // 1. Rate limit login attempts
  const attempts = await redis.incr(`login_attempts:${email}`);
  if (attempts > 5) {
    await redis.expire(`login_attempts:${email}`, 900); // 15 min lockout
    throw new Error('Too many login attempts');
  }

  // 2. Fetch user
  const user = await db.query('SELECT * FROM users WHERE email = $1', [email]);
  if (!user) throw new Error('Invalid credentials');

  // 3. Verify password (bcrypt)
  const valid = await bcrypt.compare(password, user.password_hash);
  if (!valid) throw new Error('Invalid credentials');

  // 4. Generate tokens
  const accessToken = jwt.sign(
    { userId: user.id, email: user.email },
```

```
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
  );

  const refreshToken = generateSecureToken();

  // 5. Store session
  await db.query(
    'INSERT INTO sessions (user_id, refresh_token, created_at, expires_at) VALUES
($1, $2, NOW(), NOW() + INTERVAL \'30 days\')',
    [user.id, refreshToken]
  );

  await redis.setex(`session:${accessToken}`, 3600, JSON.stringify({
    userId: user.id,
    email: user.email
  }));

  return { accessToken, refreshToken, expiresIn: 3600 };
}
```

Authorization (Connection-based)

Problem: User A shouldn't see User B's feed unless they're connected

```
async function canViewFeed(viewerId: string, targetUserId: string): Promise<boolean>
{
  // Public profiles are visible to everyone
  const targetUser = await getUser(targetUserId);
  if (targetUser.profile_visibility === 'public') return true;

  // Check if connected
  const isConnected = await redis.sismember(`connections:${viewerId}`,
targetUserId);
  if (isConnected) return true;

  // Check DB if cache miss
  const connection = await db.query(
    'SELECT * FROM connections WHERE user_id = $1 AND connected_user_id = $2 AND
status = $3',
    [viewerId, targetUserId, 'accepted']
  );

  return connection.rows.length > 0;
}

// Enforce in API
async function getFeedForUser(viewerId: string, targetUserId: string) {
  const canView = await canViewFeed(viewerId, targetUserId);
  if (!canView) {
    throw new Error('Not authorized to view this feed');
```

```
  }

  return buildFeed(targetUserId);
}
```

Rate Limiting (Token Bucket)

```
async function rateLimitCheck(userId: string, action: string): Promise<boolean> {
  const key = `rate:${userId}:${action}`;

  // Different limits per action
  const limits = {
    'post_create': { maxTokens: 10, refillRate: 1, refillInterval: 60 },
    'connection_request': { maxTokens: 50, refillRate: 5, refillInterval: 60 },
    'message_send': { maxTokens: 100, refillRate: 10, refillInterval: 60 },
    'api_request': { maxTokens: 1000, refillRate: 100, refillInterval: 60 }
  };

  const limit = limits[action];

  // Token bucket algorithm
  const now = Date.now();
  const bucket = await redis.get(key);

  let tokens = limit.maxTokens;
  let lastRefill = now;

  if (bucket) {
    const data = JSON.parse(bucket);
    const elapsed = (now - data.lastRefill) / 1000;
    const refilled = Math.floor(elapsed / limit.refillInterval) * limit.refillRate;
    tokens = Math.min(data.tokens + refilled, limit.maxTokens);
    lastRefill = data.lastRefill + (Math.floor(elapsed / limit.refillInterval) *
limit.refillInterval * 1000);
  }

  if (tokens < 1) {
    return false; // Rate limited
  }

  tokens -= 1;
  await redis.setex(key, 3600, JSON.stringify({ tokens, lastRefill }));

  return true;
}

// Middleware
async function rateLimitMiddleware(req, res, next) {
  const allowed = await rateLimitCheck(req.userId, req.route.path);
  if (!allowed) {
    return res.status(429).json({ error: 'Rate limit exceeded' });
```

```
  }
  next();
}
```

Abuse Vectors & Mitigations

1. Fake profiles / bot accounts:

- **Detection**:
  - Phone/email verification required
  - CAPTCHA on signup
  - ML model: profile completeness, activity patterns
- **Mitigation**:
  - Shadowban suspicious accounts (limited visibility)
  - Manual review for flagged accounts

2. Spam connection requests:

- **Detection**:
  - > *100 requests/day → flag*
  - Low accept rate (<10%) → flag
- **Mitigation**:
  - Reduce daily limit to 20 for flagged accounts
  - Show CAPTCHA before each request

3. Scraping (data harvesting):

- **Detection**:
  - Unusually high profile view rate (>1000/day)
  - No human-like interactions (no pauses, perfect timing)
- **Mitigation**:
  - Rate limit profile views (100/day for free users)
  - Require login for full profile access
  - Serve fake/incomplete data to suspected scrapers

4. Spam posts / phishing links:

- **Detection**:
  - URL reputation check (Google Safe Browsing API)
  - Keyword filtering (cheap drugs, MLM schemes)
  - ML model for spammy content
- **Mitigation**:
  - Auto-delete obvious spam
  - Soft-delete borderline cases (manual review)
  - Suspend repeat offenders

5. Message spam:

- **Detection**:
  - Same message to 50+ people
  - High block/report rate

- **Mitigation**:
  - Rate limit: 100 messages/day to non-connections
  - Show warning before sending to non-connections
  - Temp ban after 10 reports

Data Privacy & GDPR

Right to deletion:

```
async function deleteUserData(userId: string) {
  // 1. Soft delete user profile
  await db.query('UPDATE users SET deleted_at = NOW(), email = NULL WHERE id = $1',
[userId]);

  // 2. Remove from all connections
  await db.query('DELETE FROM connections WHERE user_id = $1 OR connected_user_id =
$1', [userId]);

  // 3. Anonymize posts (don't delete to preserve conversations)
  await db.query('UPDATE posts SET author_id = NULL, content = \'[deleted]\' WHERE
author_id = $1', [userId]);

  // 4. Delete messages
  await db.query('DELETE FROM messages WHERE sender_id = $1', [userId]);

  // 5. Remove from all caches
  await redis.del(`session:*:${userId}`);
  await redis.del(`connections:${userId}`);
  await redis.del(`feed:${userId}`);

  // 6. Schedule async cleanup (S3 uploads, analytics, etc.)
  await kafka.publish('user.deleted', { userId });
}
```

--- END OF PASS 2 ---