

Design YouTube - System Design Interview



ASHISH PRATAP SINGH

JAN 30, 2025 • PAID



108



2



3

Share

With over **2.5 billion** monthly active users, **YouTube** is the second most visited website in the world—trailing only Google.



As a **video-sharing platform**, it enables users to upload, watch, and interact with video content, while handling hundreds of millions of daily visitors, managing petabytes of data, and ensuring real-time video delivery across the globe.

In this article, we'll explore the **system design of a large-scale video streaming service like YouTube** that can accommodate hundreds of millions of daily users and billions of views, all while maintaining low latency and high availability.

We'll walk through every step of the design—from requirements and high-level architecture to database and API design—before diving deep into core use cases.

The concepts covered here are equally applicable to other large-scale video platforms such as **Netflix** and **Prime Video**.



1. Requirements Gathering

Before diving into the design, let's outline the functional and non-functional requirements.

Functional Requirements

- Users should be able to **upload** video files.
- Uploaded videos must be **transcoded** into multiple resolutions (e.g., 240p, 360p, 720p, 1080p) to support different network conditions and devices.
- Users should be able to **stream** videos in real-time with **adaptive bitrate streaming** to adjust quality based on network conditions.
- Users can **search** for videos by title, tags, or description.
- Users can **like** and **comment** on videos.
- Users should be able to create and subscribe to **channels**.

Non-Functional Requirements:

1. **Scalability:** The system should support millions of concurrent users and thousands of video uploads per minute.
2. **High Availability:** Core features like video upload, playback, and search should have minimal downtime.
3. **Low Latency:** Fast video streaming with minimal buffering and near-instantaneous search results.
4. **Durability:** Video files must be stored reliably, with redundancy mechanisms to prevent data loss due to hardware failures.
5. **Cost Efficiency:** Optimize storage and bandwidth costs.

2. Capacity Estimation

Assumptions:

- Daily Active Users (DAU): 10 million
- Upload Rate: ~100,000 videos/day
- Average Videos Watched per User per Day: 5 videos
- Average Video Size: 500 MB.
- Metadata Size per Video: 1 KB.

Storage Estimation:

- Daily Storage for Videos: $100,000 \text{ videos / day} * 500 \text{ MB / video} = 50 \text{ TB / day}$
- Daily Video Metadata Storage: $100,000 * 1\text{KB} = 100\text{MB / day}$

Network Bandwidth Estimation:

- Daily Video Consumption: $10 \text{ million users} * 5 \text{ videos/user} = 50 \text{ million views/day}$
- Daily Bandwidth Requirements (without compression & caching) : $50 \text{ million views} * 500 \text{ MB / day} = 25 \text{ PB / day}$

Given the high storage and bandwidth requirements, leveraging **cloud-based services** is the most practical approach:

1. **Content Delivery Network (CDN)**: To cache frequently accessed videos closer to users and reduce latency.
2. **Blob Storage (e.g., AWS S3)**: To store video files reliably with redundancy.

