Design a video streaming platform similar to Netflix or Youtube

Table of Contents:

Problem Statement

Design a video streaming platform similar to Netflix where content creators can upload their video content and viewers are able to play video on different devices. We should also be able to store user statistics of the videos such as the number of views, video watched duration, and so forth.

Gathering Requirements

Functional Requirements

1. Users should be able to upload videos.
2. Users should be able to share and view videos.
3. Users should be able to change video quality.
4. Users should be able to perform searches based on video titles.
5. The system should also keep the count of likes, dislikes, comments, and views to present these numbers to users.

Non-Functional Requirements

1. The system should be highly reliable, and any video uploaded should not be lost.
2. The system should be highly available. Consistency can take a hit (in the interest of availability); if a user doesn't see a video for a while, it should be fine.
3. Users should have a real-time experience while watching videos and should not feel any lag.

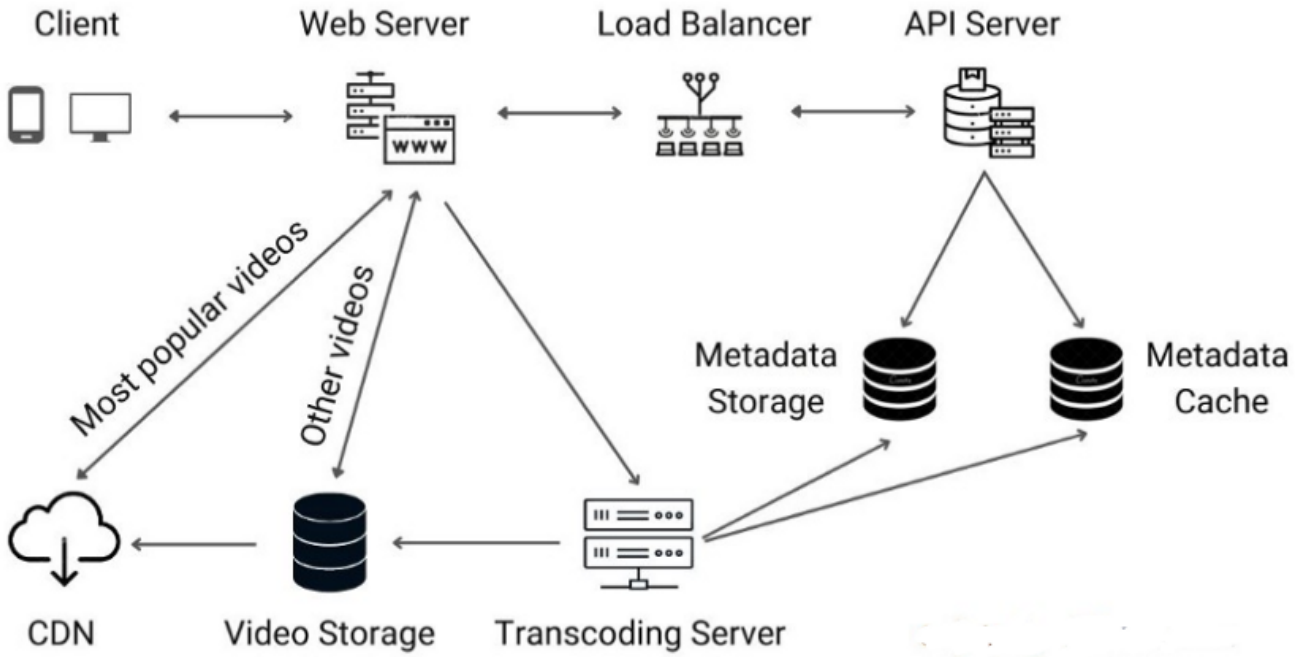
Capacity estimation and System Constraints

The following estimates are based on many assumptions:

1. Let's assume total users = 2 billion.
2. Daily active users = 400 million.
3. The number of videos watched/day/user = 5
4. Total video views/day: 400 million * 5 = 2 billion views/day

5. **Total video views/second:** 2 billion / 86400 = 23k views/sec
6. **Videos Upload per day:** Our Video streaming service would be **view-heavy (read-heavy)** system. Suppose upload to view ratio (read-to-write ratio) is 200:1, then total video upload/day = 2 billion/200 = 10 million videos upload/day.
7. **Total storage needed/day:** Suppose the average video size is 100 MB. Total storage needed/day = 20 million * 100 MB = 2000 TB/day = 2 PB/day. Here we ignore video compression and replication, which would change our estimates.

High-level design



The entire system can be divided into the following components:

- **Users/Client:** There will primarily be two kinds of users in the system: **content creators** who upload video content, and **viewers** who watch the video.
- **Load Balancer**: We use this to evenly distributes requests among API servers.
- **API server:** We use this to handle all requests except video streaming. This includes feed recommendations, generating video upload URL, updating metadata database and cache, user signup, etc. This server talks to various databases to get all the data.

- **Web Server:** This is used for handling incoming requests from the client. Based on the type of request, it routes the request to the application server or transcoding server.

- **Content Distributor Network (CDN):** It's responsible for storing the content in the locations which are geographically closest to users. This significantly enhances the user experience as it reduces the network and geographical distances that our video bits must travel during playback. It also reduces the overall demand on upstream network capacity by a great extent.

Open Connect is the global customized CDN for Netflix that delivers Netflix TV shows and movies to members world-wide. This essentially is a network of thousands of Open Connect Appliances (OCAs) which store encoded video/image files and is responsible for delivering the playable bits to client devices. OCAs are made up of tuned hardware and software which are deployed on the ISP site and custom tailored to provide optimal customer experience.

- **Control Plane:** This component will be responsible for uploading new content which will eventually be distributed across the CDNs. It will also be responsible for things such as file storage, sharding, data storage and interpretation of relevant telemetry about the playback experience. The major microservices in this component are listed below.

