How TikTok Works

> ***Target Audience**: Fullstack engineers (TS/Go, PostgreSQL, Redis)*
> ***Focus**: Video recommendations, CDN delivery, ML-driven personalization*
> ***Scale Context**: ~1B MAU, ~150M DAU, 1B+ videos uploaded, 2B+ daily views*

---

1. Problem Definition (What This System Must Achieve)

TikTok is a short-form video platform driven by algorithmic content discovery, not social graph.

Core functional requirements:

- Video upload and processing (15s-10min videos)
- Personalized "For You" feed (algorithmic, not chronological)
- Like, comment, share, follow
- Video effects, filters, sound library
- Creator analytics
- Search and hashtag discovery
- Direct messaging

Non-functional requirements:

- **Latency**: Video start < 200ms, feed scroll < 100ms
- **Scale**: 1B users, 150M DAU, 1B videos stored, 2B video views/day
- **Bandwidth**: Massive egress (video CDN costs dominate)
- **Cold start**: New users with zero watch history get good recommendations
- **Real-time**: "For You" feed updates based on immediate watch behavior
- **Video quality**: Adaptive bitrate, multi-resolution encoding
- **Global**: Low latency video delivery worldwide

What makes this hard:

- **Recommendation engine**: Must predict what video will hook you in 3 seconds
- **Video processing pipeline**: Transcode to 5+ resolutions, fast (< 30 seconds)
- **CDN costs**: 2B views/day × 5MB avg = 10 PB/day egress
- **Cold start problem**: No watch history → how to personalize?
- **Virality detection**: Identify trending videos within minutes
- **Copyright detection**: Audio fingerprinting at scale
- **Real-time learning**: User skips video at 2s → update model immediately

Unlike YouTube (search-driven, long-form), TikTok is **swipe-driven, algorithmic discovery** — the feed IS the product.

---

2. Naive Design (And Why It Fails)

The Simple Version

```
    ▼
┌──────────┐     ┌──────────┐
│Postgres  │     │    S3    │
│  (all)   │     │ (videos) │
└──────────┘     └──────────┘
```

Schema:

```
users: { id, username, bio, followers_count }
videos: { id, user_id, s3_url, title, created_at }
follows: { follower_id, followee_id }
feed: SELECT videos FROM followed users ORDER BY created_at DESC
```

Feed generation (naive):

```
async function getFeed(userId: string) {
  // Get users I follow
  const following = await db.query('SELECT followee_id FROM follows WHERE
follower_id = $1', [userId]);

  // Get their videos
  const videos = await db.query(
    'SELECT * FROM videos WHERE user_id = ANY($1) ORDER BY created_at DESC LIMIT
20',
    [following.map(f => f.followee_id)]
  );

  return videos;
}
```

Why This Breaks

1. TikTok's feed is NOT social graph-based:

- ❌ Naive design shows videos from followed users
- ✅ TikTok shows videos from **anyone** based on ML predictions
- **Result**: Core product broken — no discovery, just Instagram clone

2. Video serving is impossibly slow:

- User in Tokyo requests video from S3 in us-east-1
- 200ms+ latency to start video
- Buffering on every swipe
- **Result**: Terrible UX, users leave

3. Video storage costs explode:

- Storing only original 1080p video
- 1B videos × 20MB avg = 20 PB storage
- No adaptive bitrate → mobile users waste bandwidth
- **Result**: $500K+/month S3 bill

4. Recommendation engine doesn't exist:

- Showing chronological feed is boring
- No personalization
- No virality detection
- **Result**: Users see random/bad content, don't return

5. Video processing is blocking:

- User uploads video → server processes it synchronously
- Transcoding takes 2-3 minutes
- User waits, then gives up
- **Result**: Upload abandonment rate > 80%

6. No abuse prevention:

- Pornographic content spreads
- Copyright violations (music theft)
- Underage users
- **Result**: Platform liability, legal shutdown

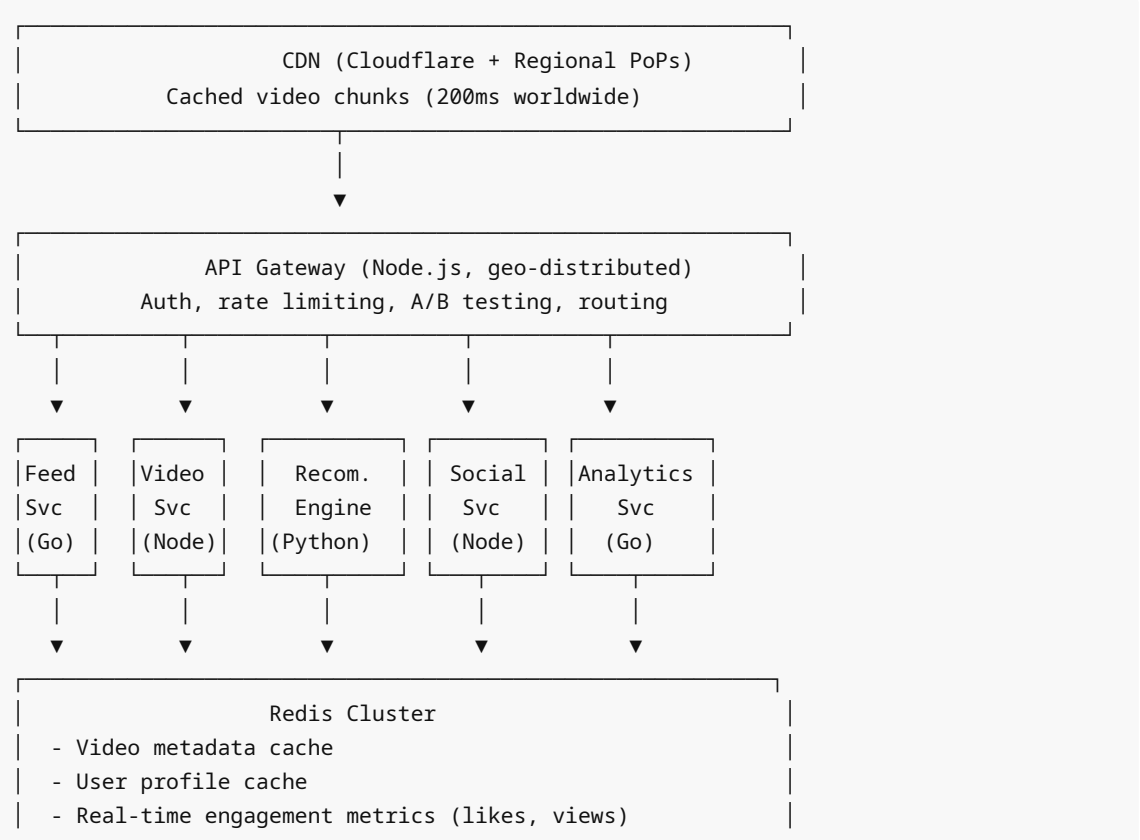
7. Database is wrong tool:

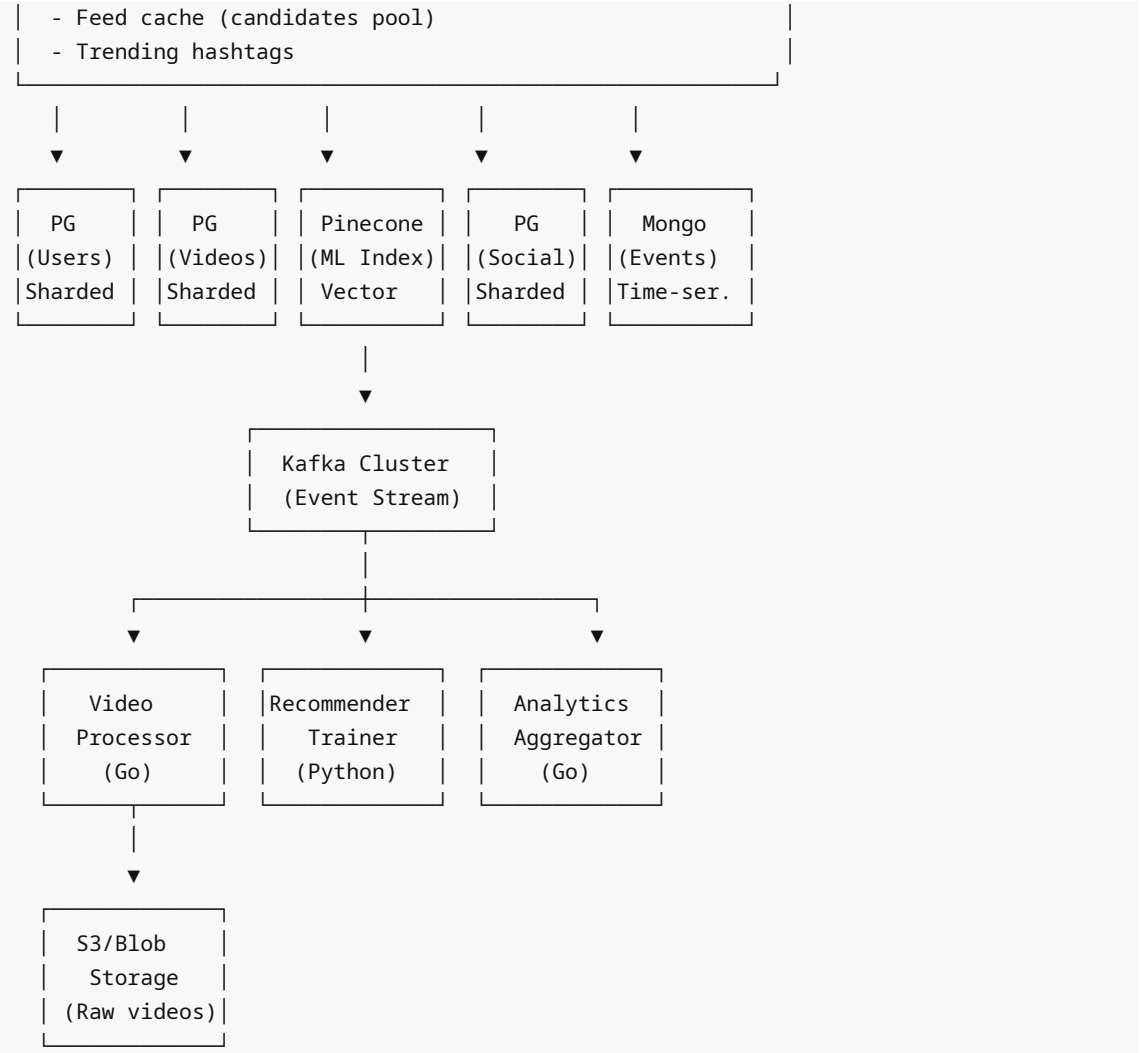
- "Get 20 random videos with watch_count < 1000" (show new content)
- Full table scan on 1B rows
- **Result**: Query timeout

---

3. High-Level Architecture

Component Overview





Service Boundaries

Feed Service (Go):

- Fetch personalized "For You" feed
- Call recommendation engine for candidate videos
- Rank and filter results
- Pre-fetch next videos for smooth scrolling
- Go chosen for: High throughput, efficient JSON parsing

Video Service (Node.js):

- Video upload (multipart, resumable)
- Video metadata CRUD
- Engagement endpoints (like, comment, share)
- Creator dashboard

Recommendation Engine (Python):

- ML model serving (TensorFlow/PyTorch)
- Generate candidate videos (retrieval stage)
- Ranking model (predict watch time)

- A/B testing framework
- Python chosen for: ML library ecosystem

Video Processor (Go):

- Background worker for video transcoding
- FFmpeg wrapper
- Generate thumbnails
- Audio extraction for copyright matching
- Go chosen for: CPU-intensive work, FFmpeg integration

Social Service (Node.js):

- Follow/unfollow
- Comments, replies
- Direct messages
- Notifications

Analytics Service (Go):

- Real-time view counting
- Creator metrics aggregation
- Trending detection
- Go chosen for: High-throughput event processing

---

4. Core Data Model

PostgreSQL Schema (Sharded)

Users Database (sharded by user_id):

```
interface User {
  id: string; // Snowflake ID
  username: string; // unique
  display_name: string;
  bio: string;
  avatar_url: string;
  verified: boolean;
  created_at: timestamp;

  // Denormalized counters
  followers_count: number;
  following_count: number;
  likes_count: number;
  videos_count: number;
}
// Index: (username), (created_at)

interface Follow {
  follower_id: string;
  followee_id: string;
  created_at: timestamp;
}
```

```
// Composite PK: (follower_id, followee_id)
// Index: (follower_id), (followee_id)
```

Videos Database (sharded by video_id):

```
interface Video {
  id: string; // Snowflake ID
  user_id: string;

  // Content
  title: string;
  description: string;
  hashtags: string[];
  music_id: string | null;

  // Processing status
  status: 'processing' | 'ready' | 'failed';

  // Video files (CDN URLs)
  original_url: string; // S3 key
  transcoded_urls: {
    '360p': string;
    '480p': string;
    '720p': string;
    '1080p': string;
  };
  thumbnail_url: string;
  duration_ms: number;

  // Engagement (eventually consistent)
  view_count: number;
  like_count: number;
  comment_count: number;
  share_count: number;

  // Privacy
  visibility: 'public' | 'followers' | 'private';

  created_at: timestamp;
  published_at: timestamp | null;
}
// Index: (user_id, created_at)
// Index: (created_at) for trending
// Index: (status) for processing queue
// GIN Index: (hashtags) for hashtag search

interface VideoEngagement {
  video_id: string;
  user_id: string;
  type: 'like' | 'share' | 'watch';
```

```
  // Watch metadata
  watch_duration_ms: number | null; // How long did user watch?
  watched_to_end: boolean;
  replayed: boolean;

  created_at: timestamp;
}
// Composite PK: (video_id, user_id, type)
// Index: (user_id, created_at) for user activity
// Index: (video_id, type, created_at) for video analytics
```

Music Library:

```
interface Music {
  id: string;
  title: string;
  artist: string;
  audio_url: string; // CDN URL
  duration_ms: number;

  // Copyright
  is_copyrighted: boolean;
  license_type: 'free' | 'licensed' | 'restricted';

  // Audio fingerprint (for copyright detection)
  fingerprint: string; // Chromaprint/AcoustID

  created_at: timestamp;
}
// Index: (title), (artist)
// Index: (fingerprint) for copyright matching
```

Pinecone (Vector Database for Recommendations)

Video embedding storage:

```
interface VideoEmbedding {
  id: string; // video_id
  vector: number[]; // 256-dim embedding
  metadata: {
    userId: string;
    hashtags: string[];
    musicId: string;
    viewCount: number;
    likeRate: number; // likes / views
    createdAt: number; // timestamp
  };
}

// Similarity search query
const similarVideos = await pinecone.query({
```

```
  vector: userPreferenceEmbedding, // User's taste vector
  topK: 100, // Get 100 candidates
  filter: {
    createdAt: { $gte: Date.now() - 7 * 86400000 } // Last 7 days
  }
});
```

MongoDB (Time-series Events)

User interactions (for ML training):

```
interface WatchEvent {
  user_id: string;
  video_id: string;

  // Watch behavior
  watch_duration_ms: number;
  total_duration_ms: number;
  completion_rate: number; // watch / total

  // Engagement signals
  liked: boolean;
  commented: boolean;
  shared: boolean;
  followed_creator: boolean;

  // Context
  source: 'fyp' | 'following' | 'search' | 'hashtag' | 'profile';
  device_type: 'ios' | 'android' | 'web';
  timestamp: Date;
}
// Index: (user_id, timestamp)
// Index: (video_id, timestamp)
// TTL: 90 days (data retention for ML)
```

Redis Cache Structure

```
// Video metadata cache
`video:${videoId}` → JSON { id, title, userId, urls, viewCount, likeCount }

// User profile cache
`user:${userId}` → JSON { id, username, avatarUrl, followersCount }

// Video engagement counters (hot path)
`video:${videoId}:views` → counter
`video:${videoId}:likes` → counter

// Trending hashtags (current hour)
`trending:${hour}` → SortedSet<hashtag, score>
```

```
// Feed candidate cache (per user)
`feed:candidates:${userId}` → JSON array of 100 video IDs

// User preference vector (cold start mitigation)
`user:embedding:${userId}` → JSON array [0.23, -0.45, ...]

// Processing queue
`video:processing:queue` → List of video IDs
```

Consistency Guarantees

Strongly consistent:

- Video upload completion
- Follow/unfollow
- Account deletion

Eventually consistent (acceptable lag: 1-5 seconds):

- View counts
- Like counts
- Follower counts
- Feed updates

Real-time (< 500ms lag):

- User's own actions reflected in UI
- Trending detection

---

5. Core Workflows

Workflow 1: User Uploads Video

Step-by-step:

1. **Client** initiates upload

```
POST /api/v1/videos/upload/init
{
  "filename": "dance.mp4",
  "filesize": 25000000, // 25MB
  "duration_ms": 15000
}

Response:
{
  "videoId": "vid-123",
  "uploadUrl": "https://s3.amazonaws.com/tiktok-uploads/...",
  "chunkSize": 5242880 // 5MB chunks
}
```

2. **Client** uploads video to S3 (multipart, resumable)

```
// Upload in chunks (parallel)
for (let i = 0; i < chunks.length; i++) {
  await uploadChunk(uploadUrl, chunks[i], i);
}

// Complete multipart upload
await completeUpload(videoId);
```