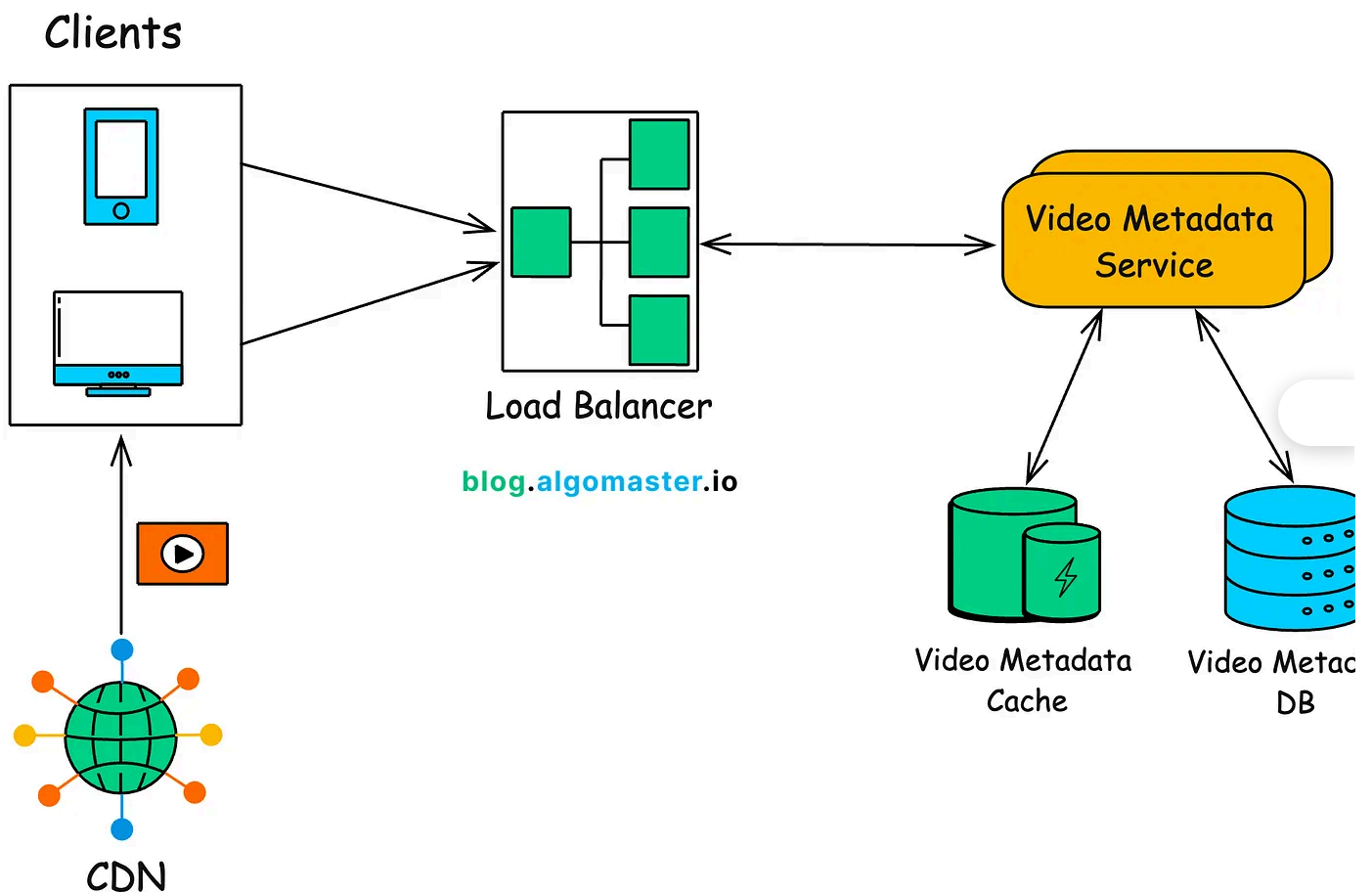
3. High Level Design



We can break the architecture of YouTube into two primary components:

- **Video Streaming** – Handles video playback, and delivery.
- **Video Upload & Processing** – Manages user uploads, transcoding, and metadata storage.

3.1 Video Streaming Architecture



Components:

Clients (Web, Mobile, Smart TVs)

- Users interact with the platform via web browsers, mobile apps, and smart applications.



- The client applications handle video playback, user interactions (likes, comments, subscriptions), and UI rendering.
- They communicate with backend services through an **API Gateway** or **Load Balancer**.

Load Balancer / API Gateway

- Acts as the single entry point for all client requests.
- Distributes incoming traffic across multiple service instances to ensure **high availability** and **scalability**.
- Enforces **rate limiting**, **authentication**, and **authorization** before forwarding requests to downstream services.

Video Metadata Service

- Stores and manages video metadata (e.g., title, description, tags, owner ID, upload timestamp).
- Exposes APIs for querying and updating video details.
- Typically backed by a **relational database** (PostgreSQL/MySQL) for structured metadata storage and a **caching layer** (e.g., Redis) to support fast retrieval of frequently queried videos.

Content Delivery Network (CDN)

- Caches and serves video segments from geographically distributed edge nodes.
- Reduces latency and offloads bandwidth from the **origin storage** by bringing content closer to users.