CDN Health Checker Service: *This microservice will be responsible to periodically check health of the CDN service, learn about the overall playback experience, and work towards optimizing it.*

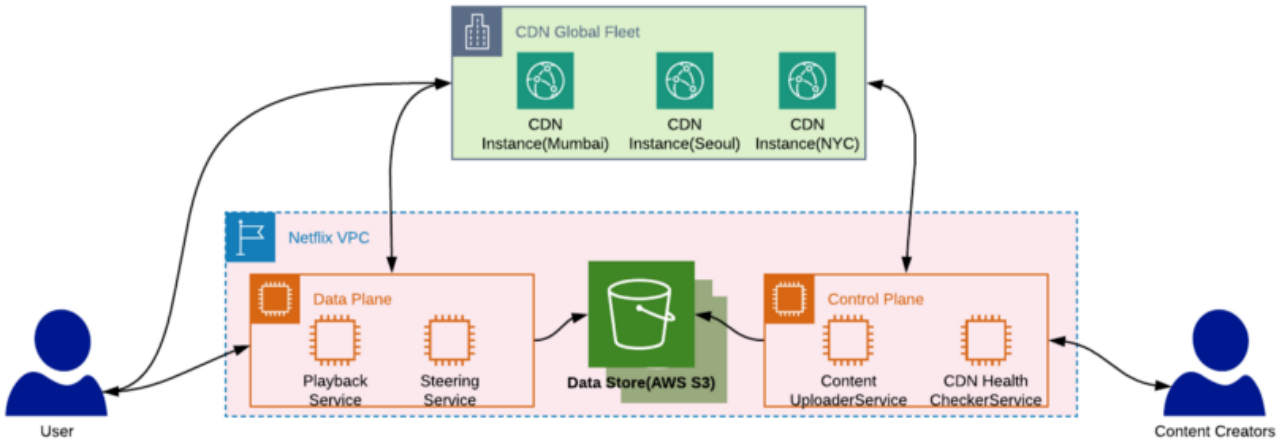
Content Uploader Service: *This microservice will consume the content provided by content generators and distribute it across the CDNs to ensure robustness and optimal playback experience. It will also be responsible for storing the metadata of video content in data-storage.*

- **Data Storage:** The video metadata (title, description, etc.) is persisted in the data storage. We will also persist the subtitle information in the optimal database and we will also store all the information about videos like title, file path in the system, uploading user, total views, likes, dislikes, etc. It will also be used to store all the video comments. We will also store users' information, e.g., name, email, address, etc.

- **Data Plane:** This is the component with which the end-users interact while playing video content. This component gets requests from different media streaming platforms (TV, mobile, tablet, etc.) and returns the URLs of the CDNs from which the requested files can be served. It will comprise two major microservices.

Playback Service: *This micro-service is responsible for determining the specific files which are required for serving a playback request.*

Steering Service: *This service determines the optimal CDN urls from which the requested playback can be fetched from.*



HLD of Netflix

In the image above, we have shown the bird's eye view of the overall system which should be able to meet all the in-scope requirements. The details of each of the component interactions are listed below.

- Step 1: The content creators upload the video content to the control plane.
- Step 2: The video content gets uploaded on the CDN which is placed geographically closer to end-users.
- Step 3: The CDN reports the status to the control plane such as health metrics, what files they have stored, the optimal BGP routes, and so forth.
- Step 4: The video metadata and the related CDN information gets persisted in the data storage.

- Step 5: A user on the client device places the request to play a particular title (TV show or movie).
- Step 6: Playback service determines the files which are required to playback a specific title.
- Step 7: The Steering service picks the optimal CDNs from which the required files can be fetched. It generates the URLs of such CDNs and provides them back to the client device.
- Step 8: The client device requests the CDNs to serve the requested files.
- Step 9: CDN serves the requested files to the client device which gets rendered to the users.

Data Model

The service would be read-heavy, so we will focus on building a system that can retrieve videos quickly. We can expect our read:write ratio to be 200:1, which means for every video upload, there are 200 video views. Before deciding which databases to use we can revise database concepts.

Video metadata storage — MySql

Videos metadata can be stored in a SQL database. The following information should be stored with each video:

- VideoID
- Title
- Description
- Size
- Thumbnail
- Uploader/User
- Total number of likes
- Total number of dislikes