3. **Client** notifies completion

```
POST /api/v1/videos/vid-123/upload/complete
{
  "title": "My dance video",
  "description": "Learning the new trend!",
  "hashtags": ["dance", "trending"],
  "musicId": "music-456",
  "visibility": "public"
}
```

4. **Video Service** (Node.js) creates video record:

```
async function completeVideoUpload(videoId: string, metadata: VideoMetadata) {
  // Create video record in DB
  await db.query(
    `INSERT INTO videos (id, user_id, title, description, hashtags, music_id,
status, original_url, visibility, created_at)
     VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, NOW())`,
    [videoId, metadata.userId, metadata.title, metadata.description,
metadata.hashtags, metadata.musicId, 'processing', metadata.s3Key,
metadata.visibility]
  );

  // Publish to Kafka for processing
  await kafka.publish('video.uploaded', {
    videoId,
    userId: metadata.userId,
    s3Key: metadata.s3Key,
    duration: metadata.duration
  });

  return { videoId, status: 'processing' };
}
```

5. **Video Processor** (Go) picks up event:

```
func (p *VideoProcessor) ProcessVideo(event VideoUploadedEvent) error {
    // Download from S3
    originalVideo, err := p.s3.Download(event.S3Key)
    if err != nil {
        return err
```

```
    }

    // Transcode to multiple resolutions (parallel)
    var wg sync.WaitGroup
    resolutions := []string{"360p", "480p", "720p", "1080p"}
    results := make(map[string]string)

    for _, res := range resolutions {
        wg.Add(1)
        go func(resolution string) {
            defer wg.Done()

            // FFmpeg transcoding
            outputPath, err := p.transcode(originalVideo, resolution)
            if err != nil {
                log.Error("Transcode failed", zap.String("resolution",
resolution), zap.Error(err))
                return
            }

            // Upload to S3
            s3Key, err := p.s3.Upload(outputPath)
            if err != nil {
                return
            }

            results[resolution] = s3Key
        }(res)
    }

    wg.Wait()

    // Generate thumbnail
    thumbnailPath, _ := p.generateThumbnail(originalVideo)
    thumbnailKey, _ := p.s3.Upload(thumbnailPath)

    // Extract audio for copyright check
    audioPath, _ := p.extractAudio(originalVideo)
    audioFingerprint := p.generateFingerprint(audioPath)
    copyrightMatch := p.checkCopyright(audioFingerprint)

    if copyrightMatch {
        // Flag for manual review
        p.db.Exec("UPDATE videos SET status = $1 WHERE id = $2",
"copyright_flagged", event.VideoId)
        return nil
    }

    // Update video record with transcoded URLs
    p.db.Exec(`
        UPDATE videos
        SET status = $1,
```

```
            transcoded_urls = $2,
            thumbnail_url = $3,
            published_at = NOW()
        WHERE id = $4`,
        "ready",
        jsonb.Marshal(results),
        thumbnailKey,
        event.VideoId,
    )

    // Generate ML embedding (async)
    p.kafka.Publish("video.ready", VideoReadyEvent{
        VideoId: event.VideoId,
        UserId:  event.UserId,
    })

    return nil
}

// FFmpeg wrapper for transcoding
func (p *VideoProcessor) transcode(inputPath, resolution string) (string,
error) {
    outputPath := fmt.Sprintf("/tmp/%s_%s.mp4", uuid.New(), resolution)

    // Resolution mapping
    params := map[string]string{
        "360p": "-vf scale=-2:360 -b:v 600k",
        "480p": "-vf scale=-2:480 -b:v 1000k",
        "720p": "-vf scale=-2:720 -b:v 2500k",
        "1080p": "-vf scale=-2:1080 -b:v 5000k",
    }

    cmd := exec.Command(
        "ffmpeg",
        "-i", inputPath,
        "-c:v", "libx264",
        "-preset", "fast",
        params[resolution],
        "-c:a", "aac",
        "-b:a", "128k",
        outputPath,
    )

    err := cmd.Run()
    return outputPath, err
}
```

6. **ML Embedding Worker** (Python) generates video embedding:

```
def generate_video_embedding(video_id: str, user_id: str):
    # Download video thumbnail + metadata
    video = get_video_metadata(video_id)
```

```
    # Generate embedding (256-dim vector)
    # Combines: visual features, hashtags, music, creator profile
    embedding = model.encode({
        'thumbnail_url': video['thumbnail_url'],
        'hashtags': video['hashtags'],
        'music_id': video['music_id'],
        'creator_followers': video['user']['followers_count']
    })

    # Store in Pinecone
    pinecone.upsert(vectors=[{
        'id': video_id,
        'values': embedding.tolist(),
        'metadata': {
            'user_id': user_id,
            'hashtags': video['hashtags'],
            'created_at': video['created_at']
        }
    }])

    # Invalidate user's feed cache (new content available)
    redis.delete(f"feed:candidates:{user_id}")
```

7. **Client** polls for processing status

```
GET /api/v1/videos/vid-123/status

Response:
{
  "videoId": "vid-123",
  "status": "ready", // or "processing" or "failed"
  "urls": {
    "360p": "https://cdn.tiktok.com/...",
    "720p": "https://cdn.tiktok.com/..."
  }
}
```

Performance targets:

- Upload to S3: 10-30 seconds (depends on user bandwidth)
- Transcoding: 15-30 seconds (4 resolutions parallel)
- ML embedding: 5 seconds
- **Total time to "ready"**: 30-60 seconds

Failure handling:

- S3 upload fails → client retries with exponential backoff
- Transcoding fails → mark video as "failed", notify user
- Copyright match → hold for manual review
- ML embedding fails → video still published, embedding computed later

---

Workflow 2: User Scrolls "For You" Feed

Challenge: Predict which video will hook user within 3 seconds

Step-by-step:

1. **Client** opens app, requests feed

```
GET /api/v1/feed?count=20&cursor=0
```

2. **Feed Service** (Go) generates personalized feed:

```
func (s *FeedService) GetFeed(userId string, count int, cursor int) ([]Video,
error) {
    // Check if user has cached candidates
    cachedCandidates, err := s.redis.Get(ctx,
fmt.Sprintf("feed:candidates:%s", userId)).Result()

    var candidateIds []string

    if err == redis.Nil {
        // No cache: Generate candidates
        candidateIds, err = s.generateCandidates(userId)
        if err != nil {
            return nil, err
        }

        // Cache for 10 minutes
        s.redis.Set(ctx, fmt.Sprintf("feed:candidates:%s", userId),
json.Marshal(candidateIds), 10*time.Minute)
    } else {
        json.Unmarshal([]byte(cachedCandidates), &candidateIds)
    }

    // Paginate candidates
    start := cursor
    end := cursor + count
    if end > len(candidateIds) {
        end = len(candidateIds)
    }

    pageIds := candidateIds[start:end]

    // Hydrate videos from cache/DB
    videos := s.hydrateVideos(pageIds)

    // Pre-fetch next page (background)
    go s.prefetchVideos(candidateIds[end:end+count], userId)

    return videos, nil
}
```

3. **Generate candidates** (Recommendation Engine):

```
func (s *FeedService) generateCandidates(userId string) ([]string, error) {
    // Get user's preference embedding
    userEmbedding, err := s.getUserEmbedding(userId)
    if err != nil {
        // Cold start: Use default embedding
        userEmbedding = s.getDefaultEmbedding()
    }

    // Retrieve candidates from Pinecone (vector similarity search)
    results, err := s.pinecone.Query(userEmbedding, 500) // Get 500 candidates
    if err != nil {
        return nil, err
    }

    // Diversify results (avoid all similar videos)
    diversified := s.diversify(results, userId)

    // Ranking model (predict watch time)
    rankedIds := s.rankCandidates(diversified, userId)

    return rankedIds[:100], nil // Return top 100
}

func (s *FeedService) getUserEmbedding(userId string) ([]float32, error) {
    // Check cache
    cached, err := s.redis.Get(ctx, fmt.Sprintf("user:embedding:%s",
userId)).Result()
    if err == nil {
        var embedding []float32
        json.Unmarshal([]byte(cached), &embedding)
        return embedding, nil
    }

    // Compute from user's watch history
    watchHistory, _ := s.getWatchHistory(userId, 100) // Last 100 videos

    // Average embeddings of watched videos (weighted by watch time)
    var aggregated []float32
    for _, watch := range watchHistory {
        videoEmbedding := s.getVideoEmbedding(watch.VideoId)
        weight := watch.WatchDuration / watch.TotalDuration // completion rate

        aggregated = addWeighted(aggregated, videoEmbedding, weight)
    }

    // Normalize
    embedding := normalize(aggregated)

    // Cache for 1 hour
```

```
    s.redis.Set(ctx, fmt.Sprintf("user:embedding:%s", userId),
json.Marshal(embedding), 1*time.Hour)

    return embedding, nil
}

// Diversify to avoid filter bubble
func (s *FeedService) diversify(candidates []string, userId string) []string {
    var diversified []string
    seen := make(map[string]bool)

    for _, videoId := range candidates {
        video := s.getVideoMetadata(videoId)

        // Avoid showing too many videos from same creator
        if seen[video.UserId] {
            continue
        }

        diversified = append(diversified, videoId)
        seen[video.UserId] = true

        if len(diversified) >= 200 {
            break
        }
    }

    return diversified
}
```