Video Streaming Workflow:

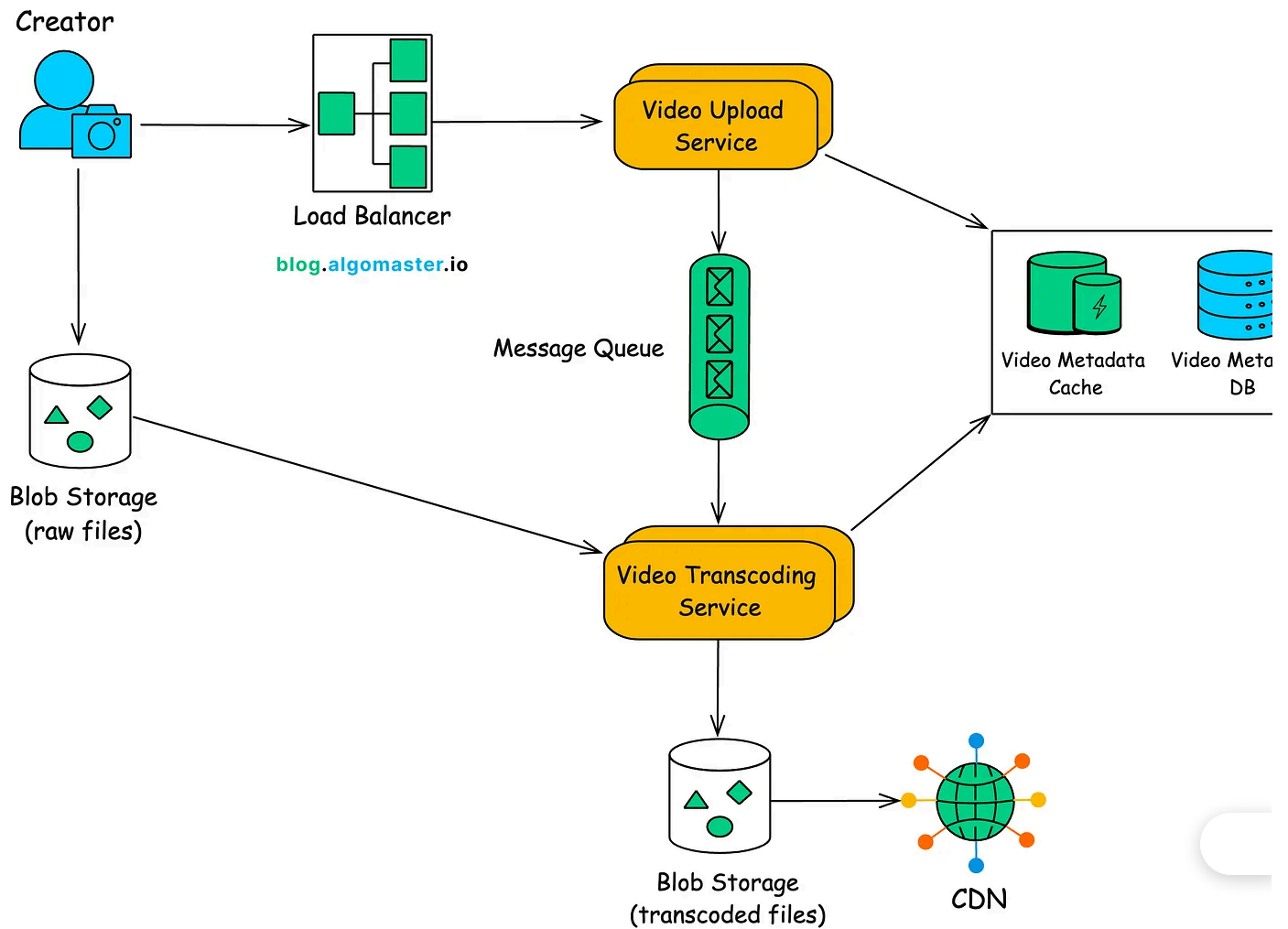
1. The user clicks on a video thumbnail or opens a video page.



2. The client app (on web, mobile or TV) sends a request to the **Video Metadata Service** (via the API Gateway or Load Balancer) to fetch:
 - Video metadata (title, description, thumbnail, channel info)
 - A **streaming manifest URL** (e.g., HLS `.m3u8` or DASH `.mpd` file), which contains links to video segments stored in CDN.
3. The video player (e.g., HTML5 player, Android/iOS native player, Smart TV app) downloads the manifest file.
4. This manifest file contains references to video segments in multiple resolution and bitrates, allowing **adaptive streaming** based on the user's internet speed.
5. The **CDN**, which maintains copies of the video stored at various locations worldwide, serves the video content. The edge server closest to the user handles the request, ensuring low latency and optimized bandwidth usage.
6. The video begins playing immediately, with the player continuously fetching small video chunks (segments) from the CDN edge server.
7. If a requested segment is not found in the CDN cache, it is:
 - a. Fetched from the origin storage.
 - b. Cached for future requests.
 - c. Delivered to the user in real-time.

