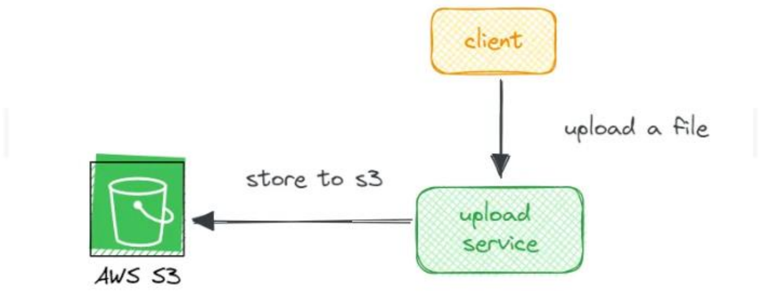
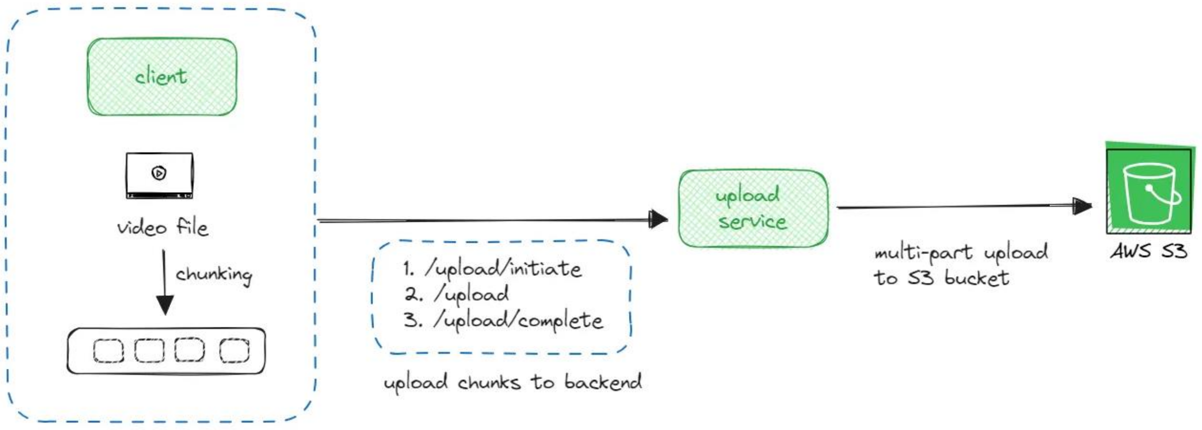
Youtube Design:

Building the project:

- Upload Service:
  - Client video upload requests will be sent to upload service. Upload service will store the video into S3 bucket.



- OAuth Login:
  - Implement OAuth login with NextAuth to allow logging in using Google, Facebook, etc.
- Uploading videos is different than uploading images because videos are much larger in size. So chunking videos on client side before uploading is required.
- Saving chunks to S3 using multi-part upload:
  - S3 gives us a provision to store the chunks of the video together
  - multipart upload→ S3 saves the chunks as an entire new video
  - initiate- include uploadID in every part upload. this uploadID will be unique to the entire uploaded file
  - upload parts- send parts/chunks along with uploadID and also send the chunk_index (part number). In response u get ETag (entity tag)
  - multipart completion- when all parts uploaded, S3 will create an object by concatenating the parts in ascending order based on the part number.