4. **Ranking model** (Python ML service):

```
def rank_candidates(candidate_ids: List[str], user_id: str) -> List[str]:
    # Load user features
    user_features = get_user_features(user_id)

    # Load candidate video features
    video_features = [get_video_features(vid) for vid in candidate_ids]

    # Predict watch time for each video
    predictions = []
    for i, vid in enumerate(candidate_ids):
        # Features: user profile, video metadata, interaction history
        features = combine_features(user_features, video_features[i])

        # Model outputs predicted watch duration
        predicted_watch_time = model.predict(features)

        predictions.append({
            'video_id': vid,
            'score': predicted_watch_time
        })
```

```
    # Sort by predicted engagement
    ranked = sorted(predictions, key=lambda x: x['score'], reverse=True)

    return [p['video_id'] for p in ranked]
```

5. **Client** receives videos, starts playing first one

```
Response:
{
  "videos": [
    {
      "id": "vid-123",
      "user": {...},
      "title": "Amazing dance",
      "urls": {
        "360p": "https://cdn.tiktok.com/...",
        "720p": "https://cdn.tiktok.com/..."
      },
      "thumbnail": "https://cdn.tiktok.com/...",
      "likeCount": 15000,
      "commentCount": 234
    },
    // ...19 more
  ],
  "nextCursor": 20
}
```

6. **Client** sends watch events (real-time)

```
// User watched 8 seconds of 15-second video, then swiped
POST /api/v1/analytics/watch
{
  "videoId": "vid-123",
  "watchDuration": 8000,
  "totalDuration": 15000,
  "liked": false,
  "commented": false,
  "shared": false,
  "source": "fyp"
}
```

7. **Analytics Service** (Go) updates user embedding in real-time:

```
func (a *AnalyticsService) RecordWatch(event WatchEvent) error {
    // Store event in MongoDB (for ML training)
    a.mongo.InsertOne("watch_events", event)

    // Update video counters (eventually consistent)
    a.redis.Incr(ctx, fmt.Sprintf("video:%s:views", event.VideoId))
```

```
    // Update user embedding immediately (online learning)
    if event.WatchDuration > 0.5 * event.TotalDuration {
        // User watched >50%: positive signal
        a.updateUserEmbedding(event.UserId, event.VideoId, 1.0)
    } else {
        // User skipped: negative signal
        a.updateUserEmbedding(event.UserId, event.VideoId, -0.5)
    }

    // Invalidate feed cache (next feed will be more personalized)
    a.redis.Del(ctx, fmt.Sprintf("feed:candidates:%s", event.UserId))

    return nil
}

func (a *AnalyticsService) updateUserEmbedding(userId, videoId string, weight
float64) {
    // Get current user embedding
    userEmb := a.getUserEmbedding(userId)

    // Get video embedding
    videoEmb := a.getVideoEmbedding(videoId)

    // Update with momentum (exponential moving average)
    updated := make([]float32, len(userEmb))
    for i := range userEmb {
        updated[i] = 0.9*userEmb[i] + 0.1*float32(weight)*videoEmb[i]
    }

    // Cache updated embedding
    a.redis.Set(ctx, fmt.Sprintf("user:embedding:%s", userId),
json.Marshal(updated), 1*time.Hour)
}
```

Performance targets:

- Candidate generation: 50-100ms
- Ranking: 50-100ms
- Video metadata hydration: 20-50ms
- **Total feed generation**: 150-300ms

Workflow 3: User Likes Video

Step-by-step:

1. **Client** sends like request

```
POST /api/v1/videos/vid-123/like
```

2. **Video Service** (Node.js) records like:

```typescript
async function likeVideo(userId: string, videoId: string) {
  // Idempotent insert
  const result = await db.query(
    `INSERT INTO video_engagement (video_id, user_id, type, created_at)
     VALUES ($1, $2, 'like', NOW())
     ON CONFLICT (video_id, user_id, type) DO NOTHING
     RETURNING *`,
    [videoId, userId]
  );

  if (result.rows.length === 0) {
    return { alreadyLiked: true };
  }

  // Increment Redis counter (fast path)
  await redis.incr(`video:${videoId}:likes`);

  // Publish event for analytics
  await kafka.publish('video.liked', {
    videoId,
    userId,
    timestamp: Date.now()
  });

  return { success: true };
}
```

3. **Background worker** updates DB counter:

```go
// Batch updates every 10 seconds
func (w *CounterWorker) FlushLikes() {
    ticker := time.NewTicker(10 * time.Second)

    for range ticker.C {
        // Get all video like counters from Redis
        keys, _ := w.redis.Keys(ctx, "video:*:likes").Result()

        for _, key := range keys {
            videoId := extractVideoId(key)
            count, _ := w.redis.Get(ctx, key).Int64()

            // Update DB
            w.db.Exec(
                "UPDATE videos SET like_count = like_count + $1 WHERE id =
$2",
                count, videoId,
            )

            // Reset Redis counter
            w.redis.Set(ctx, key, 0, 0)
        }
```

```
    }
}
```

4. **Recommendation model** updates (real-time signal):
    - Like is strong positive signal
    - Update user embedding immediately
    - Boost similar videos in future recommendations

---

6. API Design

REST Endpoints

Authentication:

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

Feed:

```
GET /api/v1/feed?count=20&cursor=0
GET /api/v1/feed/following?count=20 // Videos from followed users
```

Video:

```
POST   /api/v1/videos/upload/init
POST   /api/v1/videos/:id/upload/complete
GET    /api/v1/videos/:id
DELETE /api/v1/videos/:id
POST   /api/v1/videos/:id/like
DELETE /api/v1/videos/:id/like
POST   /api/v1/videos/:id/comment
GET    /api/v1/videos/:id/comments?limit=20
```

User:

```
GET    /api/v1/users/:userId
GET    /api/v1/users/:userId/videos?limit=20
POST   /api/v1/users/:userId/follow
DELETE /api/v1/users/:userId/follow
GET    /api/v1/users/:userId/followers?limit=20
GET    /api/v1/users/:userId/following?limit=20
```

Search:

```
GET /api/v1/search?q=query&type=videos|users
GET /api/v1/hashtag/:hashtag/videos?limit=20
```

Analytics:

```
POST /api/v1/analytics/watch (client → server events)
GET  /api/v1/analytics/creator (creator dashboard)
```

CDN Video URLs

Structure:

```
https://cdn.tiktok.com/videos/{region}/{video_id}/{resolution}.m3u8

Example:
https://cdn.tiktok.com/videos/us-east/vid-123/720p.m3u8
```

Adaptive bitrate (HLS):

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=600000,RESOLUTION=640x360
https://cdn.tiktok.com/videos/us-east/vid-123/360p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1000000,RESOLUTION=854x480
https://cdn.tiktok.com/videos/us-east/vid-123/480p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2500000,RESOLUTION=1280x720
https://cdn.tiktok.com/videos/us-east/vid-123/720p.m3u8
```

Client automatically switches resolution based on network speed.

Idempotency

Video upload:

- Client generates video ID before upload
- Multiple "complete" requests with same ID → idempotent

```
// Client-side
const videoId = generateUUID();

// If network fails during upload, retry with same videoId
await uploadVideo(videoId, file);
```

---

7. Cold Start Problem

Challenge: New user has no watch history → how to personalize?

Solutions

1. Onboarding quiz:

```
// During signup
POST /api/v1/onboarding/interests
{
```

```
  "interests": ["dance", "cooking", "comedy", "tech"]
}

// Backend maps interests to seed videos
const seedEmbedding = averageEmbeddings(
  getVideosByHashtags(["dance", "cooking", "comedy", "tech"])
);

redis.set(`user:embedding:${userId}`, seedEmbedding);
```

2. Popular videos (warm start):

```
func (s *FeedService) generateCandidatesForNewUser(userId string) []string {
    // Show globally popular videos (high engagement)
    popularIds, _ := s.redis.ZRevRange(ctx, "trending:global", 0, 100).Result()

    // Mix with random diverse content
    randomIds, _ := s.getRandomDiverseVideos(100)

    // Shuffle together
    candidates := append(popularIds, randomIds...)
    shuffle(candidates)

    return candidates[:100]
}
```

3. Fast convergence:

- After user watches 3-5 videos → generate embedding
- After 10 videos → personalization kicks in
- After 50 videos → fully personalized

4. Multi-armed bandit (exploration vs exploitation):