3.2 Video Upload & Processing Architecture





Components:

Video Upload Service

- Handles video uploads from users, often using [multi-part uploads](#) for large file
- Generates a [pre-signed URL](#) for direct upload to Object Storage (e.g., AWS S3) from the user's device.
- Creates a new entry in the video metadata database when upload is initiated.
- Sends a transcoding job to a **Message Queue** after a successful upload.

Video Transcoding Service



- Reads raw video files from Blob Storage and encodes it into multiple resolutions/bitrates (e.g., 240p, 480p, 720p, 1080p) for adaptive streaming.
- Generates thumbnails and extracts metadata (e.g., duration, codec).
- Stores transcoded video segments in Object Storage or CDN-backed storage for efficient delivery.

Message Queue

- Decouples video upload from transcoding to ensure scalability.
- Stores **job requests** from the Upload Service and allows the Transcoding Service to process them asynchronously.

Storage

- **Raw Video Storage** – Stores unprocessed uploads in a blob store before transcoding.
- **Processed Video Storage** – Stores transcoded video segments in a **CDN-backed** system.
- **Metadata Storage** – Stores structured video metadata in a relational database.

Video Upload workflow:

1. Initiating the Upload

- A logged-in content creator selects a video file via a frontend client (web, mobile or smart TV interface).
- The client sends an upload request to the **Video Upload Service** via the **API Gateway**.
 - Includes metadata such as `channel_id`, `title`, `description`, `tags`, and other relevant fields.



- The server generates a [pre-signed URL](#) for an **Object Storage** bucket (e.g., AWS S3).
- The client uploads the video **directly to Object Storage** using the pre-signed URL.

2. Upload to Object Storage

- Large files often use **multi-part** or **chunked** uploads. The client splits a large file into smaller “parts” (chunks), each typically ranging from a few MBs to tens of MBs.
 - If the connection drops in the middle of the upload, only the **incomplete chunk** needs to be re-sent rather than re-uploading the entire file from scratch.
 - In many multi-part implementations, each chunk can be **uploaded concurrently** using multiple threads or connections.

3. Metadata Creation & Status Update

- Once the upload call is initiated, the **Upload Service** creates a new record in the **Videos** table with:
 - `video_id` (primary key)
 - `channel_id` (which user or channel this belongs to)
 - `title, description, tags`
 - `status = "processing"` (since it's not yet ready for viewing)
 - `upload_date = now()`
- The **Videos** table record includes a reference or URL to the uploaded file in Object Storage, e.g., `raw_file_url: <https://bucket/raw/12345.mp4>`
- The Upload Service returns a response to the client with the newly created `video_id`.

4. Sending a Transcoding Job

- The **Upload Service** places a message on a **queue** (e.g., RabbitMQ, AWS SQS, Kafka) containing:
 - `video_id`
 - `raw_file_url`
 - `target_resolutions`
 - `target_formats`

Video Transcoding Workflow:

1. Processing the Job

- A **Transcoding Service** worker polls the queue, retrieves the message, and extracts: `raw_file_url`, `video_id`, `target_resolutions` and `target_formats`.
- The worker downloads the raw file from the Object Storage (raw files bucket)

2. Video Transcoding Process

- The worker uses video processing tools (e.g., FFmpeg) to create **adaptive bitrate** variants:
 - **Low** resolution (240p or 360p) - For slow connections.
 - **Standard** resolution (480p or 720p) - For average connections.
 - **High** resolution (1080p or 4K) - For high-bandwidth users.
- Each variant is **split** into small segments (e.g., **2-10 seconds long**) for **adaptive streaming**.
- The worker writes the final video segments and streaming manifests to a “transcoded” bucket in Object Storage or a CDN-backed storage path, e.g.:
 - `https://cdn.provider.com/videos/11111/720p/....`



- The worker may also generate a **thumbnail** at this stage (capturing a frame at X seconds into the video).


3. Status Update & Database Sync

- Once transcoding finishes successfully, the Transcoding Service calls an **internal API** (e.g., `PUT /videos/{video_id}/status`) on the **Metadata Service**.
- The **Videos** metadata table record is updated:
 - `status = "live"` (video is now available for streaming).
 - `transcoded_url` fields updated with resolution-based URLs.
 - `thumbnail_url` updated.
- If transcoding **fails**, the worker marks `status = "failed"`, optionally storing an error message.
- Once the transcoded files are in object storage or an origin server, the **CDN** automatically caches content at edge locations to serve playback requests.
- Future **playback requests** are served from CDN edge nodes, reducing origin bandwidth usage and improving streaming performance.

