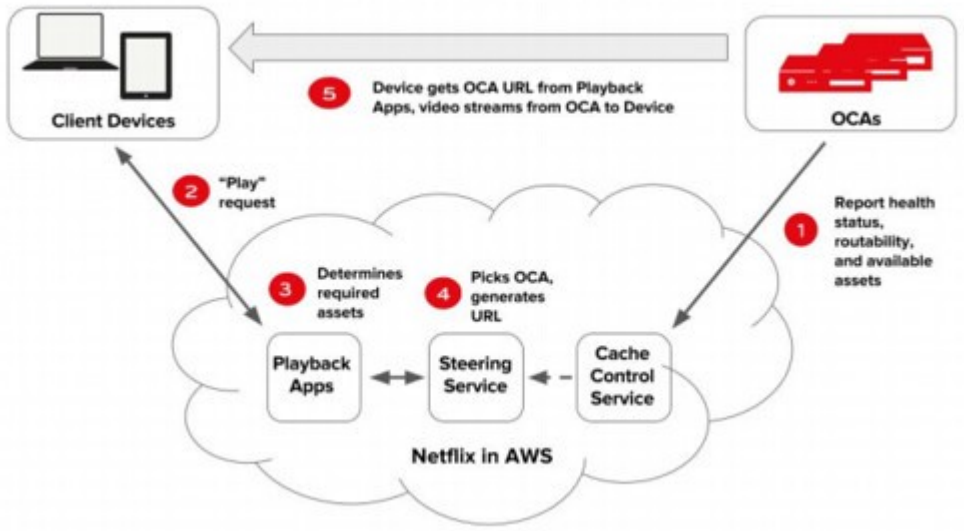
Netflix System Architecture

Let's discuss how to design Netflix.



I believe everyone likes to watch drama series and movies through certain applications. I like Netflix, but today I'm not recommending any movies, instead, I want to show the system logic behind the scenes which is quite amazing.

