Step 1: Outline use cases and constraints

Use cases

- **Video streaming/CDN**
    - OpenConnect
    - After user hits the play button on video all further communication is with CDN network
- **Backend/Service**
  Handles everything except video streaming.
    - Video onboarding (uploading)
        - Preprocessing
            - Validation
            - Converting to different formats (mp3, 3gp…) and resolutions.
                - 1200 different device types
                - Network bandwidth optimization
    - Highly available
- **Client**
    - Netflix application
        - User home page
        - Search
        - Watch video
    - Video onboarding client
        - Upload high quality videos

Out of scope

- Data analytics

Constraints and assumptions

State assumptions

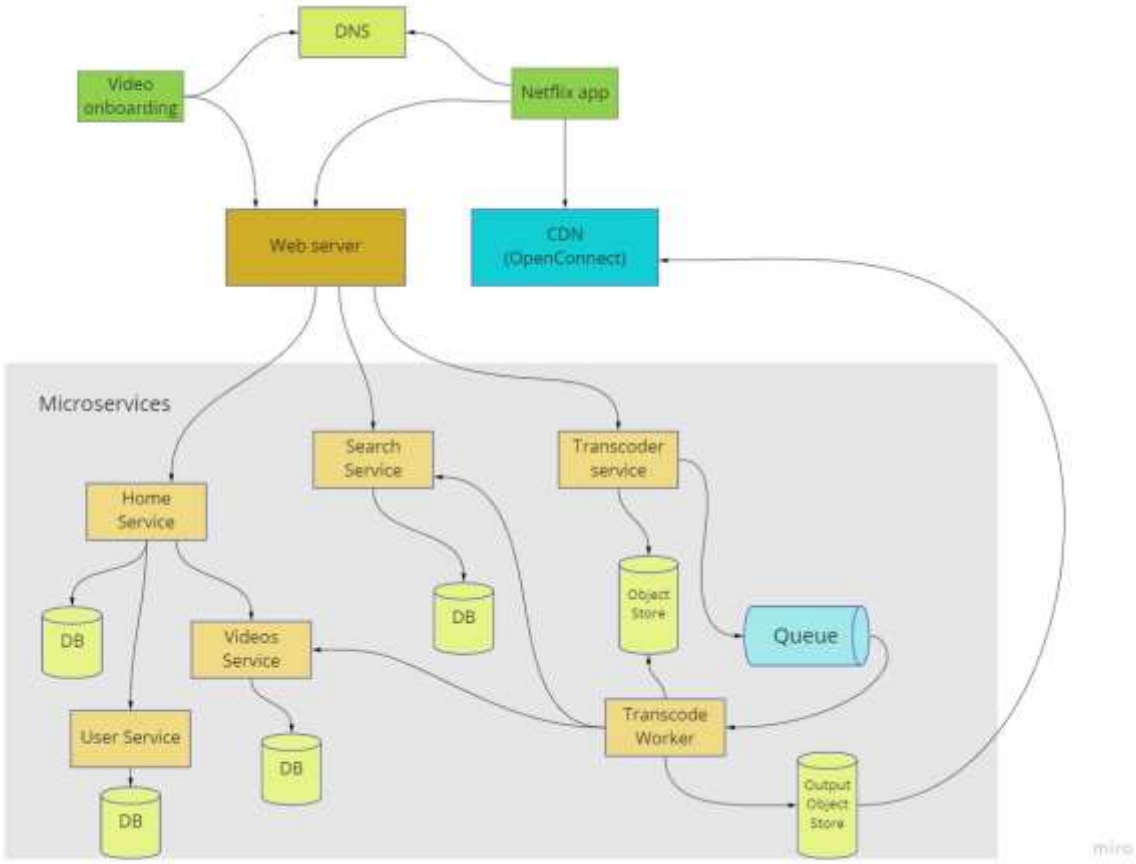
- 180M subscribers
- 200+ countries
- 2200 different client devices
- Traffic is not evenly distributed

Calculate usage.