```
func (s *FeedService) exploreExploit(userId string) []string {
    // 80% personalized (exploitation)
    personalized := s.generatePersonalizedCandidates(userId, 80)

    // 20% random/trending (exploration)
    exploratory := s.getTrendingVideos(20)

    return append(personalized, exploratory...)
}
```

---

8. Consistency, Ordering & Concurrency

Consistency Model by Feature

Strong consistency:

- Video upload completion (can't be "partially uploaded")

- Follow/unfollow operations
- Account deletion

Eventually consistent:

- View counts (acceptable lag: 1-5 seconds)
- Like counts
- Follower counts
- Feed recommendations

Real-time (< 500ms):

- User's own likes/comments reflected immediately
- Video they just uploaded appears in their profile

Video Processing Race Condition

Problem:

```
User uploads video → Processing starts
User deletes video mid-processing
Processed video appears on platform (zombie video)
```

Solution: Status flag + atomic updates

```
func (p *VideoProcessor) ProcessVideo(event VideoUploadedEvent) error {
    // Check video still exists and is in "processing" state
    video, err := p.db.QueryRow(
        "SELECT id, status FROM videos WHERE id = $1 AND status = $2 FOR UPDATE",
        event.VideoId, "processing",
    ).Scan(&id, &status)

    if err == sql.ErrNoRows {
        // Video was deleted or already processed
        log.Info("Video no longer needs processing", zap.String("videoId",
event.VideoId))
        return nil
    }

    // Process video...
    transcodedUrls := p.transcode(video)

    // Atomic update: Only update if still in "processing" state
    result := p.db.Exec(
        "UPDATE videos SET status = $1, transcoded_urls = $2 WHERE id = $3 AND
status = $4",
        "ready", transcodedUrls, event.VideoId, "processing",
    )

    if result.RowsAffected == 0 {
        // Video was deleted during processing
        log.Info("Video deleted during processing", zap.String("videoId",
event.VideoId))
        // Clean up transcoded files
```

```
        p.s3.DeleteFiles(transcodedUrls)
    }

    return nil
}
```

View Counter Race Condition

Problem:

- 10K simultaneous views on viral video
- All incrementing same counter
- Lost updates

Solution: Redis atomic increment + batch flush

```
// Real-time: Increment Redis (atomic)
func (a *AnalyticsService) RecordView(videoId string) {
    a.redis.Incr(ctx, fmt.Sprintf("video:%s:views", videoId))
}

// Background: Batch flush to DB
func (w *CounterWorker) FlushViews() {
    ticker := time.NewTicker(10 * time.Second)

    for range ticker.C {
        // Lua script for atomic get-and-reset
        script := `
            local count = redis.call('GET', KEYS[1])
            redis.call('SET', KEYS[1], 0)
            return count
        `

        keys, _ := w.redis.Keys(ctx, "video:*:views").Result()

        for _, key := range keys {
            videoId := extractVideoId(key)

            // Atomic get-and-reset
            count, _ := w.redis.Eval(ctx, script, []string{key}).Int64()

            if count > 0 {
                // Update DB
                w.db.Exec(
                    "UPDATE videos SET view_count = view_count + $1 WHERE id = $2",
                    count, videoId,
                )
            }
        }
    }
}
```

Feed Candidate Generation Concurrency

Problem:

- Multiple feed requests from same user
- All trigger candidate generation
- Wasted computation

Solution: Distributed lock

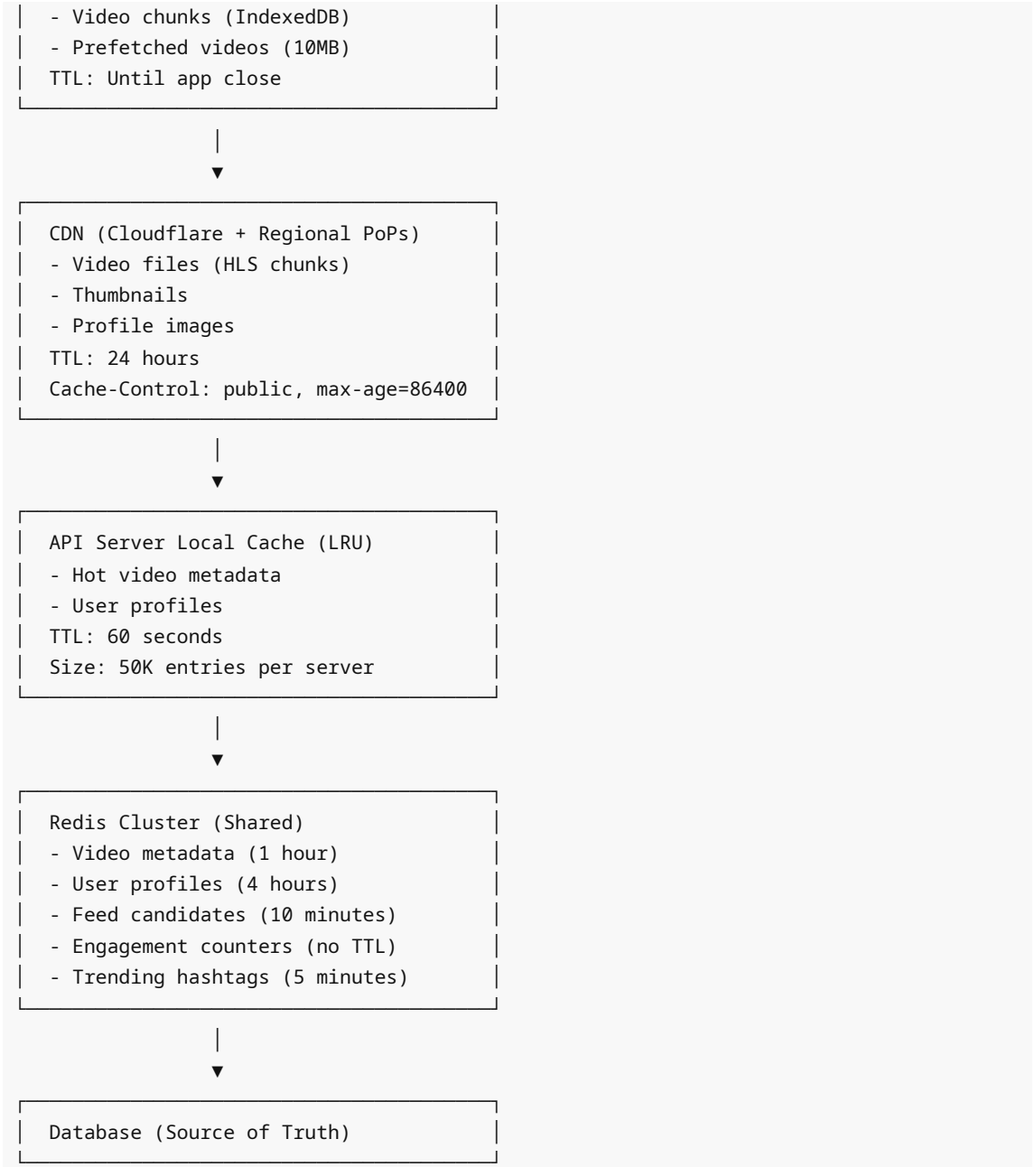
```
func (s *FeedService) generateCandidates(userId string) ([]string, error) {
    lockKey := fmt.Sprintf("lock:feed:gen:%s", userId)

    // Try to acquire lock (TTL: 30 seconds)
    acquired, err := s.redis.SetNX(ctx, lockKey, "1", 30*time.Second).Result()

    if !acquired {
        // Another request is already generating candidates
        // Wait briefly and check cache
        time.Sleep(100 * time.Millisecond)

        cached, _ := s.redis.Get(ctx, fmt.Sprintf("feed:candidates:%s",
userId)).Result()
        if cached != "" {
            var candidates []string
            json.Unmarshal([]byte(cached), &candidates)
            return candidates, nil
        }

        // Still not ready: return error (client will retry)
        return nil, errors.New("feed generation in progress")
    }

    defer s.redis.Del(ctx, lockKey)

    // Generate candidates
    candidates := s.computeCandidates(userId)

    // Cache result
    s.redis.Set(ctx, fmt.Sprintf("feed:candidates:%s", userId),
json.Marshal(candidates), 10*time.Minute)

    return candidates, nil
}
```

---

9. Caching Strategy

Multi-Layer Cache Architecture

```
┌─────────────────────────────────────┐
│  Browser Cache                      │
```



Video Metadata Caching

```
async function getVideo(videoId: string): Promise<Video> {
  // Layer 1: Local cache
  let video = localCache.get(videoId);
  if (video) return video;

  // Layer 2: Redis
  const cached = await redis.get(`video:${videoId}`);
  if (cached) {
    video = JSON.parse(cached);
    localCache.set(videoId, video);
```

```
    return video;
  }

  // Layer 3: Database
  video = await db.query('SELECT * FROM videos WHERE id = $1', [videoId]);