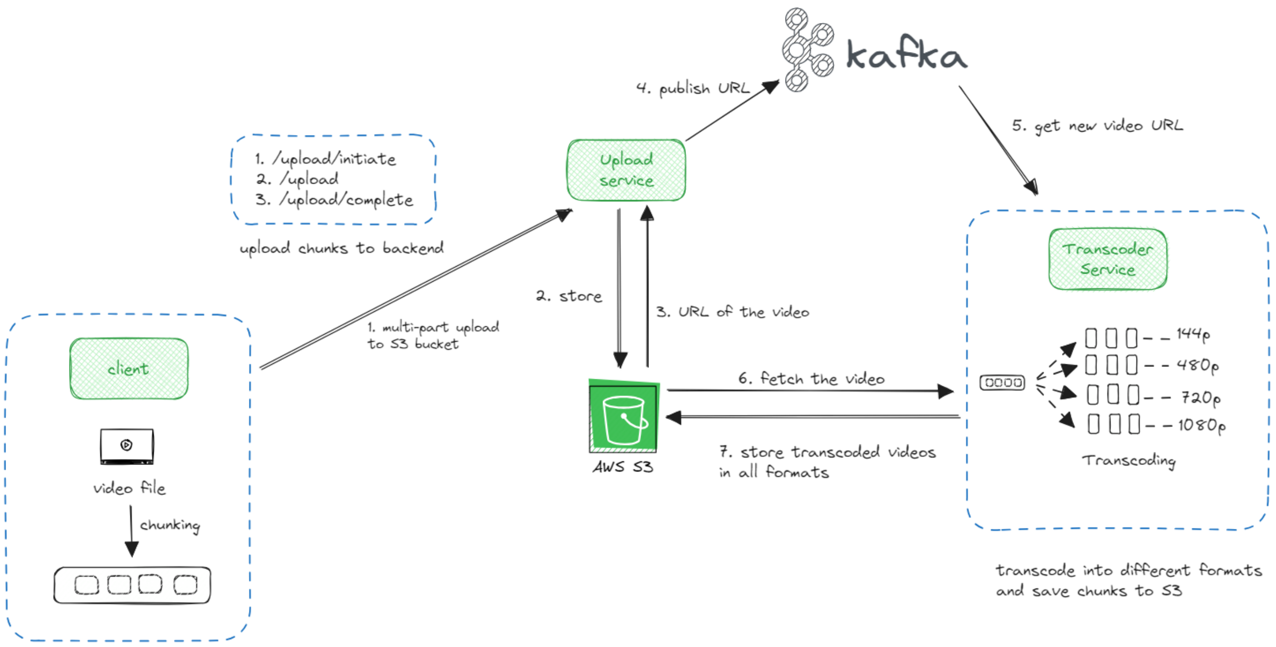


Multi-part upload of Video

- The chunks can be parallelly uploaded which makes the uploading very fast.

Adaptive streaming:

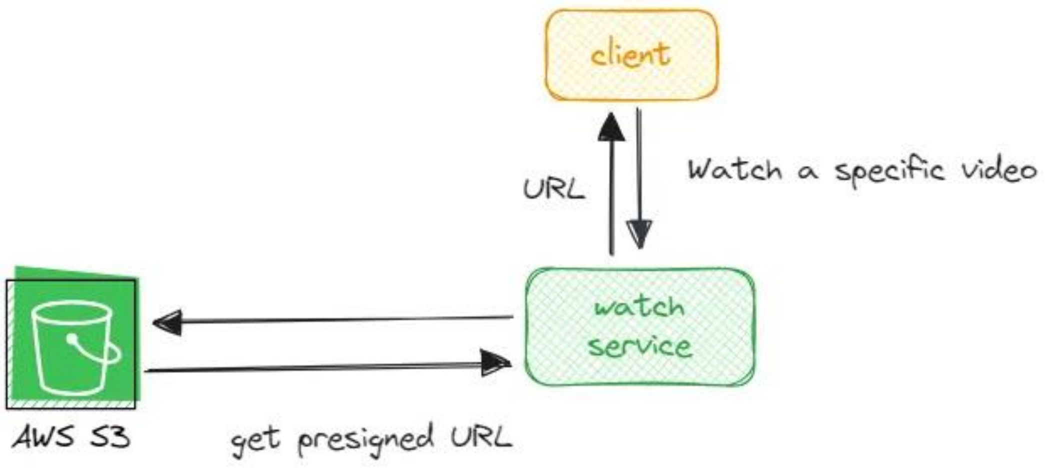
- In adaptive streaming, based on the bandwidth, we change the resolution of the video so that we minimize buffering and provide a better user experience. For this next step, each video needs to be transcoded and stored into different formats to allow adaptive streaming.
- So, while playing the video, we fetch the correct chunk (based on available bandwidth) of the correct resolution and play on the client side.
- The transcoding can be done separately and parallelly so we can leverage distributed MQ like Kafka to decouple the upload and transcode service.
- This way we can asynchronously convert a given video into different formats and store it to S3 bucket.