Functional Requirement

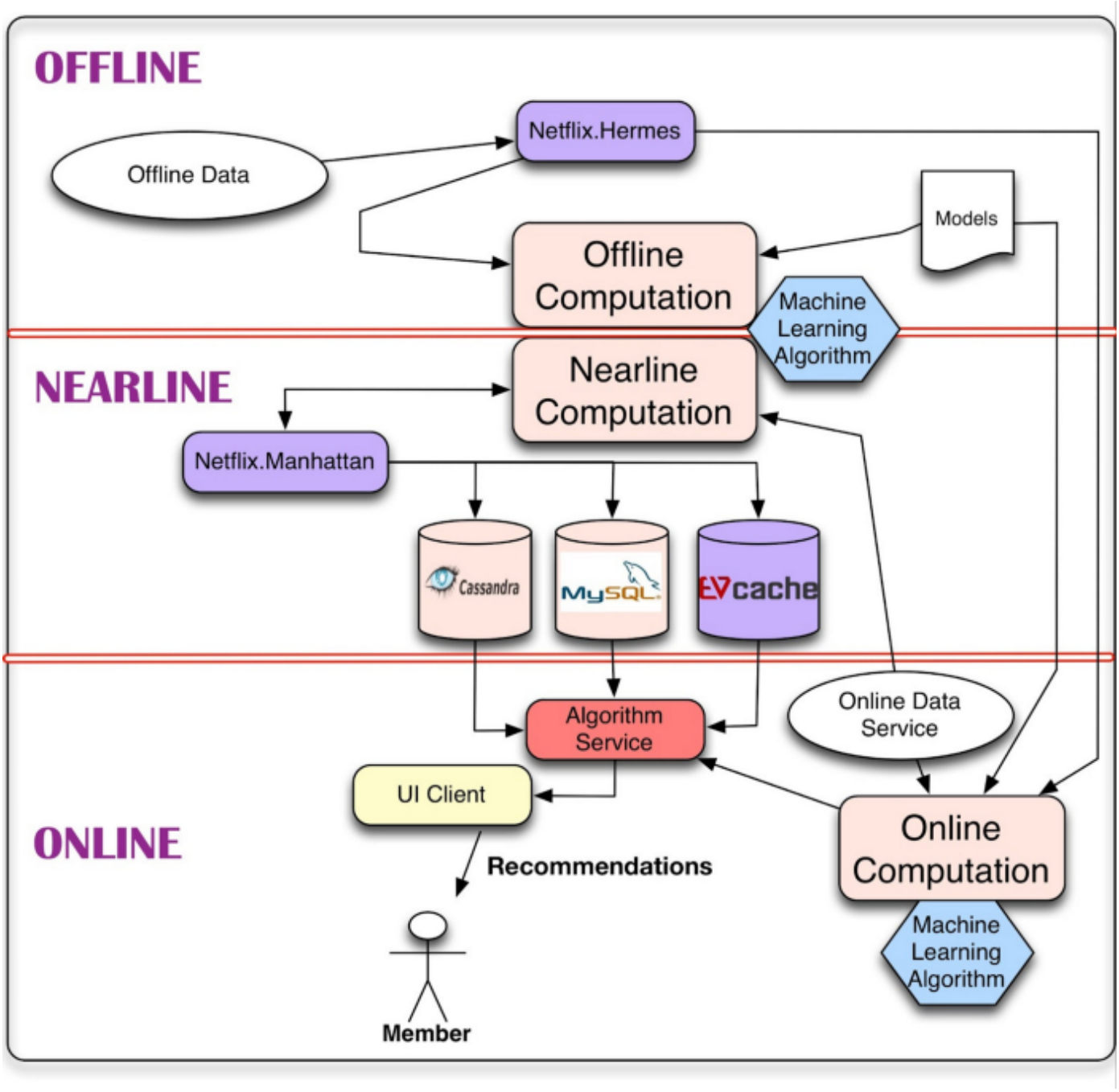
1. Create accounts, log in and delete the account
2. Subscribe or unsubscribe to different plans
3. Allow users to have and handle multiple accounts
4. Allow users to watch videos
5. Allow users to download and go watching offline mode
6. Allow users to search and discover the video through the video title
7. Netflix developers can upload videos from the backend and show them on the platform

8. The platform will show the trend, most popular videos, and by category which easy for users to choose
9. language selection for subtitles so that users can watch any videos even don’t speak the languages
10. Grouping of videos (TV series, drama series, movie series and treat each video as an independent)
11. Analytics also provide the user's suggestions or recommendations for the similar type of videos to users based on the user's behavior
12. Synchronize for the different device under the same account, which means that users can use a different device to continue watching the same episode without replay
13. Playback 24/7
14. Fallback