  // Cache in Redis (1 hour for normal, 24 hours for viral)
  const ttl = video.viewCount > 100000 ? 86400 : 3600;
  await redis.setex(`video:${videoId}`, ttl, JSON.stringify(video));

  localCache.set(videoId, video);

  return video;
}
```

Feed Candidate Caching

```
func (s *FeedService) getCachedCandidates(userId string) ([]string, bool) {
    // Cache key includes user ID
    key := fmt.Sprintf("feed:candidates:%s", userId)

    cached, err := s.redis.Get(ctx, key).Result()
    if err == redis.Nil {
        return nil, false
    }

    var candidates []string
    json.Unmarshal([]byte(cached), &candidates)

    return candidates, true
}

func (s *FeedService) cacheCandidates(userId string, candidates []string) {
    key := fmt.Sprintf("feed:candidates:%s", userId)

    // TTL: 10 minutes (feed personalizes quickly)
    s.redis.Set(ctx, key, json.Marshal(candidates), 10*time.Minute)
}
```

Prefetching Strategy

Client-side video prefetch:

```
class VideoFeedPlayer {
  private prefetchQueue: string[] = [];

  async playVideo(videoId: string) {
    // Start playing current video
    this.player.load(videoId);
```

```
    // Prefetch next 3 videos in background
    const nextVideos = this.feedVideos.slice(
      this.currentIndex + 1,
      this.currentIndex + 4
    );

    for (const video of nextVideos) {
      this.prefetchVideo(video.id, video.urls['720p']);
    }
  }

  private async prefetchVideo(videoId: string, url: string) {
    if (this.prefetchQueue.includes(videoId)) return;

    this.prefetchQueue.push(videoId);

    // Download first 2MB (first 3-5 seconds)
    const response = await fetch(url, {
      headers: { 'Range': 'bytes=0-2097152' }
    });

    const blob = await response.blob();

    // Store in IndexedDB
    await this.cache.put(videoId, blob);
  }
}
```

Server-side:

```
func (s *FeedService) prefetchNextVideos(videoIds []string, userId string) {
    // Warm up video metadata cache
    for _, videoId := range videoIds {
        go func(vid string) {
            // Trigger cache load
            s.getVideo(vid)
        }(videoId)
    }

    // Warm up CDN (send HEAD request to CDN URLs)
    for _, videoId := range videoIds {
        go func(vid string) {
            video := s.getVideo(vid)

            // HEAD request to CDN (causes CDN to fetch from origin)
            http.Head(video.TranscodedUrls["720p"])
        }(videoId)
    }
}
```

Cache Invalidation

On video update:

```
async function updateVideo(videoId: string, updates: VideoUpdate) {
  await db.query('UPDATE videos SET ... WHERE id = $1', [videoId, ...]);

  // Invalidate all caches
  await redis.del(`video:${videoId}`);
  localCache.delete(videoId);

  // Purge CDN cache
  await cloudflare.purgeCache([
    `https://cdn.tiktok.com/videos/*/vid-${videoId}/*`
  ]);
}
```

On user follow:

```
func (s *SocialService) Follow(followerId, followeeId string) error {
    // Update DB
    s.db.Exec("INSERT INTO follows (follower_id, followee_id, created_at) VALUES
($1, $2, NOW())", followerId, followeeId)

    // Invalidate follower's feed candidates
    s.redis.Del(ctx, fmt.Sprintf("feed:candidates:%s", followerId))

    // Invalidate user embedding (interests changed)
    s.redis.Del(ctx, fmt.Sprintf("user:embedding:%s", followerId))

    return nil
}
```

---

10. Scaling Strategy

Horizontal Scaling

API Gateway:

- Stateless Node.js servers
- 500+ instances globally (geo-distributed)
- Auto-scale on CPU (target 70%)
- Sticky sessions not needed

Video Service:

- Stateless Node.js
- 200+ instances
- Heavy S3 integration
- Connection pooling critical

Feed Service (Go):

- Stateless
- 100+ instances
- CPU-heavy (JSON parsing, ranking)
- gRPC for internal communication

Video Processors (Go):

- Stateful (processing videos)
- 50-100 instances
- Each handles 5-10 videos concurrently
- Auto-scale based on queue depth
- GPU instances for ML encoding

ML Recommendation Service (Python):

- GPU instances (T4/V100)
- 20-30 instances
- Model served via TensorFlow Serving
- Load balanced by request latency

Database Sharding

Videos table (sharded by video_id):

```
// 64 shards (video_id is Snowflake ID)
function getVideoShard(videoId: string): number {
  // Use first 6 bits of Snowflake ID
  const idNum = BigInt(videoId);
  return Number(idNum % 64n);
}

async function getVideo(videoId: string) {
  const shard = getVideoShard(videoId);
  const db = getDBConnection(`videos_shard_${shard}`);
  return db.query('SELECT * FROM videos WHERE id = $1', [videoId]);
}
```

Users table (sharded by user_id):

```
function getUserShard(userId: string): number {
  return xxhash(userId) % 32;
}

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 query (user's videos):

```
// BAD: Videos sharded by video_id, not user_id
// Can't efficiently query "all videos by user X"

// SOLUTION: Denormalize with secondary index
// Maintain separate table: user_videos (sharded by user_id)
interface UserVideo {
  user_id: string;
  video_id: string;
  created_at: timestamp;
}
// Shard by user_id

async function getUserVideos(userId: string) {
  const shard = getUserShard(userId);
  const db = getDBConnection(`user_videos_shard_${shard}`);

  const videoIds = await db.query(
    'SELECT video_id FROM user_videos WHERE user_id = $1 ORDER BY created_at DESC
LIMIT 20',
    [userId]
  );

  // Fetch full video objects (scatter-gather across video shards)
  const videos = await Promise.all(
    videoIds.map(id => getVideo(id.video_id))
  );

  return videos;
}
```

CDN Scaling

Multi-CDN strategy:

```
Primary: Cloudflare (global)
Secondary: AWS CloudFront (Asia-Pacific)
Tertiary: Akamai (enterprise customers)

Routing logic:
- User in US/EU → Cloudflare
- User in Asia → CloudFront (lower latency)
- Enterprise customer → Akamai (SLA guarantees)
```

Video chunk distribution:

```
Origin: S3 (us-east-1, eu-west-1, ap-southeast-1)

CDN PoPs: 200+ locations worldwide
- Each PoP caches hot videos (top 10% by views)
- Cold videos fetched from nearest origin on-demand
- Cache size: 100TB per PoP
```

```
Cost optimization:
- Viral video (10M views): Cached at all PoPs → $0.01/GB egress
- New video (<100 views): Served from origin → $0.09/GB egress
```

Pinecone Vector DB Scaling

Index structure:

```
Index: "video-embeddings" (256 dimensions)
Pods: 2 (p1.x1) → 2M vectors, 100ms p95 query latency

Scaling:
- 10M vectors: 10 pods
- 100M vectors: 100 pods
- 1B vectors: 1000 pods + metadata filtering
```

Query optimization:

```python
# Bad: Search all 1B vectors
results = index.query(user_embedding, top_k=100)

# Good: Filter by metadata (recent videos only)
results = index.query(
    user_embedding,
    top_k=100,
    filter={
        "created_at": {"$gte": int(time.time()) - 7*86400},  # Last 7 days
        "view_count": {"$gte": 10}  # Exclude spam/low-quality
    }
)
```

Read Replicas

Setup per shard:

```
Primary (writes only)
├── Replica 1 (reads)
├── Replica 2 (reads)
└── Replica 3 (reads)

Routing:
- Writes → primary
- Reads → round-robin replicas
- Critical reads (just after write) → primary
```

Replication lag handling:

```typescript
async function getVideoAfterUpload(videoId: string) {
  // Just uploaded: read from primary (avoid replication lag)
  const shard = getVideoShard(videoId);
```

```
  const primaryDB = getPrimaryDB(shard);
  return primaryDB.query('SELECT * FROM videos WHERE id = $1', [videoId]);
}

async function getBrowseVideo(videoId: string) {
  // Browsing: read from replica (lag acceptable)
  const shard = getVideoShard(videoId);
  const replicaDB = getRandomReplica(shard);
  return replicaDB.query('SELECT * FROM videos WHERE id = $1', [videoId]);
}
```

---

11. Fault Tolerance & Reliability

Failure Scenarios

1. Video processing fails:

- **Impact**: User's video stuck in "processing"
- **Detection**: Worker timeout (5 minutes)
- **Mitigation**:
  - Retry 3 times with exponential backoff
  - If still fails → mark as "failed", notify user
  - Manual review queue for debugging
- **User experience**: "Video processing failed. Please try again."