1. Upload the video chunks to upload service
2. Store the video leveraging S3 multi-part upload to S3
3. Get the URL of the newly stored video.
4. Publish this URL to a transcoding topic on Kafka to decouple the application, allow separately adding and removing instances of services based on load.
5. Consume the URL from the topic
6. Get the video from S3 using the consumed URL and transcode the video to different formats
7. Save all the transcoded files to S3.

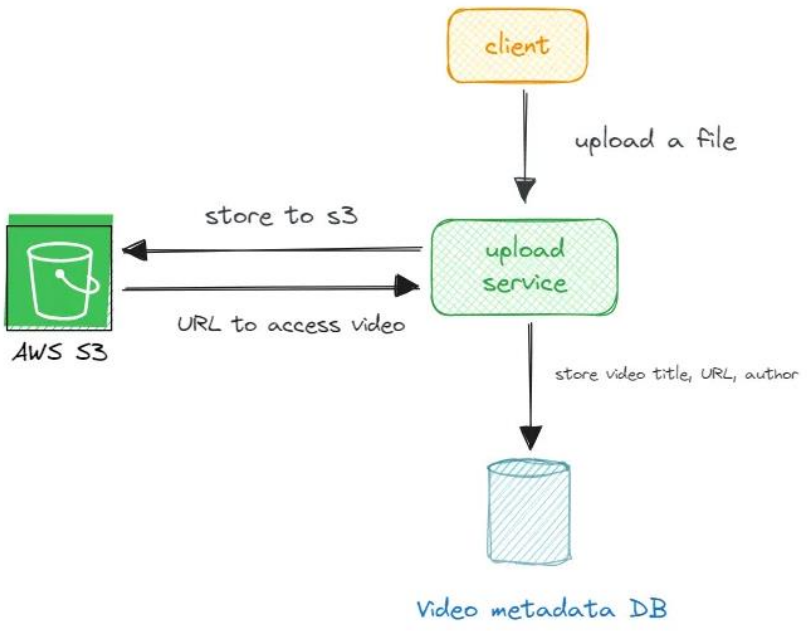
Watch Service:

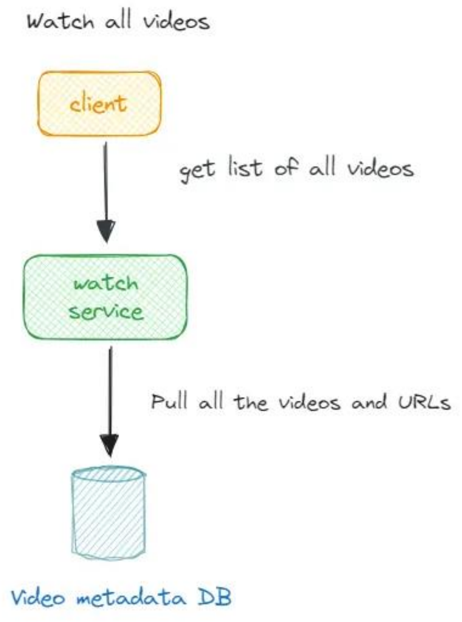
- Once a video is stored to S3, watch service will fetch all the videos and show on the landing screen. We can also get a specific video using watch service and play on client.
- We can leverage presigned URLs to allow access to certain videos only after sign in.