- Standard plan (1 user at a time per subscription)
- 200M users/day
    - 2.5K users/sec
- Processed video size, avg. 500MB
- 400M watched movies/day
    - **5K movies/s**
    - **~ 5K searches/s**
    - 200PB/day
    - **2.3TB/s**
- **500 movies uploaded/day**
    - 250GB/day
    - **5.8MB/s**
    - **460TB in 5 years**
- *It is out of scope, data analytics part, it is good to know.*
    - *500B data events to persist (write),*
    - *1.3 PB/day , **15GB/s***
    - *8M events during peak time (24G/sec)*
        - *Error logs*
        - *UI activities*
        - *Video viewing activities*
        - *Performance events*
        - *Diagnostic*
        - *…..*
    - *9:1 write to read ratio*
        - ***1.6GB/s read***

Just to better understand how much has to be wrote.

Step 2: Create a high level design



Step 3: Design core components

Use case: Video onboarding.

- **Client** (**Video onboarding**) uploads a high-quality video to the **Web Server** which acts as API gateway and reverse proxy.
- **Web Server** routes request to the **Transcoder Service**
- **Transcoder Service**
    - Validates video and saves it in **Object Store**.
    - Informs all interested services that video is onboarded, **Videos Service,** which needs to know about a new content (status = soon).
    - Publish a message/task to the queue about video transcode.
    - **Transcode Worker**
        - Consume a message/task.
        - Download a video from **Object Store,** creates multiple replicas in different formats and resolutions.
        - Uploads created replicas to the **Output Object Store**
        - Informs **Videos Service** that a video is available for watching (status = active).
        - Informs **Search Service** that video is available.

REST

Request

```
POST /api/v1/transcode
{
      title,
      content
}
```

Response

```
{
      Status: 202
}
```

Status 202: means that request has been accepted and is going to be processed later (asynchronously)

Use case: Homepage.

- **Client (Netflix app)** user logged in.
- **Client** sends request to the **Web Server** which routes it to the **Home Service.**
- User view history (titles, gender, actors, release year, etc.), location, rates, device…. are stored in **User Service** DB.
    - Result of data analytics part of the system
- **Videos Service** serves user videos list, based on different criteria from user view history.
- **Home Service** assembles different categories of videos personalized to the logged user.

Users

id	name	subscriptionid	planid	countrid
4	50	4	4	4

Views

id	userid	videoid	watched_untill	created
4	4	4	10	10

Videos

id	categoryid	title	thumbnail_path	director	rank	actor	status	created
4	4	50	100	50	2	50	1	10

REST

Request

GET /api/v1/home?userid=123

Response

```
{
      userid: 123,
      videos:
      [
            {
                  thumbnail,
                   category,
                   rank,
                   watched_until
            },
            {
                   thumbnail,
                   category,
                   rank,
                   watched_until
            }
      ]
}
```

Use case: Search a video

- **Client** requests a video by its title.
- **Web Server** forwards request the full text **Search Service.**
- **Search Service** return results ranked by different criteria (user preference, user location or overall popularity, similar content, release date…)