2. CDN cache miss during viral event:

- **Scenario**: Trending video not in CDN cache
- **Impact**: Sudden 100K requests hit origin S3
- **Detection**: Origin request rate spike
- **Mitigation**:
  - S3 request rate limit (3500 req/sec per prefix)
  - Use different prefixes: `videos/ABC/vid-123`, not `videos/vid-123`
  - Pre-warm CDN for trending videos
- **Degradation**: Some users experience 1-2 second load time

3. Recommendation service down:

- **Impact**: Can't generate feed candidates
- **Detection**: Health check fails, gRPC timeout
- **Mitigation**:
  - Fallback to trending videos (globally popular)
  - Show following feed as backup
  - Circuit breaker prevents cascading failures
- **User experience**: Less personalized feed temporarily

```
func (s *FeedService) generateCandidatesWithFallback(userId string) []string {
    // Try personalized recommendations
    candidates, err := s.recommendationService.GetCandidates(userId)

    if err != nil {
```

```
        log.Error("Recommendation service failed, falling back to trending",
zap.Error(err))

        // Fallback 1: Trending videos
        trending, err := s.getTrendingVideos(100)
        if err == nil {
            return trending
        }

        // Fallback 2: Following feed
        following, err := s.getFollowingFeed(userId, 100)
        if err == nil {
            return following
        }

        // Fallback 3: Random popular videos
        return s.getRandomPopularVideos(100)
    }

    return candidates
}
```

4. Database shard failure:

- **Impact**: Videos on that shard unavailable (1/64 of videos)
- **Detection**: Connection timeout, health check fails
- **Mitigation**:
  - Automatic failover to read replica (Patroni)
  - RPO: ~1 second (replication lag)
  - RTO: ~2 minutes
- **During failover**: Affected videos return 503, client retries

5. Kafka partition leader failure:

- **Impact**: Video processing events delayed
- **Detection**: Consumer lag metric spikes
- **Mitigation**:
  - Kafka auto-elects new leader
  - RTO: ~10 seconds
- **Effect**: Videos take 10-20 seconds longer to process

6. S3 region outage:

- **Scenario**: us-east-1 S3 down (rare but happens)
- **Impact**: Videos stored in us-east-1 unavailable
- **Mitigation**:
  - Cross-region replication (S3 CRR)
  - Automatic failover to eu-west-1
  - CDN fetches from backup region
- **RTO**: ~5 minutes (DNS propagation)

Partial Failure Handling

Video upload succeeds, processing fails:

```
func (s *VideoService) CompleteUpload(videoId string, metadata VideoMetadata) error
{
    // CRITICAL: Mark video as uploaded
    err := s.db.Exec(
        "UPDATE videos SET status = $1, title = $2, description = $3 WHERE id = $4",
        "processing", metadata.Title, metadata.Description, videoId,
    )
    if err != nil {
        return err // Fail request if DB write fails
    }

    // NON-CRITICAL: Trigger processing
    err = s.kafka.Publish("video.uploaded", VideoUploadedEvent{VideoId: videoId})
    if err != nil {
        // Log but don't fail request
        log.Error("Failed to publish processing event", zap.Error(err))

        // Add to retry queue
        s.retryQueue.Add(RetryTask{Type: "video_processing", VideoId: videoId})
    }

    return nil
}
```

Feed generation fails:

```
async function getFeed(userId: string) {
  try {
    const feed = await feedService.generate(userId);
    return feed;
  } catch (err) {
    logger.error('Feed generation failed', {userId, error: err});

    // Don't return error to user
    // Show fallback content
    return {
      videos: await getTrendingVideos(20),
      isFallback: true
    };
  }
}
```

RPO/RTO Targets

Component	RPO	RTO	Strategy
Videos (metadata)	0	2 min	Synchronous replication
Video files (S3)	15 min	5 min	Cross-region replication

User profiles	0	2 min	Synchronous replication
Engagement counts	10 sec	30 sec	Async replication
Feed cache	N/A	10 min	Rebuild from DB/ML
ML embeddings	1 hour	30 min	Recompute from Pinecone

---

12. Observability & Operations

Key Metrics

Video Processing:

```
video_processing_duration_seconds{resolution="720p", quantile="0.95"}
video_processing_failures_total{reason="transcode_timeout"}
video_processing_queue_depth
```

Feed Generation:

```
feed_generation_duration_seconds{quantile="0.95"}
feed_cache_hit_rate
recommendation_service_latency_seconds
```

Video Playback:

```
video_start_time_seconds{region="us-east", quantile="0.95"}
video_buffering_events_total
video_completion_rate (watched to end)
```

CDN:

```
cdn_hit_rate{pop="us-east"}
cdn_bandwidth_bytes{direction="egress"}
cdn_origin_requests_total
```

Database:

```
db_query_duration_seconds{operation="SELECT", quantile="0.95"}
db_connections_active{shard="1"}
db_replication_lag_seconds{shard="1"}
```

ML:

```
pinecone_query_latency_seconds{quantile="0.95"}
ml_model_prediction_latency_seconds
user_embedding_update_rate
```

Structured Logging

```
logger.info({
  event: 'video_uploaded',
  videoId: 'vid-123',
  userId: 'user-456',
  fileSize: 25000000,
  duration: 15000,
  resolution: '1080p',
  timestamp: Date.now()
});

logger.error({
  event: 'video_processing_failed',
  videoId: 'vid-789',
  userId: 'user-999',
  error: err.message,
  stack: err.stack,
  ffmpeg_exit_code: 1,
  attempt: 3,
  timestamp: Date.now()
});
```

Alerts

🔴 **Critical (page on-call):**

- Video processing queue depth > 10K for 5 minutes
- CDN error rate > 5%
- Database shard down
- Video start time p95 > 3 seconds
- Recommendation service down

🟡 **Warning:**

- Video processing time p95 > 60 seconds
- Feed generation time p95 > 500ms
- Cache hit rate < 80%
- Replication lag > 5 seconds

🟢 **Informational:**

- New deploy completed
- Viral video detected (10K views in 1 hour)
- Auto-scaling triggered

Distributed Tracing

```
import { trace } from '@opentelemetry/api';

async function generateFeed(userId: string) {
  const span = trace.getTracer('tiktok').startSpan('generate_feed');
  span.setAttribute('userId', userId);
```

```
  try {
    // Check cache
    const cacheSpan = trace.getTracer('tiktok').startSpan('check_feed_cache');
    const cached = await redis.get(`feed:candidates:${userId}`);
    cacheSpan.end();

    if (!cached) {
      // Generate candidates
      const recSpan =
trace.getTracer('tiktok').startSpan('call_recommendation_service');
      const candidates = await recommendationService.getCandidates(userId);
      recSpan.setAttribute('candidate_count', candidates.length);
      recSpan.end();

      // Rank
      const rankSpan = trace.getTracer('tiktok').startSpan('rank_candidates');
      const ranked = await rankingService.rank(candidates, userId);
      rankSpan.end();

      return ranked;
    }

    span.setAttribute('cache_hit', true);
    return JSON.parse(cached);
  } finally {
    span.end();
  }
}
```

Trace output:

```
generate_feed (420ms)
├─ check_feed_cache (8ms) ✓ cache_hit=false
├─ call_recommendation_service (280ms)
│  ├─ get_user_embedding (50ms)
│  ├─ pinecone_query (180ms)
│  └─ diversify_results (50ms)
└─ rank_candidates (132ms)
   ├─ load_ml_model (12ms)
   ├─ predict_batch (100ms)
   └─ sort_results (20ms)
```

Debugging Runbook

"My video is stuck in processing!"

1. Check video status:

   ```
   SELECT id, status, created_at FROM videos WHERE id = 'vid-123';
   ```

2. Check processing queue:

```
redis-cli> LLEN video:processing:queue
```

3. Check worker logs:

```
kubectl logs -l app=video-processor --tail=100 | grep vid-123
```

4. Manually trigger reprocessing:

```
curl -X POST https://api.tiktok.com/internal/videos/vid-123/reprocess \
  -H "X-Admin-Token: ..."
```

"Feed is not personalized!"