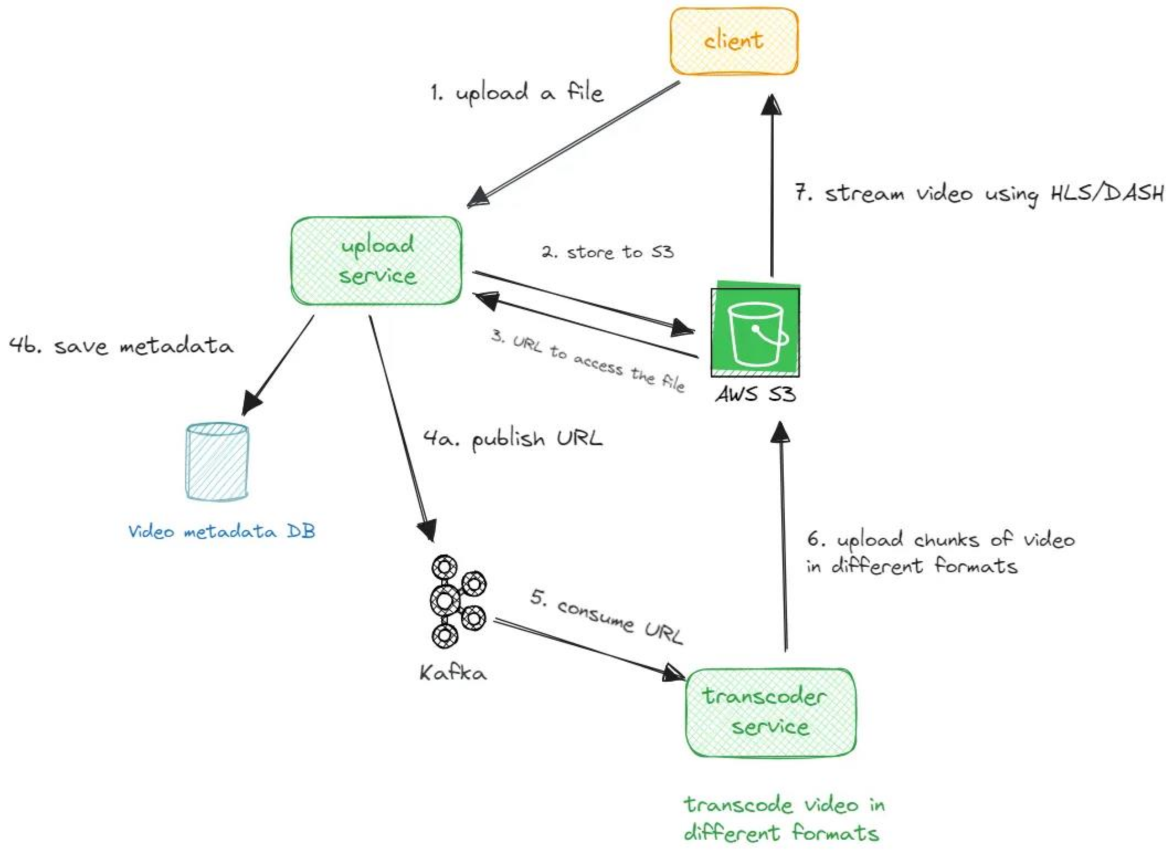
Database:

- We can use a relational database to store the Video metadata like title, author, description, likes, etc. In the project we leveraged PostgreSQL to store video metadata.