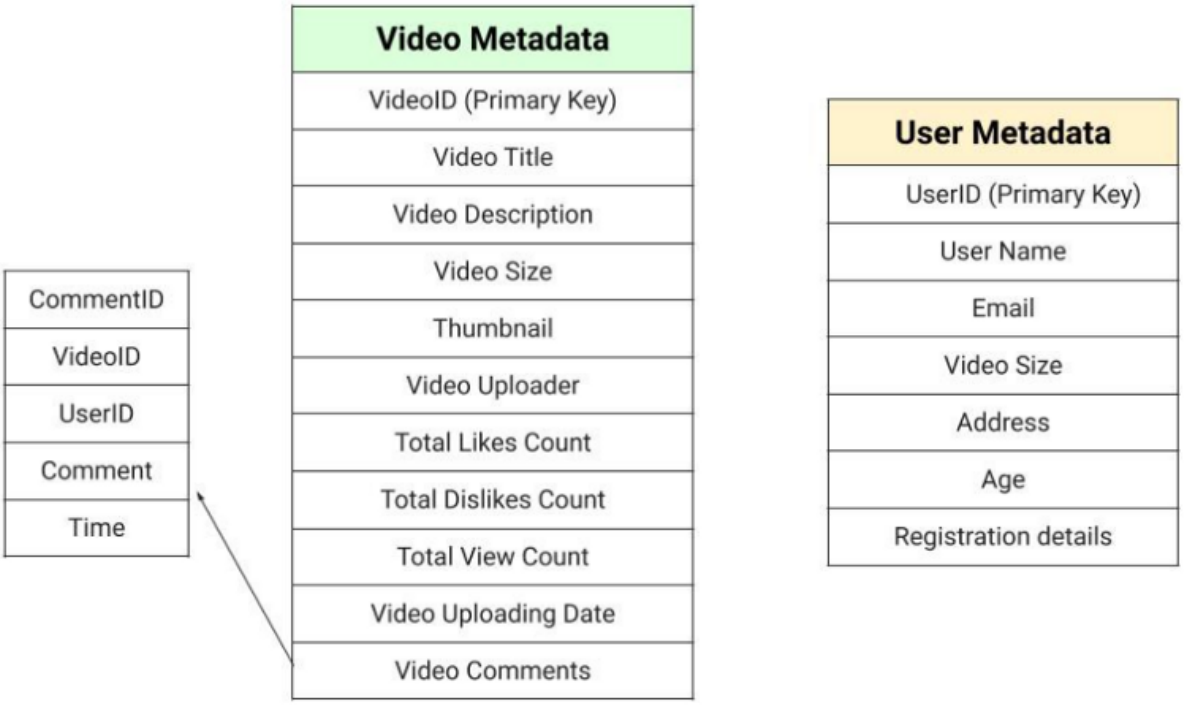
- Total number of views

For each video comment, we need to store the following information:

- CommentID
- VideoID
- UserID
- Comment
- TimeOfCreation

User data storage — MySql

- UserID, Name, email, address, age, registration details, etc.



Sub-titles Data Storage: Casandra

We can use a time-series database such as OpenTSDB, which builds on top of Cassandra, to store the sub-titles. We have shown below a snippet of the data-model which can be used to store video sub-titles. In this model(let's call it Media Document), we have provided an event-based representation where each event occupies a time-interval on the timeline.

```
{

“events”: [
{
       “startTime”: T0,
       “endTime”: T1,
       “metadata”: {
              “subtitle”: “Hi there! How are you?”
       }
},
{
       “startTime”: T2,
       “endTime”: T3,
       “metadata”: {
              “subtitle”: “Thanks for asking”
       }
}]

}
```

Where would videos be stored?

Videos can be stored in a distributed file storage system like HDFS or GlusterFS or Amazon S3.

Where would thumbnails be stored?

There will be a lot more thumbnails than videos. If we assume that every video will have five thumbnails, we need to have a very efficient storage system that can serve huge read traffic. There will be two considerations before deciding which storage system should be used for thumbnails:

1. Thumbnails are small files, say, a maximum of 5KB each.
2. Read traffic for thumbnails will be huge compared to videos. Users will be watching one video at a time, but they might be looking at a page with 20 thumbnails of other videos.

Let’s evaluate storing all the thumbnails on a disk. Given that we have a huge number of files, we have to perform many seeks to different locations on the disk to read these files. This is quite inefficient and will result in higher latencies.

Bigtable can be a reasonable choice here as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data. Both of these are the two most significant requirements for our service. Keeping hot thumbnails in the cache will also help improve the latencies and, given that thumbnail files are small in size, we can easily cache a large number of such files in memory.

Detailed Component Design

Control Plane

This component will mainly comprise three modules/microservices: **Content Uploader, CDN Health Checker, and Title Indexer.** We have covered the details of each of these modules in the section below.

Content Uploader

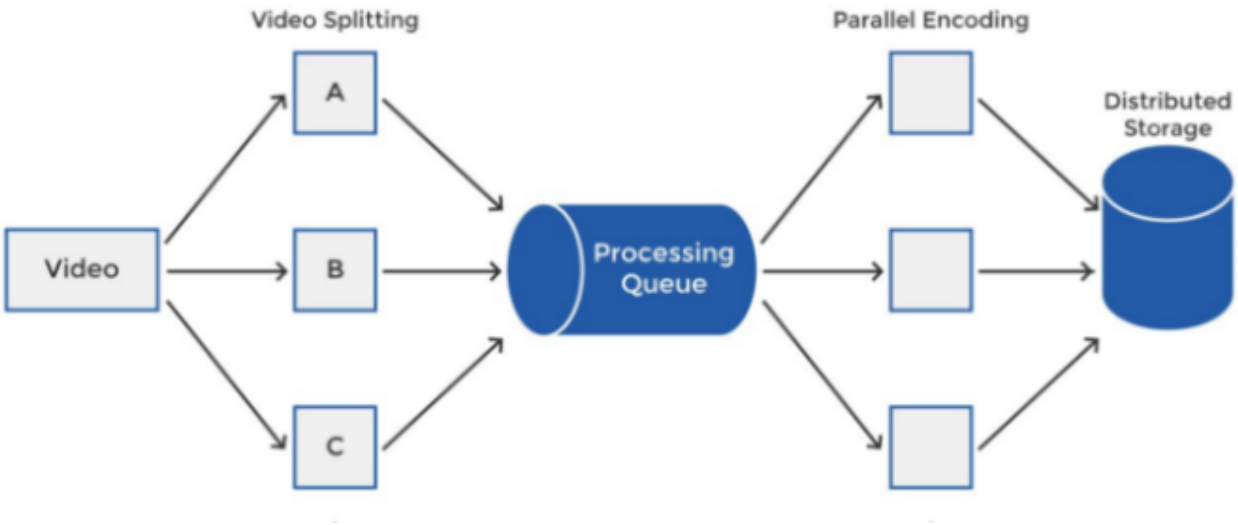
This module is executed when a content creator uploads content. It is responsible for distributing the content on CDN to provide an optimal customer experience.