4. Database Design

A large-scale video platform like YouTube requires handling both **structured data** (e.g., user accounts, video metadata, subscriptions) and **unstructured/semistructured data** (e.g., video files, search indexes, logs).

Typically, you'll combine multiple database solutions to handle different workloads

1. **Relational Databases (SQL)** for user profiles, channel data, and video met. 

2. **NoSQL / Key-Value Stores** for high-volume event logs (e.g., watch history) or caching frequently accessed data.
3. **Search Indexes** for keyword-based searches (title, tags, description).
4. **Object Storage** for the actual video files and thumbnail images.

4.1 Relational Tables

Given the structured nature of user profiles, video metadata, subscriptions, and relationships, a relational database (like **PostgreSQL** or **MySQL**) is often well-suited.

Schema:



users	videos	channels
<u>user_id</u> : bigint	<u>video_id</u> : bigint	<u>channel_id</u> : bigint
username: varchar	channel_id: bigint	user_id: varchar
email: varchar	title: varchar	channel_name: varchar
profile_pic_url: varchar	description: text	description: text
country: varchar	tags: text	subscriber_count: bigint
hashed_password: varchar	visibility: enum {public, private, unlisted}	created_at: date
created_at: date	status: enum {processing, live, failed}	
	duration: int	
	upload_date: date	
	view_count: bigint	
	like_count: bigint	
	raw_file_url: varchar	
	thumbnail_url: varchar	
	resolution_urls: jsonb	

comments	subscriptions
<u>comment_id</u> : int	<u>user_id</u> : bigint
video_id: varchar	channel_id: bigint
user_id: text	subscribed_at: date
content: text	
created_at: date	

blog.algomaster.io

- **Users Table:** Stores user account details.
- **Channels Table:** Each user can have one or more **channels**.
- **Videos Metadata Table:** Stores **video metadata**, but not the actual video files.



- **Comments Table:** Stores video comments.
- **Subscriptions Table:** Tracks user subscriptions to channels.

Example resolution_urls JSON format:

```
{  
  "240p": "https://cdn.provider.com/videos/12345/240p.m3u8",  
  "480p": "https://cdn.provider.com/videos/12345/480p.m3u8",  
  "720p": "https://cdn.provider.com/videos/12345/720p.m3u8",  
  "1080p": "https://cdn.provider.com/videos/12345/1080p.m3u8"  
}
```

4.2 Search Indexes

For fast video searches, we can store video metadata in Elasticsearch.

Example Record:

```
{  
  "video_id": 12345,  
  "title": "Learn System Design",  
  "description": "This is an in-depth guide...",  
  "tags": ["system design", "architecture", "scalability"],  
  "views": 500000,  
  "upload_date": "2025-01-30T10:15:00Z"  
}
```

5. API Design

Below is a RESTful API design covering some of the core use cases.

5.1 Video Upload & Processing APIs

Initiate Video Upload

POST /videos/upload
Authorization: Bearer jwt_token

Request (JSON or multipart/form-data):

```
{  
  "channel_id": 45678,  
  "title": "How to Learn System Design",  
  "description": "A complete guide to mastering system design",  
  "tags": ["system design", "architecture", "scalability"]  
}
```

Response:

```
{  
  "video_id": 456,  
  "status": "processing",  
  "upload_url": "https://s3.example.com/upload/456.mp4"  
}
```

Complete Upload & Start Processing

POST /videos/{video_id}/process
Authorization: Bearer jwt_token

Response:

```
{  
  "message": "Video processing started."  
}
```

5.2 Video Metadata & Streaming APIs

Get Video Details

GET /videos/{video_id}

Response:

```
{  
  "video_id": 456,  
  "title": "How to Learn System Design",  
  "description": "A complete guide to mastering system design",  
  "category": "Education",  
  "tags": ["system design", "architecture", "scalability"],  
  "views_count": 150000,  
  "likes_count": 1200,  
  "comments_count": 300,  
  "upload_date": "2025-01-30T10:15:00Z",  
  "status": "live",  
  "thumbnail_url": "https://cdn.example.com/thumbnails/456.jpg",  
  "resolution_urls": {  
    "240p": "https://cdn.example.com/videos/456/240p.m3u8",  
    "480p": "https://cdn.example.com/videos/456/480p.m3u8",  
    "720p": "https://cdn.example.com/videos/456/720p.m3u8"  
  }  
}
```

Stream Video (HLS/DASH)




```
GET /videos/{video_id}/stream
```

Response:

- Redirects to the CDN URL for HLS/DASH manifest file.