Putting all the pieces together-



1. Client uploads a file.
2. Upload service uploads to S3 (multi-part upload of chunks)

3. S3 returns the URL to access the video

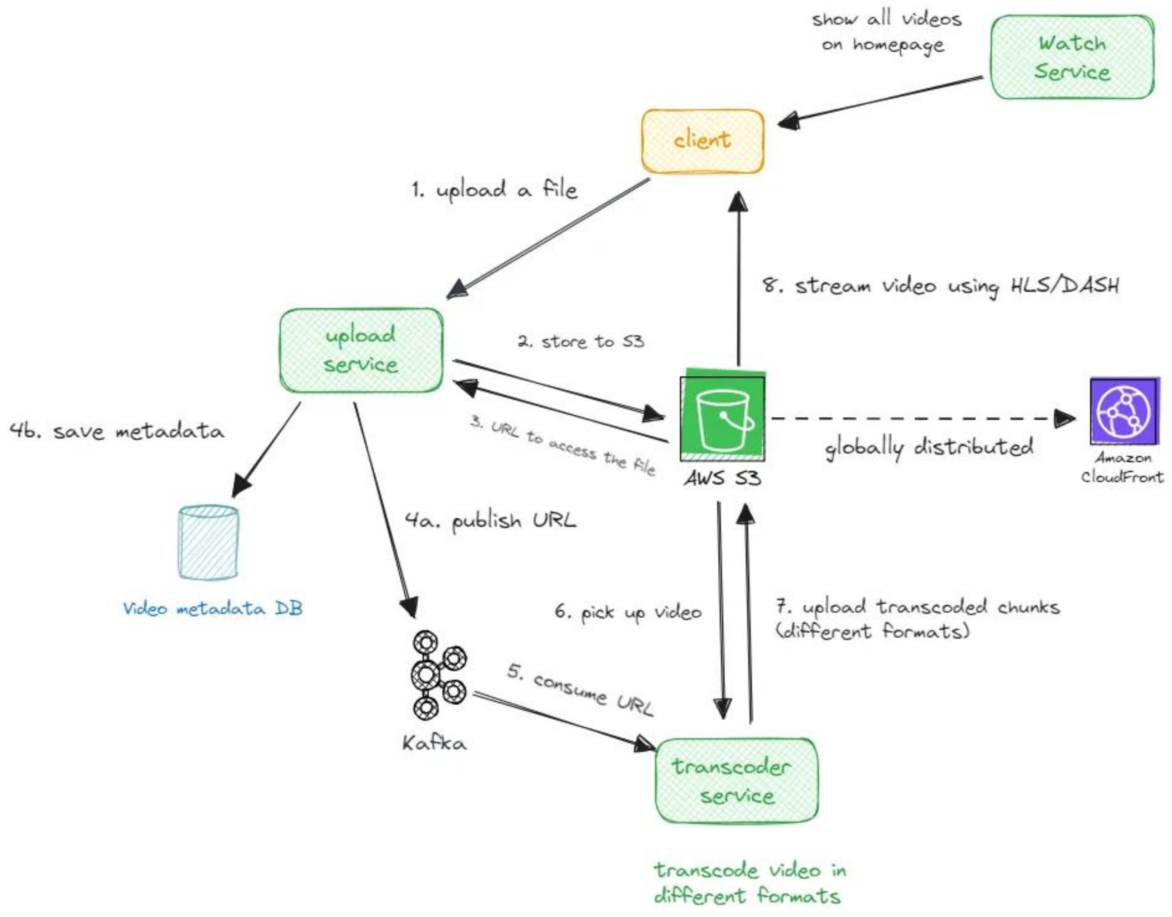
4a. The URL is published to a transcoding topic on Kafka

4b. Update metadata DB with new record.

5. Transcoder service gets the URL, gets the video from S3 bucket, encodes into multiple different resolutions using HLS/DASH.
6. All the files are stored on S3 along with configuration files.
7. While playing a video, the client will fetch the relevant files based on the bandwidth availability and use ABS (Adaptive Bitrate Streaming) to stream the content.