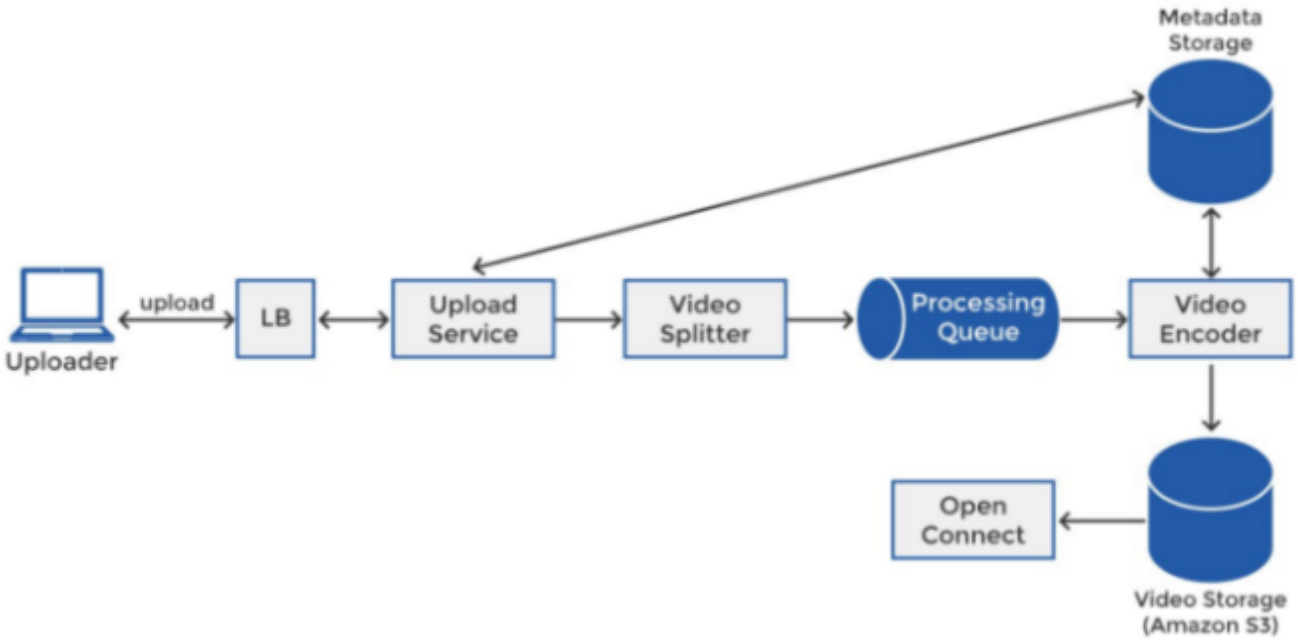
the sequence of operations which gets executed when content creators upload the video content (TV Show or movie).

1. The content creator uploads the raw video content which can be TV Show or movie.
2. **Video Splitter:** Instead of being stored as a single large file, each uploaded video will be stored in several chunks. This is necessary because a content creator can upload a large video. Processing or streaming a single heavy file can be time-consuming. When the video is stored and available to the viewer in chunks, the viewer will not need to download the entire video before playing it. It will request the first chunk from the server, and while that chunk is playing, the client requests for the next chunk so there's a minimum lag between the chunks and the user can have a seamless experience while watching the video.
   The video splitter will divide the video into smaller chunks and put them in the processing queue. As they are de-queued, the chunks will be encoded.
3. A **processing queue** is needed because there are several chunks for each video and Netflix will use several parallel workers to process them. This is made easier by pushing them into the queue. The workers (or the encoders which we will discuss next) will pick

up the tasks from the processing queue, encode them into different formats and store them in the distributed file storage.

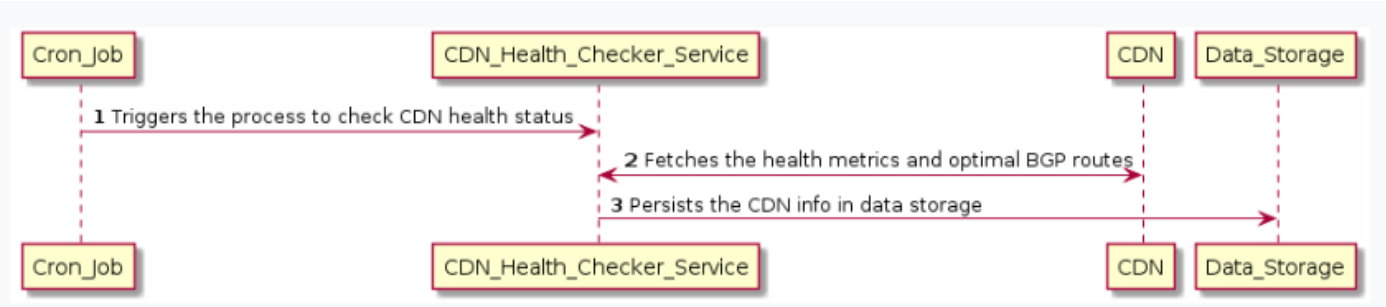
4. The **Video_Encoder or Transcoding Server**: We use this for video transcoding or encoding purposes, which converts a video into multiple formats and resolutions (144p, 240p, 360p, 480p, 720p, 1080p & 4K) to provide the best video streaming for different devices and bandwidth requirements.
5. The encoded file segments are stored in the file storage.
6. The **Video_Distributor** reads the encoded file segments from the distributed file storage system.
7. The **Video_Distributor** distributes the encoded file segments in CDN.
8. The **Video_Distributor** persists the CDN URL links of the videos in the data_storage.