REST

Request

```
GET /api/v1/search?title=once_upon_a_time
```

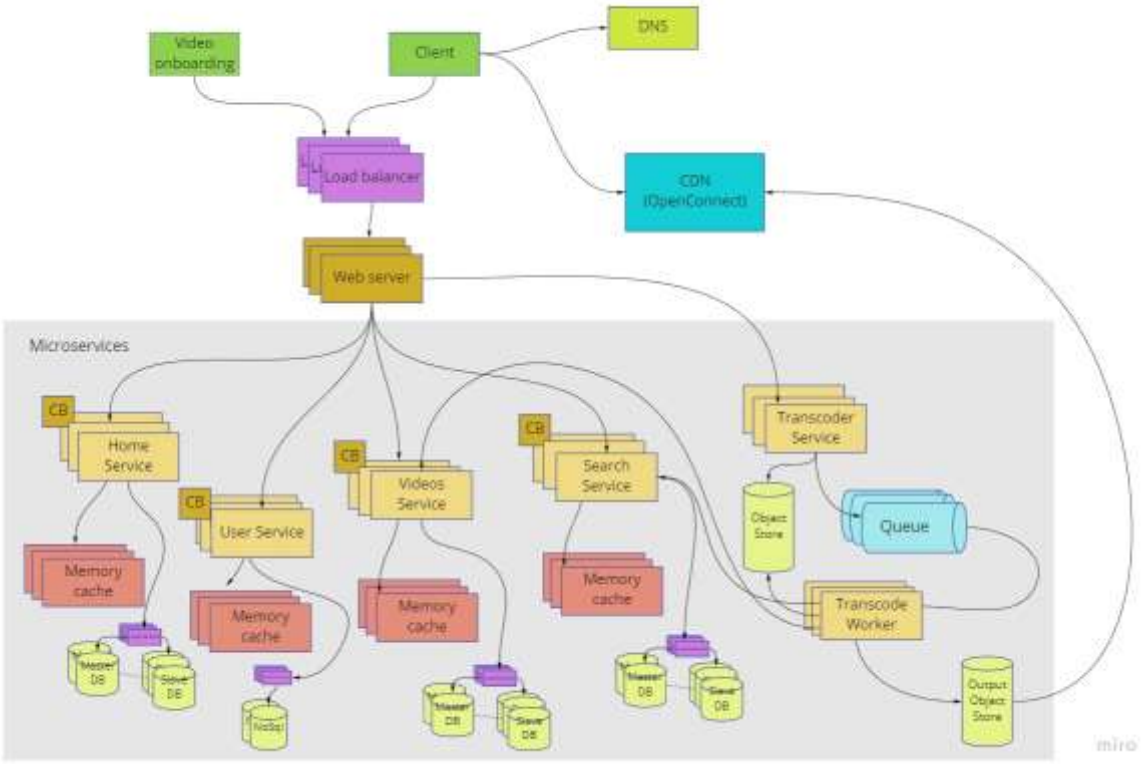
Response

```
{
      title: once upon a time…,
      videos:
      [
            {
                  thumbnail,
                  rank
            },
            {
                  thumbnail,
                  rank
            }
      ]
}
```

Use case: Play a video

- User hits a play button on selected video.
- **Client** sends a request to the CDN and depends on the client location, device type and network bandwidth appropriate video format from the closest server is going to be streamed.
    - CDN, for the best user experience, adapts video format on the fly to any change in the streaming condition.

Step 4: Scale the design



After

- Benchmark/Load Test,
- Profile for bottlenecks,
- Address bottlenecks while evaluating alternatives and trade-offs,
- Repeat.

Web Server is a bottleneck, particularly during peak days and hours, single point of failure and has to be **horizontally scaled**. High availability dictates adequate failover and replication patterns implementation. Which server, database is the most appropriate for the requested job? **Load Balancer** is going to decide. To protect against failure multiple load balancers must be set up.

How microservices are going to be implemented and integrated can be discussed, but some other time. Right now, it is important to understand that they have to do one thing, be autonomous, reactive, isolated, resilient, stateless, independently deployable, replaceable, observable, etc. There are competent authors on this subject like Eric Evans, Vaughn Vernon, Sam Newman… **Microservices** are going to be horizontally scaled in terms of high availability request and better responsiveness.

Amount of data and level of I/O interactions implies that some **RDMBS** databases have to be replicated and some replaced with **NoSQL** solution, more appropriate for big amount of data and heavy write and read, e.g., user view history data. Some data, like video thumbnail can be moved to **Object Store.**

To reduce the number of hits to databases and improve response times, in-memory caching can be used for the most frequent requests. **In-Memory caches**, **Redis or Memcached**, key-value stores, can be used as an application cache solution.

Highly intensive and demanding video transcoding has to be done asynchronously and in **parallel**.