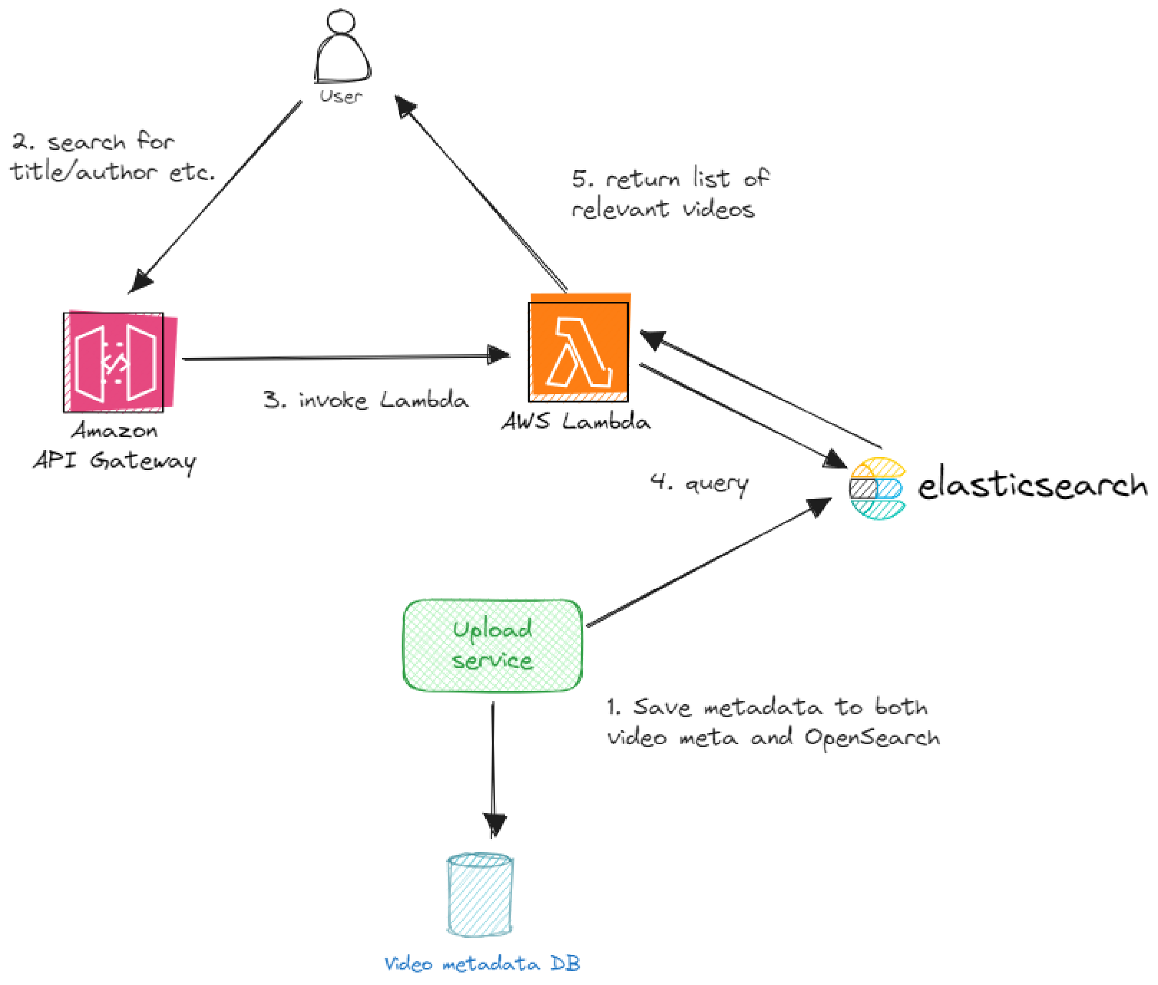
CDN-

- The videos can be accessed via a CDN to reduce latency on different locations.
- So we allow S3 bucket access through CDN and use AWS Cloudfront.