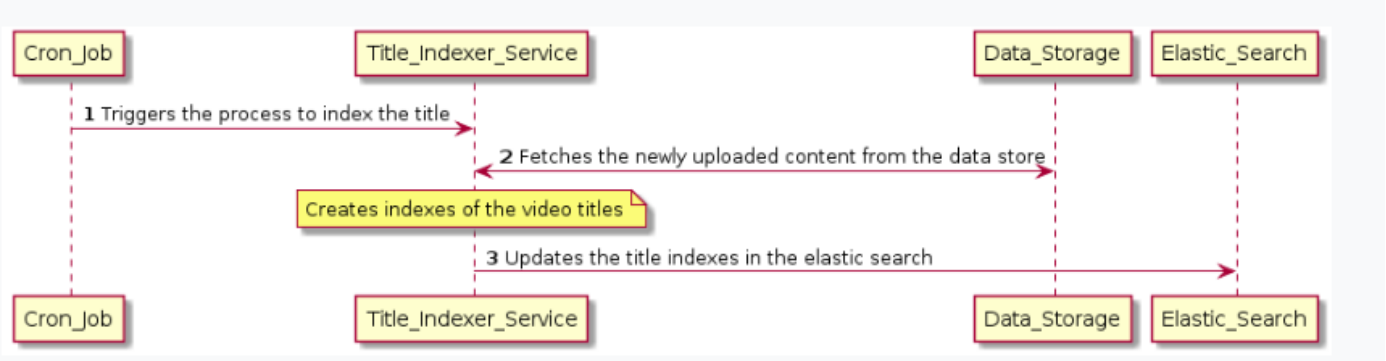
CDN Health Checker

This module ingests the health metrics of the CDNs and persists them in the data storage. This data is used by the data plane to get optimal CDN URLs when users request playback.



Title Indexer

This module is responsible for creating the indexes of the video titles and updates them in the elastic search to enable faster content discovery for end users.



Data Plane

This component will be processing the user requests in real-time and will comprise of two major workflows:

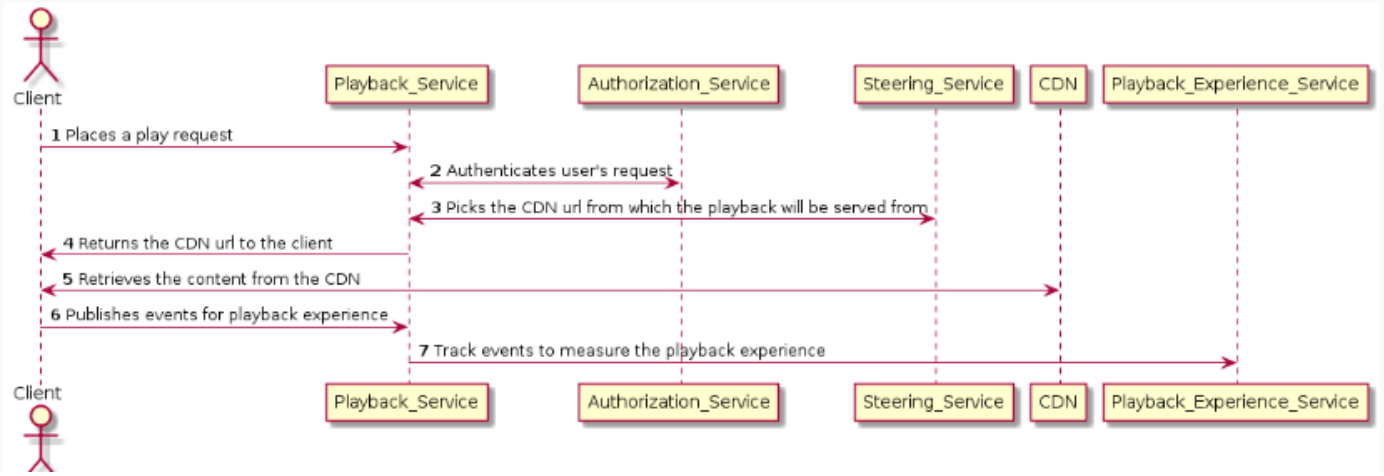
1. **Playback Workflow**
2. **Content Discovery Workflow.**

Playback Workflow

This workflow is responsible for orchestrating operations when a user places a playback request.

It co-ordinates between different microservices such as Authorization Service (for checking user authorization and licensing), Steering Service (for deciding the best playback experience) and Playback Experience Service (for tracking the events to measure playback experience).

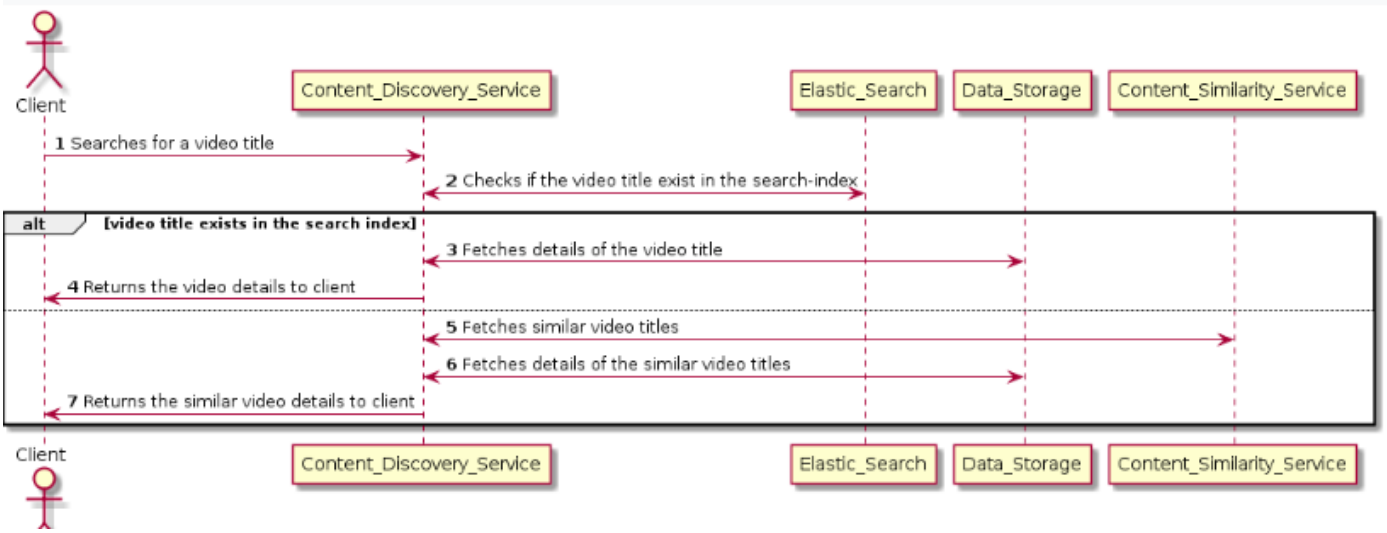
Steering Service ensures the best customer experience by finding the most optimal CDN url based on user request such as user's device, bandwidth and so forth. The orchestration process will be handled by the Playback_Service as shown in the image below.