5.3 Search APIs

Search Videos

```
GET /search?query=system+design&limit=10&offset=0&sort=views
```

Response:

```
{
  "results": [
    {
      "video_id": 456,
      "title": "How to Learn System Design",
      "description": "A complete guide to mastering system design",
      "thumbnail_url": "https://cdn.example.com/thumbnails/456.jpg",
      "channel_id": 45678,
      "view_count": 150000,
    },
    ...
  ],
}
```

5.4 Engagement APIs

Like a Video



```
POST /videos/{video_id}/like
Authorization: Bearer jwt_token
```

Comment on a Video

```
POST /videos/{video_id}/comments
Authorization: Bearer jwt_token
```

Request:

```
{
  "content": "Great video! Very informative."
}
```

Response:

```
{
  "comment_id": 987,
  "message": "Comment added successfully."
}
```

Get Comments on a Video

```
GET /videos/{video_id}/comments?limit=10&offset=0
```

Response:

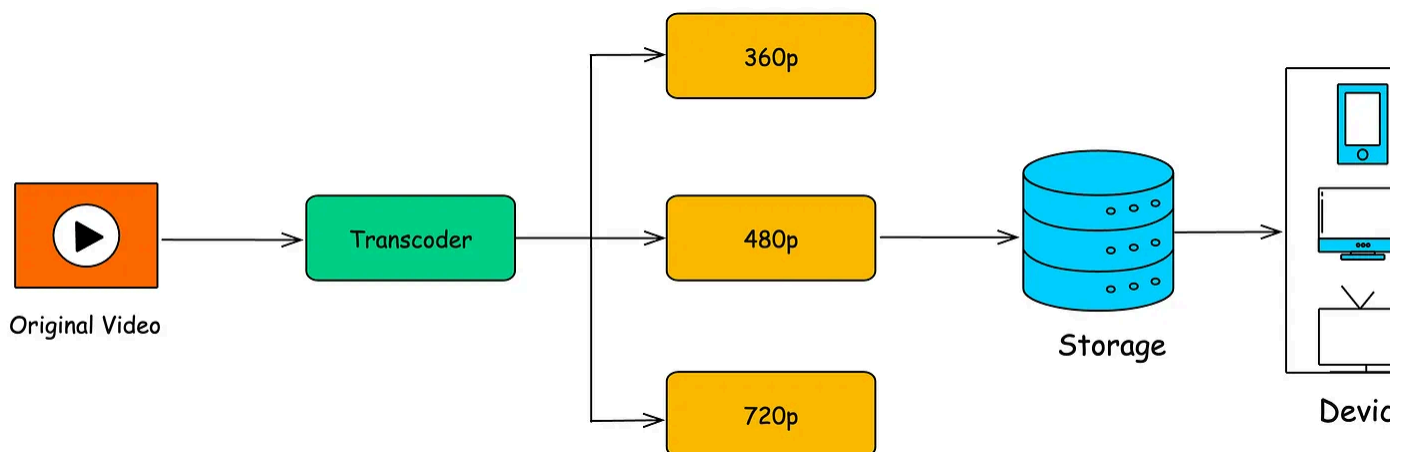



```
{  
  "comments": [  
    {  
      "comment_id": 987,  
      "user": "john_doe",  
      "content": "Great video! Very informative.",  
      "created_at": "2025-01-30T10:30:00Z"  
    }  
  ]  
}
```

6. Diving Deep into Core Use-Cases

6.1 Video Transcoding

Video transcoding is the process of converting a raw video file into multiple formats, resolutions, and bitrates to ensure smooth playback across different devices, network speeds, and screen sizes.



This process is critical for video streaming platforms like YouTube and Netflix.  videos need to be accessible on a wide range of devices—from high-end desktops w

fast internet to mobile phones on limited bandwidth.

To achieve smooth and adaptive streaming, we use **Adaptive Bitrate Streaming (ABR)**. This allows the client to dynamically switch between different quality levels based on the user's network speed, ensuring an optimal viewing experience.

How Video Transcoding Works

When a video is uploaded, it undergoes **post-processing** to convert it into a streamable format.

This process is often executed in steps using a **pipeline** to produce the final output.