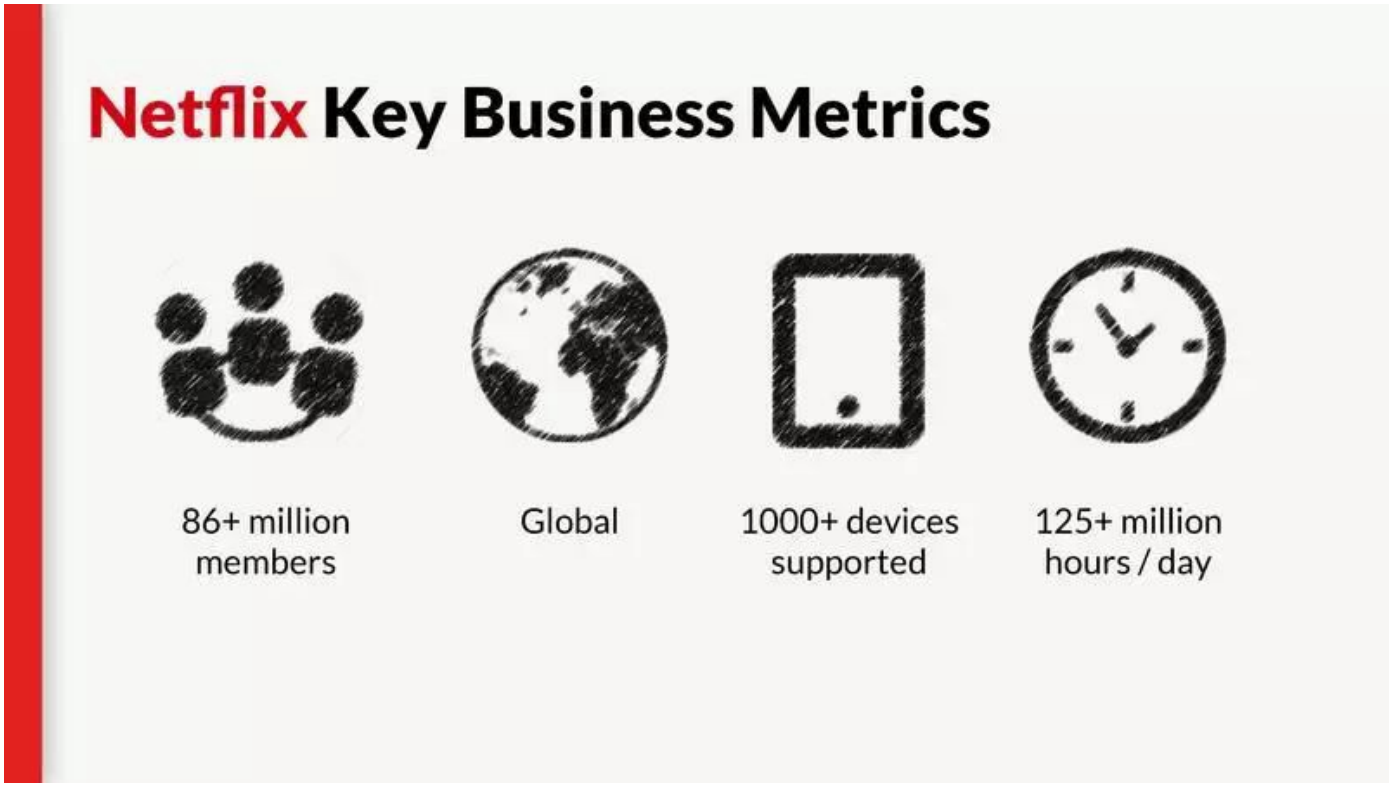
Non-functional Requirements

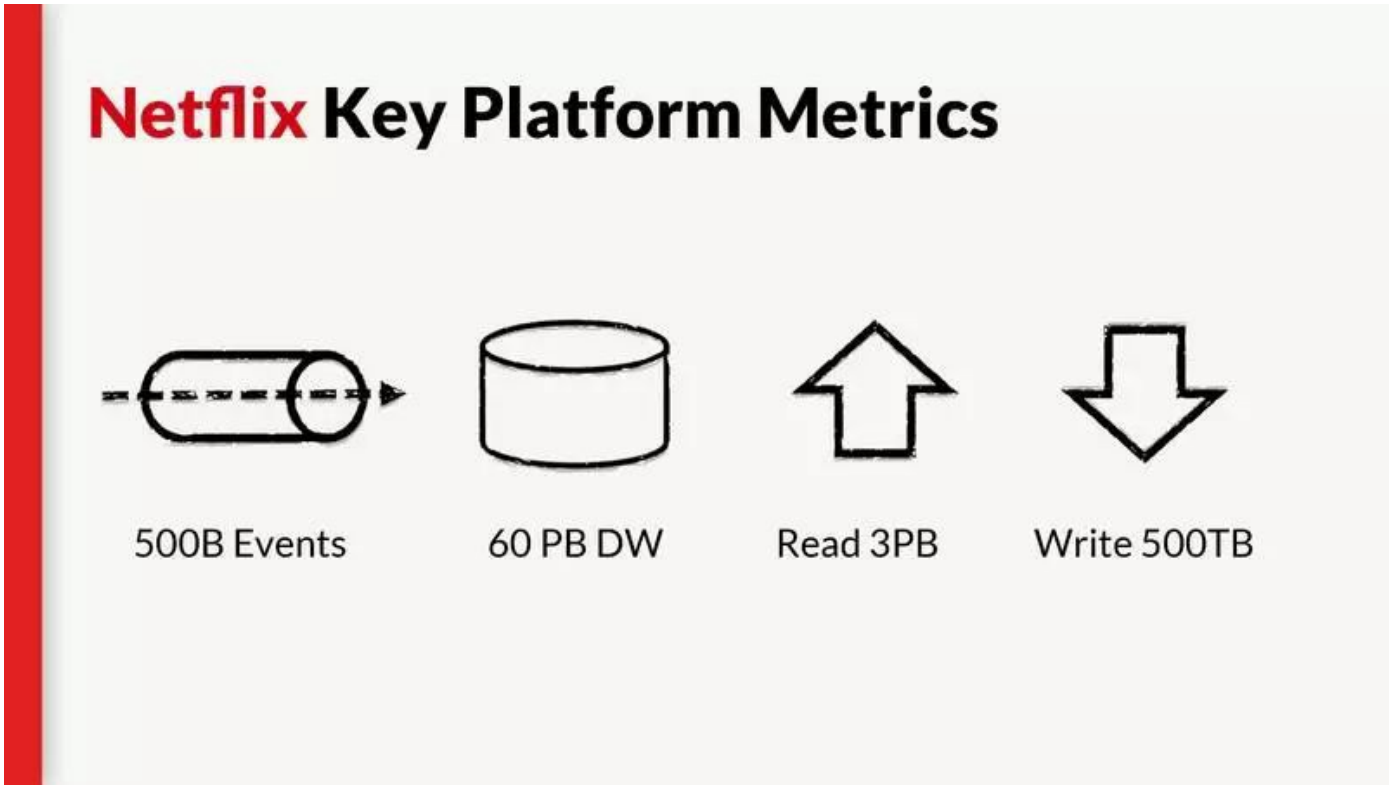
1. Users can stream videos in realtime without any lag or latency issue
2. The system will be highly reliable