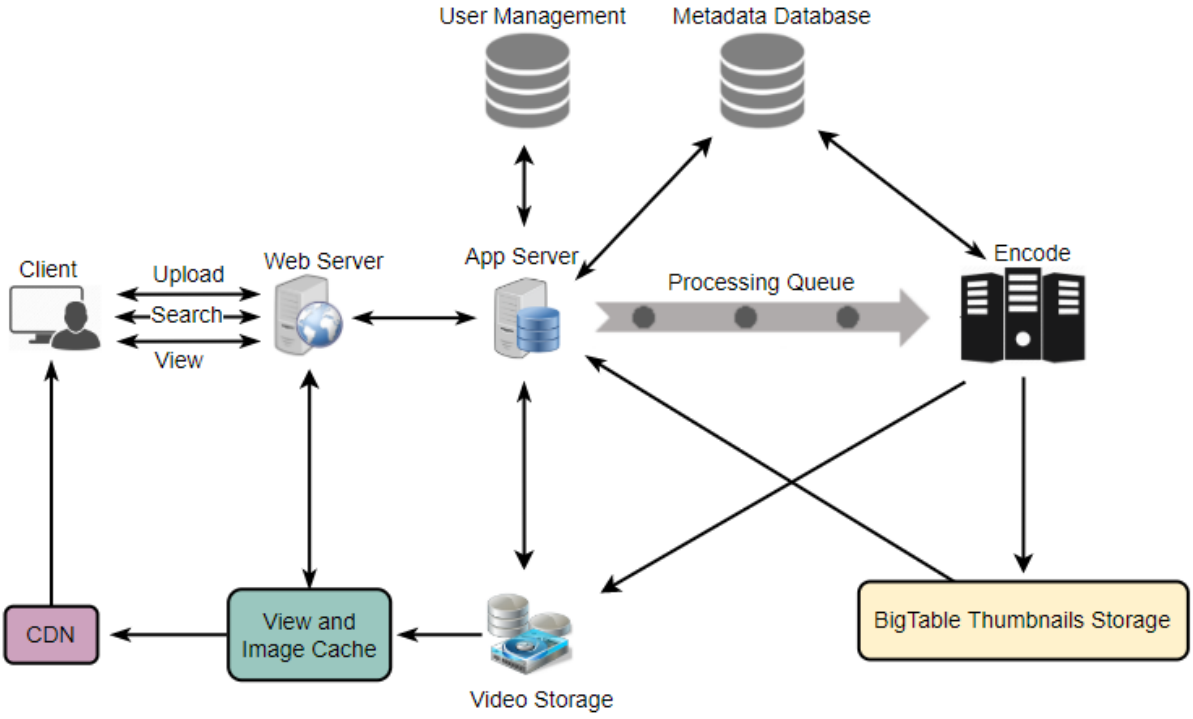


Content Lookup Workflow

This workflow is triggered when user searches for a video title and comprises of two microservices: **Content Discovery Service and Content Similarity Service.**

The Content Discovery Service gets invoked when user requests for the video title. On the other hand, the Content Similarity Service returns the list of similar video title if the exact video title doesn't exist in our data-store.





Detailed component design of Youtube

Replication in MySQL DB:

Approaches to data replication:

- Single leader (Multi Slave Architecture)
- Multi leader (Multi-Master)
- leaderless

We need to discuss various approaches to replication. And which one do we need to use and why?

Out of the above all as we know , MySQL uses Master-Slave architecture in which leader replicates data to its followers using ***write-ahead log*** *mechanism.*

Master-Slave Architecture:

To efficiently manage read traffic We should segregate our read traffic from write traffic. Since we will have multiple copies of each video, we can distribute our read traffic on different servers. For metadata, we can have primary-secondary configurations where writes will go to primary first and then get applied at all the secondaries.

Read requests will be routed to slave replicas parallelly to reduce the load on the master. This could help us to increase the read throughput.

Such a design may cause staleness in data from the read replica. How? Let's think!

Suppose we performed a write operation by adding a new video, then its metadata would be first inserted in the master. Now before this new data gets updated to the slave, a new read request came. At this point, slaves would not be able to see it and return stale data to the user.

For example, this inconsistency may create a difference in view counts for a video between the master and the replica. But this can be okay for the user if there is a slight inconsistency (for a short duration) in the view count.

But here is a problem with this approach:

Since we have a huge number of new videos every day and our read operation is extremely high, the master-slave architecture will suffer from replication lag. On another side, update operation causes cache misses, which go to disk where slow I/O causes slow replication. **Now the critical question is: how can we improve the performance of the read/write operations?** Think!

Metadata Sharding:

Sharding is one of the ways of scaling a relational database along with the master-slave replication. In this process, we distribute our data across multiple machines so that we can perform read/write operations efficiently.

Now instead of a single master handling the write requests, it could be done on various sharded machines and increase the write performance. We can also create separate replicas to improve redundancy and throughput.