Step 1: Uploading & Storing the Raw Video

- A user uploads a raw video file (e.g., MP4, MOV, AVI).
- The raw file is stored in **Object Storage** (e.g., AWS S3) directly from the user's device.
- Metadata is recorded in the Videos Metadata Table, including:
 - `video_id`
 - `duration`
 - `status = "processing"`

Step 2: Job Dispatching via Message Queue

- Once the raw file is uploaded, the **Upload Service** sends a message to a **Message Queue** (e.g., Kafka).
- The **Transcoding Service** workers poll messages from the queue and process jobs asynchronously.
- Example Message Sent to Queue



```
{  
  "video_id": 456,  
  "raw_file_url": "s3://video-uploads/12345.mp4",  
  "target_resolutions": ["240p", "480p", "720p", "1080p"],  
  "target_formats": ["HLS", "DASH"]  
}
```

- The target formats are [HLS](#) (HTTP Live Streaming) and [DASH](#) (Dynamic Adaptive Streaming over HTTP), the two most widely used adaptive streaming protocols.

Step 3: Decoding the Raw Video

- The raw video file is split into smaller segments (e.g., 10-second chunks) using a tool like FFmpeg.
- Each segment is transcoded separately into multiple formats.
- Segments can be transcoded parallelly using multiple FFmpeg instances or worker nodes.


Step 4: Encoding into Multiple Resolutions

- Each video is encoded into multiple bitrates/resolutions (e.g., 240p, 360p, 480p, 720p, 1080p).
- A “master” manifest references these resolutions, letting the player decide which one to fetch.

Step 5: Generating Video Segments for Streaming

- Each resolution is split into short segments (2-10 seconds) for HLS and DASH adaptive streaming.

- **Why Segment-Based?**

Short segments allow **quick adaptation**. If the network changes mid-video, the next segment can be requested at a different bitrate/resolution with minimal playback disruption. 

Step 6: Generating Thumbnails

- A thumbnail image is created as a preview.
- The user can later update the thumbnail manually.

Step 7: Storing Transcoded Files in Object Storage

- The transcoded videos, thumbnails, and manifest files are stored in **Object Storage** (e.g., AWS S3).
- Example storage structure:

```
s3://video-platform/videos/12345/240p.m3u8  
s3://video-platform/videos/12345/480p.m3u8  
s3://video-platform/videos/12345/thumbnail.jpg
```

Step 8: Updating the Database & CDN

- The **Video Metadata Table** is updated with:
 - Transcoded URLs for different resolutions
 - Thumbnail URL
 - `status = "live"` (video is ready to stream).
- The **CDN** caches video segments for low-latency streaming and fast access.

6.2 Video Streaming

The Video Streaming Component is responsible for **orchestrating video playback** and **adaptive streaming**.

Rather than pushing large media files through a single backend endpoint, modern platforms provide **manifests** describing how to fetch the media segments.



Clients then download video segments directly from the CDN, which offloads huge bandwidth demands from your core application servers.

Below is the typical **workflow** to support smooth video streaming:

1. User Initiates Playback

- a. The user clicks on a video thumbnail or opens a video page on a web browser, mobile app, or smart TV app.

2. Fetching Video Metadata

- a. The client app sends a request to the Video Metadata Service via the API Gateway or Load Balancer to retrieve:
 - i. Video metadata (title, description, thumbnail, channel info).
 - ii. A streaming manifest URL (e.g., HLS `.m3u8` or DASH `.mpd` file), which guides the player in fetching and playing the video.

3. Downloading the Manifest File

- a. The client makes an HTTP GET request to the CDN to download the manifest file.
- b. This manifest file contains links to video segments in multiple resolutions and bitrates (generated in the video transcoding process), allowing **adaptive streaming** based on the user's internet speed.

4. Retrieving Video Segments & Adaptive Bitrate

- a. The video player chooses an **initial bitrate** (often mid or low) and downloads the first segment from the CDN.
- b. It continuously monitors **download speed** and **buffer level**. If downloads arrive quickly, it may switch to higher quality; if the connection slows, it drops to lower quality.

5. Continuous Playback



- a. The player sequentially requests segments: `segment_01.ts`, `segment_02.ts`, etc., or corresponding `.m4s` fragments for DASH.
- b. It buffers data, rendering frames to the user in a near real-time fashion.

6. End of Playback / Seeking

- a. When playback concludes or the user seeks (e.g., moves the timeline forward) the player calculates which segments to request next and continues the same segment download process.