1. Check if user has watch history:

```
mongo> db.watch_events.count({user_id: "user-123"})
```

2. Check user embedding:

```
redis-cli> GET user:embedding:user-123
```

3. Check Pinecone index:

```
curl https://api.pinecone.io/describe_index_stats
```

4. Manually regenerate embedding:

```
curl -X POST https://api.tiktok.com/internal/users/user-
123/regenerate_embedding \
  -H "X-Admin-Token: ..."
```

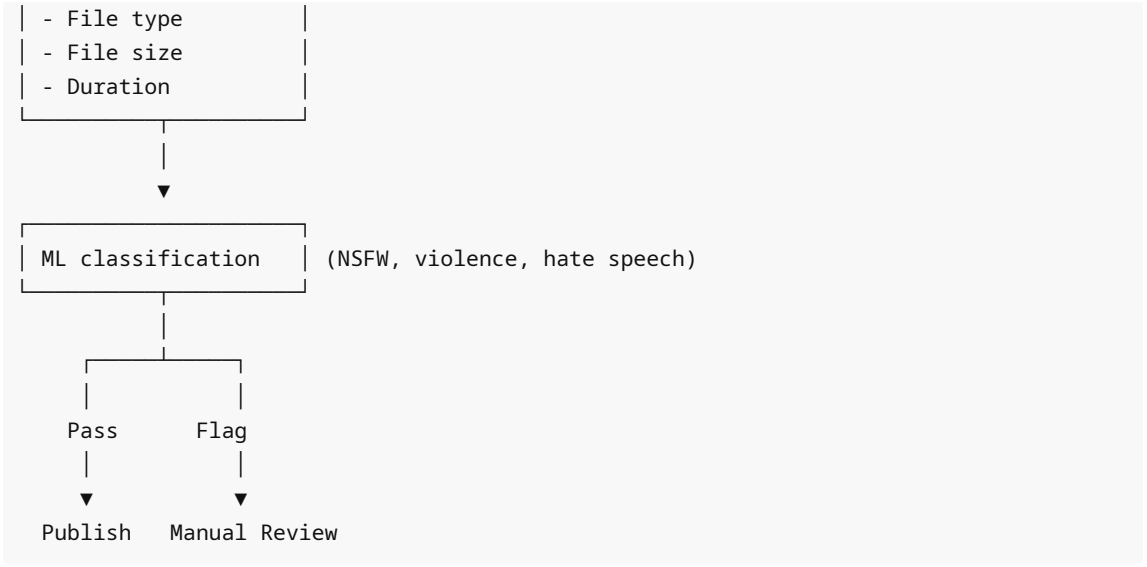
---

13. Security & Abuse Prevention

Content Moderation Pipeline

Multi-stage filtering:

```
Video Upload
     │
     ▼
┌─────────────────────┐
│ Client-side check   │ (hash against known bad content)
└──────────┬──────────┘
           │
           ▼
┌─────────────────────┐
│ Server-side checks  │
```



ML-based content moderation:

```
def moderate_video(video_id: str) -> dict:
    video = get_video_metadata(video_id)

    # Extract frames for analysis
    frames = extract_frames(video['original_url'], num_frames=10)

    # Run through moderation model
    results = {
        'nsfw_score': 0.0,
        'violence_score': 0.0,
        'hate_speech_score': 0.0
    }

    for frame in frames:
        # Vision model
        vision_result = vision_model.predict(frame)
        results['nsfw_score'] = max(results['nsfw_score'], vision_result['nsfw'])
        results['violence_score'] = max(results['violence_score'],
vision_result['violence'])

    # Audio analysis
    audio = extract_audio(video['original_url'])
    audio_result = audio_model.predict(audio)
    results['hate_speech_score'] = audio_result['hate_speech']

    # Text analysis (title, description)
    text_result = text_model.predict(video['title'] + ' ' + video['description'])
    results['hate_speech_score'] = max(results['hate_speech_score'],
text_result['hate_speech'])

    # Apply thresholds
    if results['nsfw_score'] > 0.8 or results['violence_score'] > 0.8:
        return {'action': 'block', 'reason': 'explicit_content'}
```

```
    elif results['nsfw_score'] > 0.5 or results['violence_score'] > 0.5:
        return {'action': 'review', 'reason': 'potentially_explicit'}
    elif results['hate_speech_score'] > 0.7:
        return {'action': 'block', 'reason': 'hate_speech'}
    else:
        return {'action': 'approve'}
```

Copyright Detection (Audio Fingerprinting)

Implementation:

```
func (p *VideoProcessor) checkCopyright(videoId string) (bool, string) {
    // Extract audio
    audioPath := p.extractAudio(videoId)

    // Generate fingerprint using Chromaprint
    fingerprint := chromaprint.Generate(audioPath)

    // Search against known copyrighted music database
    matches, err := p.db.Query(`
        SELECT music_id, title, artist
        FROM music_library
        WHERE fingerprint <-> $1 < 0.2
        ORDER BY fingerprint <-> $1
        LIMIT 1`,
        fingerprint,
    )

    if matches.Next() {
        var musicId, title, artist string
        matches.Scan(&musicId, &title, &artist)

        // Check if license allows use
        license := p.getLicense(musicId)
        if license.Type == "restricted" {
            return true, musicId // Copyright violation
        }
    }

    return false, ""
}
```

Action on copyright match:

```
if (copyrightMatch) {
  // Option 1: Block video
  await db.query('UPDATE videos SET status = $1 WHERE id = $2',
['copyright_blocked', videoId]);

  // Option 2: Mute audio
  await videoProcessor.muteAudio(videoId);
```

```
  // Option 3: Replace with licensed music
  await videoProcessor.replaceAudio(videoId, licensedMusicId);

  // Notify user
  await notificationService.send(userId, {
    type: 'copyright_claim',
    message: 'Your video contains copyrighted music and has been muted.'
  });
}
```

Bot Detection

Signals:

```
async function detectBot(userId: string): Promise<number> {
  const signals = await Promise.all([
    // 1. Posting rate (>100 videos/day = suspicious)
    checkPostingRate(userId),

    // 2. Watch patterns (no human watches 1000 videos/day)
    checkWatchPatterns(userId),

    // 3. Engagement rate (bots don't like/comment)
    checkEngagementRate(userId),

    // 4. Device fingerprint (multiple accounts from same device)
    checkDeviceFingerprint(userId),

    // 5. Profile completeness (bots have sparse profiles)
    checkProfileCompleteness(userId)
  ]);

  // Weighted score
  const botScore =
    signals[0] * 0.3 +
    signals[1] * 0.25 +
    signals[2] * 0.2 +
    signals[3] * 0.15 +
    signals[4] * 0.1;

  if (botScore > 0.8) {
    await redis.sadd('suspected_bots', userId);

    // Shadow ban (videos not shown in feed)
    await db.query('UPDATE users SET shadow_banned = true WHERE id = $1', [userId]);
  }

  return botScore;
}
```

Rate Limiting

Tiered limits:

```
const RATE_LIMITS = {
  video_upload: {
    free: 3,        // 3 uploads per day
    verified: 10,   // 10 uploads per day
    creator: 50     // 50 uploads per day
  },
  follow: {
    free: 200,      // 200 follows per day
    verified: 500
  },
  like: {
    free: 500,      // 500 likes per day
    verified: 2000
  }
};

async function checkRateLimit(userId: string, action: string): Promise<boolean> {
  const userTier = await getUserTier(userId);
  const limit = RATE_LIMITS[action][userTier];

  const key = `rate:${userId}:${action}:${getDateString()}`;
  const current = await redis.incr(key);
  await redis.expire(key, 86400); // 24 hours

  if (current > limit) {
    logger.warn('Rate limit exceeded', {userId, action, current, limit});
    return false;
  }

  return true;
}
```

Age Verification

Requirement: Comply with COPPA (Children's Online Privacy Protection Act)

```
async function verifyAge(userId: string, birthdate: Date): Promise<void> {
  const age = calculateAge(birthdate);

  if (age < 13) {
    // COPPA: Under 13 not allowed
    await db.query('UPDATE users SET status = $1 WHERE id = $2', ['age_restricted',
userId]);
    throw new Error('User must be 13 or older');
  }

  if (age < 18) {
```

```
    // Teen account: restricted features
    await db.query('UPDATE users SET account_type = $1 WHERE id = $2', ['teen',
userId]);

    // Restrictions:
    // - Private account by default
    // - No DMs from non-followers
    // - No appearance in search
    // - Limited data collection
  }
}
```

Data Privacy & GDPR

Right to deletion:

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

  // 2. Delete videos
  const videos = await db.query('SELECT id, original_url, transcoded_urls FROM
videos WHERE user_id = $1', [userId]);
  for (const video of videos) {
    // Delete from S3
    await s3.deleteObject(video.original_url);
    for (const url of Object.values(video.transcoded_urls)) {
      await s3.deleteObject(url);
    }

    // Delete from DB
    await db.query('DELETE FROM videos WHERE id = $1', [video.id]);
  }

  // 3. Delete engagement data
  await db.query('DELETE FROM video_engagement WHERE user_id = $1', [userId]);
  await db.query('DELETE FROM follows WHERE follower_id = $1 OR followee_id = $1',
[userId]);

  // 4. Delete from caches
  await redis.del(`user:${userId}`);
  await redis.del(`user:embedding:${userId}`);
  await redis.del(`feed:candidates:${userId}`);

  // 5. Delete from ML index
  await pinecone.deleteByMetadata({ user_id: userId });

  // 6. Delete from MongoDB (watch history)
  await mongo.deleteMany('watch_events', { user_id: userId });
```

```
  // 7. Async cleanup (CDN purge, backups)
  await kafka.publish('user.deleted', { userId });
}
```

--- END OF PASS 2 ---