Sharding can increase the system complexity and we need an abstract system to handle the scalability and manageability challenges. This requirement led to the development of **Vitess**!

Vitess supports the following types of sharding operations:

- **Horizontal sharding:** Splitting or merging shards in a sharded keyspace
- **Vertical sharding:** Moving tables from an unsharded keyspace to a different keyspace.

Vitess allows you to choose the type of sharding scheme by the choice of your **Primary Vindex** for the tables of a shard.

Primary Vindex Can be chosen as :

1. **UserID**
   We can try storing all the data for a particular user on one server. While storing, we can pass the UserID to our hash function, which will map the user to a database server where we will store all the metadata for that user's videos.

This approach has a couple of issues:

- What if a user becomes popular? There could be a lot of queries on the server holding that user; this could create a performance bottleneck. This will also affect the overall performance of our service.
- Over time, some users can end up storing a lot of videos compared to others. Maintaining a uniform distribution of growing user data is quite tricky.

2. VideoID

Our hash function will map each VideoID to a random server where we will store that Video's metadata. To find videos of a user, we will query all servers, and each server will return a set of videos. A centralized server will aggregate and rank these results before returning them to the user. This approach solves our problem of popular users but shifts it to popular videos.

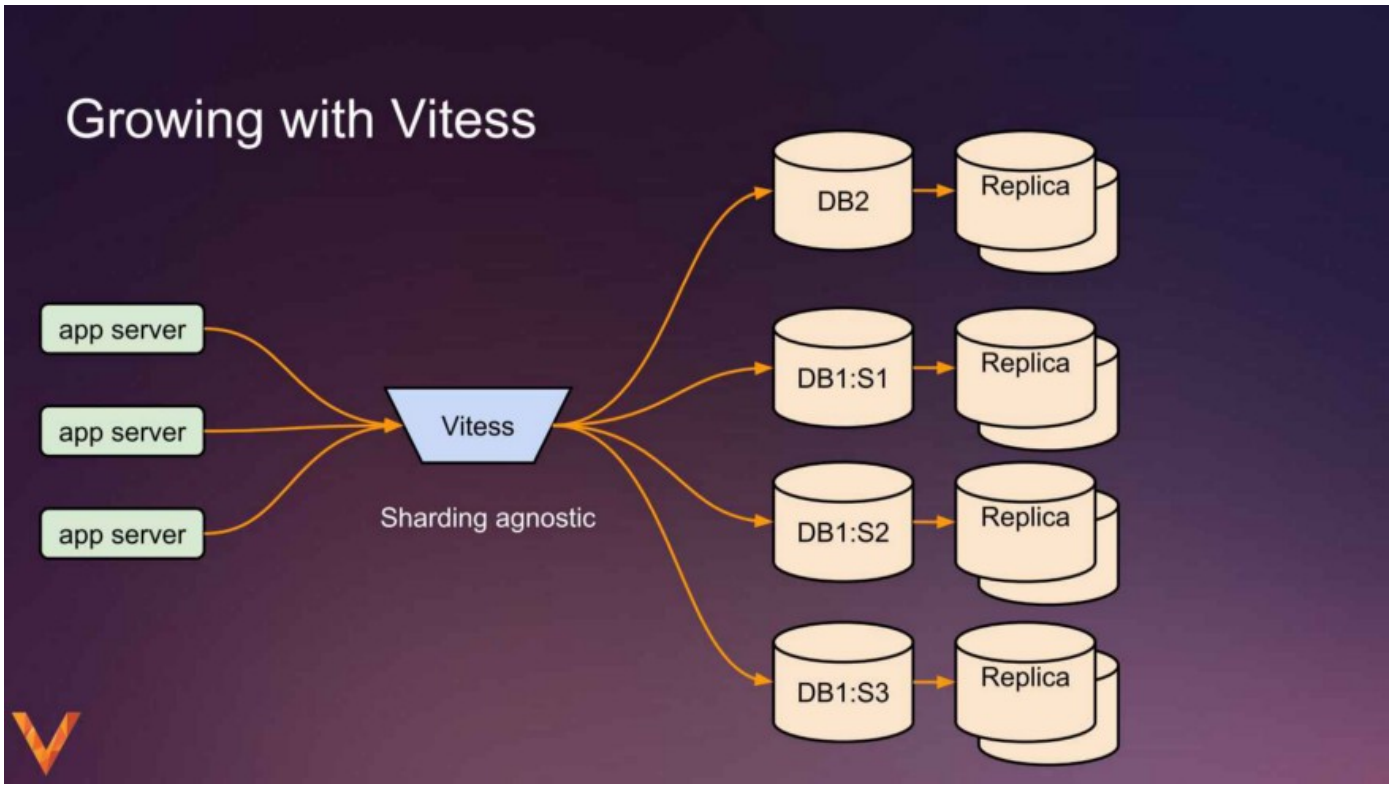
Here are some important features of Vitess:

Scalability: Its built-in sharding features let you grow your database without adding sharding logic to your application.

Performance: It automatically rewrites bad queries to improve database performance. It also uses caching mechanisms and prevents duplicate queries.

Manageability: It improves manageability by automatically handling failovers and backups functionalities.

Sharding management: MySQL doesn't natively support sharding, but we will need it as your database grows. It helps us to enable live resharding with minimal read-only downtime.



Cache

We can introduce a cache for metadata servers to cache hot database rows. Using Memcache to cache the data and Application servers before hitting the database can quickly check if the cache has the desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

Content Delivery Network (CDN)

A CDN is a system of distributed servers that deliver web content to a user based on the user's geographic location, the origin of the web page, and a content delivery server.

Our service can move popular videos to CDNs:

- CDNs replicate content in multiple places. There's a better chance of videos being closer to the user and, with fewer hops, videos will stream from a friendlier network.
- CDN machines make heavy use of caching and can mostly serve videos out of memory.