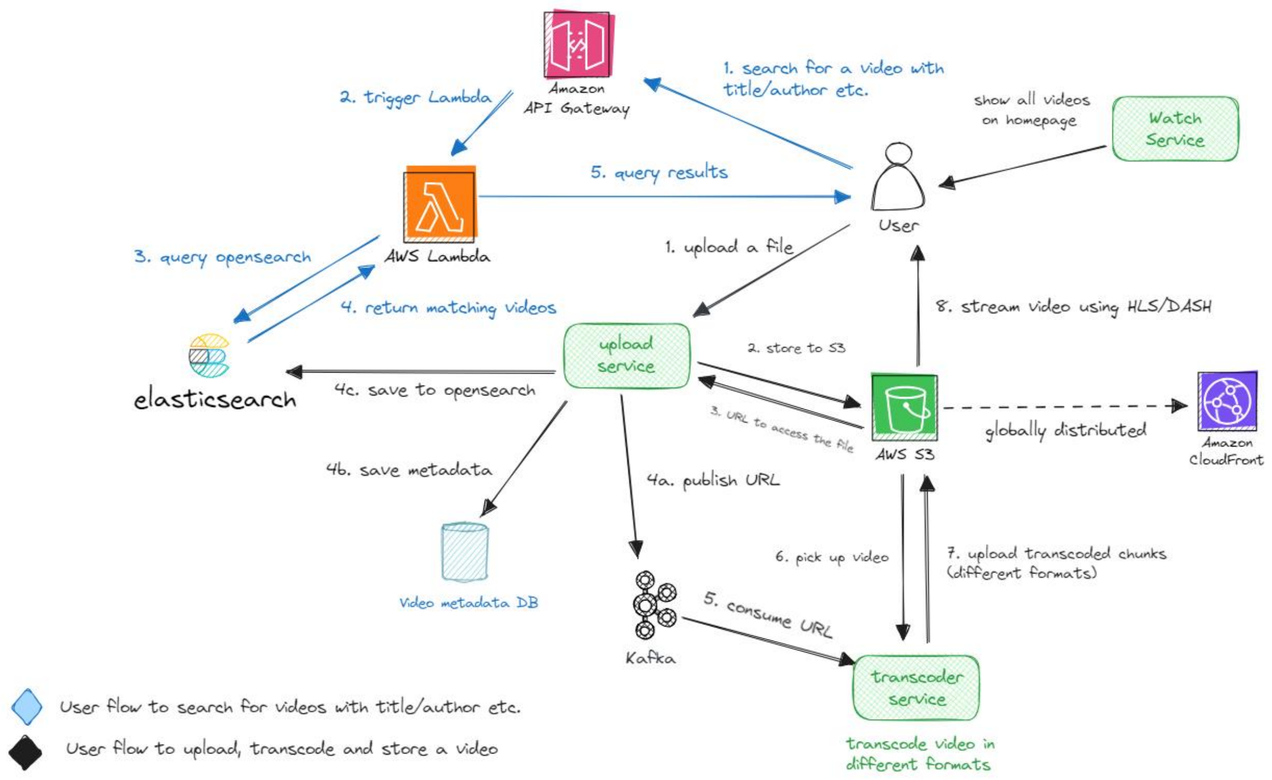
Youtube Search functionality:

- We want to allow users to search for relevant videos on title or channel name or a person's name.
- With increase in demand, it is important to think about optimizing the query process for all the relevant links.
- For this purpose, we can use ElasticSearch/OpenSearch. It is build on top of Apache Lucene which uses inverted indices to make the searching process extremely faster.
- The backend logic for querying can be written in a separate service but we can also use AWS Lambda functions which can easily run the required logic in a serverless manner.
- An API Gateway can be created which exposes an endpoint to trigger Lambda function. So client hits the API Gateway, API Gateway invokes Lambda function, Lambda function brings all the relevant data from the OpenSearch and user can see relevant videos quickly.


User
2. search for title/author etc.
5. return list of relevant videos
Amazon API Gateway
3. invoke Lambda
AWS Lambda
4. query
elasticsearch
Upload service
1. Save metadata to both video meta and OpenSearch
Video metadata DB

1. On each upload, the upload service pushes the data to video metadata DB as well as OpenSearch
2. Client searches for a title or author or any other parameter to the API gateway endpoint
3. API Gateway invokes AWS Lambda
4. Lambda function queries the OpenSearch for relevant videos
5. Return the list of all videos matching the searched query.

Combining the whole picture:


Amazon API Gateway
1. search for a video with title/author etc.
2. trigger Lambda
show all videos on homepage
Watch Service
AWS Lambda
5. query results
User
3. query opensearch
4. return matching videos
1. upload a file
8. stream video using HLS/DASH
elasticsearch
upload service
2. store to S3
4c. save to opensearch
3. URL to access the file
AWS S3
globally distributed
Amazon CloudFront
4b. save metadata
4a. publish URL
Video metadata DB
6. pick up video
7. upload transcoded chunks (different formats)
Kafka
5. consume URL
transcoder service
transcode video in different formats
User flow to search for videos with title/author etc.
User flow to upload, transcode and store a video