6.3 Video Search

A video platform's **Search Engine** enables users to quickly find videos by keywords, titles, descriptions, tags, or even transcribed captions.

This requires:

1. **Indexing:** Transforming raw metadata into a data structure optimized for retrieval (e.g., inverted indexes).
2. **Query Parsing:** Understanding user queries, possibly with keyword-based or NLP based improvements.
3. **Ranking/Scoring:** Determining how relevant each video is to the user's query, often incorporating signals like view count, recency, or user preferences.

Architecture and Components:

- Whenever a video is uploaded or updated, the system pushes the new/updated metadata to a **search indexing service** (e.g., Elasticsearch, Solr, or a custom system).
- This process might be asynchronous—meaning a short delay between upload and availability in search results.



- **Inverted Index:** Each word maps to a list of video IDs where it appears, enabling fast lookups.
- The **Search Service** queries the inverted index with relevant filters (e.g., language, upload date).
- A **scoring/ranking** algorithm sorts results by relevance, which might incorporate additional signals like popularity or watch time.
- If the user is logged in, search could factor in watch history, subscriptions, or topic preferences.

7. Addressing Non-Functional Requirements

7.1 Scalability

Horizontal Scaling

Decompose services into microservices (Upload Service, Transcoding Service, Metadata Service). Deploy microservices using Kubernetes (K8s) + Docker for auto-scaling.

Use Load Balancers to distribute traffic across multiple service instances

Metadata Sharding

A single metadata database cannot handle millions of video records efficiently.

Shard Metadata Across Multiple Databases:

Shard by `video_id` to distribute the load across multiple database instances.

Example:



- video_1 → shard_1 (DB1)
- video_2 → shard_2 (DB2)
- video_1000000 → shard_n (DB_n)

Use a Shard Lookup Service:

- Maintains a mapping of video IDs to shards.
- Clients query the lookup service before fetching metadata.

Cache Frequently Accessed Metadata:

- Use Redis or Memcached to cache video metadata and avoid excessive database reads.

Distributed & Parallel Transcoding

Transcoding a single video file sequentially is slow and inefficient, especially for large video files.

Break Video into Segments & Transcode in Parallel:

- Split the video into smaller independent segments (e.g., 2-10 seconds each).
- Distribute segments across multiple transcoding workers.
- Merge the transcoded segments back into a single stream.

7.2 Availability

Multi-Region Deployment

- Deploy compute, storage, and databases across multiple availability zones (AZs) and regions.
- Example: AWS (US-East, US-West, Europe, Asia).



Redundant Storage & Replication

- Store videos in multi-region object storage (AWS S3, Google Cloud Storage) with geo-redundancy.
- Enable replication to a backup region.

Database Replication & Failover

- Use primary-replica databases (PostgreSQL, MySQL, CockroachDB) for failover
- In case of failure, the load balancer routes requests to a healthy replica.

7.3 Cost Efficiency

Video Deduplication

Users might upload duplicate videos, wasting storage and processing power.

Identify and Deduplicate Videos Before Processing

- **Hashing-Based Deduplication**
 - Generate a unique hash (SHA-256, MD5) from the video file.
 - If a hash matches an existing video, store a reference instead of a new copy.
- **Perceptual Hashing (PHash) for Similar Videos**
 - Detect similar but slightly altered videos (e.g., re-uploads with minor edits).
- **Machine Learning Deduplication**
 - Use ML models to compare video/audio frames and detect near-duplicates.

Multi-Tiered Storage (Hot, Warm, Cold)

- Frequently watched videos → Hot Storage (SSD-backed, fast access).
- Less popular videos → Warm Storage (HDD-backed, moderate latency).
- Rarely accessed videos → Cold Storage (AWS Glacier, Google Coldline).

Hope you enjoyed reading this article.

If you found it valuable, hit a like ❤️ and if you have any questions or suggestions, leave a comment.



108 Likes • 3 Restacks

← Previous

Next →

Discussion about this post

Comments Restacks



Write a comment...



Kannan Feb 25

❤️ Liked by Ashish Pratap Singh

Crystal clear explanation... informative!!

♡ LIKE (1) 💬 REPLY



1 reply by Ashish Pratap Singh

1 more comment...



© 2025 Ashish Pratap Singh • [Privacy](#) • [Terms](#) • [Collection notice](#)
[Substack](#) is the home for great culture