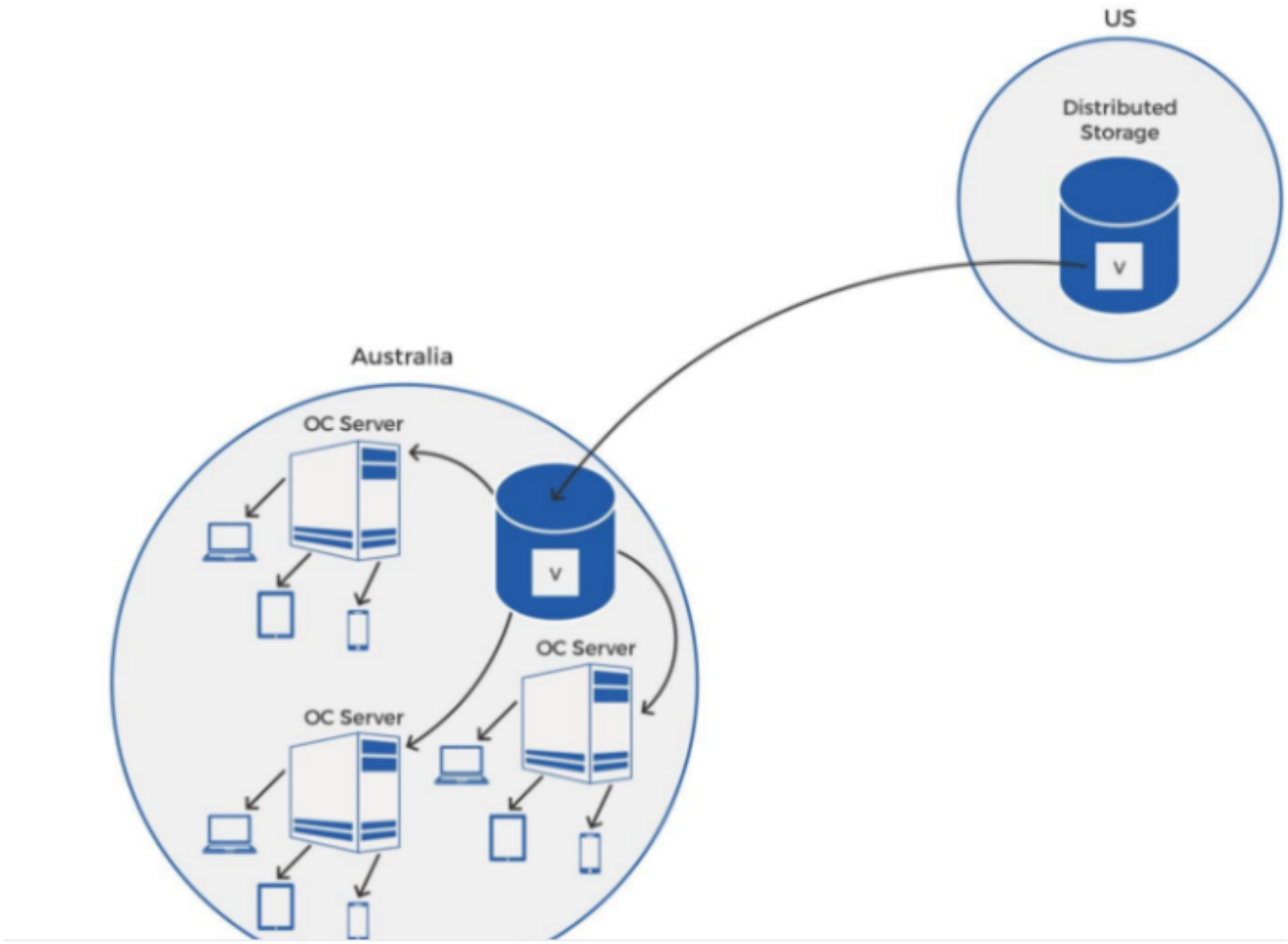
Less popular videos (1–20 views per day) that are not cached by CDNs can be served by our servers in various data centers.

Open Connect is Netflix's customized CDN (Content Delivery Network). CDN is a network of distributed servers and their data centers to cache web content and deliver it quickly to the users by decreasing the physical distance between the user and the content.

Netflix's Open Connect has partnered with many ISPs across the globe to cache popular Netflix videos so that they may be delivered to the nearby viewers locally, eliminating the need for connection with Netflix's US-based servers. The Netflix videos are available in the data centers of local ISPs through Open Connect.

How Does A Newly Uploaded Video Reach The Viewer using CDN:

So once the video is processed (split and encoded) and ready for viewing, it is stored in Amazon S3. It is also pushed to all the Open Connect servers in the world. If a new video is uploaded on Netflix and is to be made available to Australian audiences, it will need to reach the continent via undersea cables.



Instead of directing all the Netflix traffic through this expensive route, Netflix copies the video file (denoted by V in the diagram above) from US-based storage to a storage location in Australia once during off-peak hours. Once the video has reached the continent, it's copied to all the Open Connect servers present in the ISP networks.

When the viewer presses the play button, the video is streamed from the nearest Open Connect Appliance (OCA) installed at the local ISP and displayed on the viewer's device.

References:

1. https://techtakshila.com/system-design-interview/chapter-2/
2. https://www.enjoyalgorithms.com/blog/design-youtube-system-design-interview-question
3. https://www.educative.io/module/lesson/grokking-system-design-interview/N0jgjk3l9qz#7.-Detailed-Component-Design

4. https://www.linkedin.com/pulse/netflix-youtube-design-saral-saxena/

5. https://www.scaleyourapp.com/youtube-database-how-does-it-store-so-many-videos-without-running-out-of-storage-space/#:~:text=MySQL%20is%20the%20primary%20database%20used.