3. High availability
4. Scalability
5. Video data is durable and easy to be accessible

Capacity Estimation



Let us do some mathematics to estimate the bandwidth and storage required

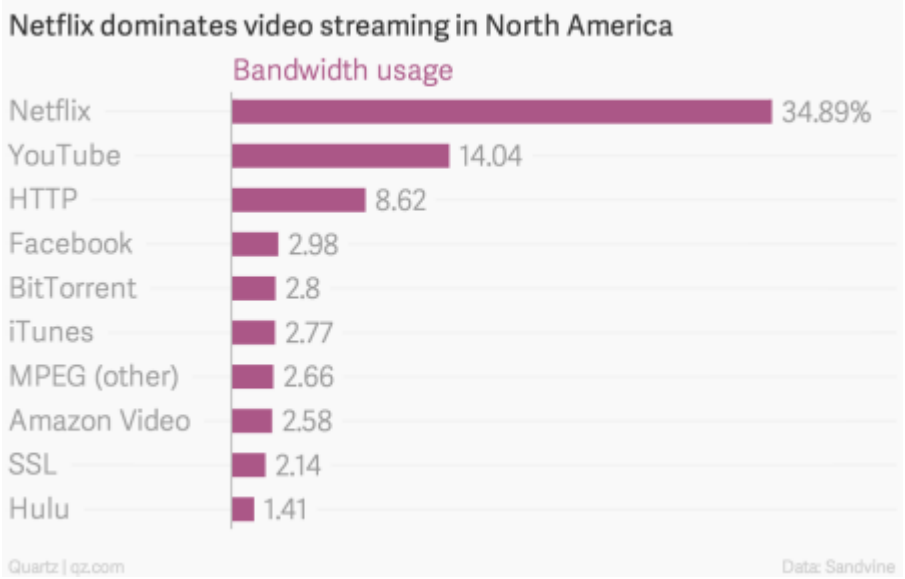


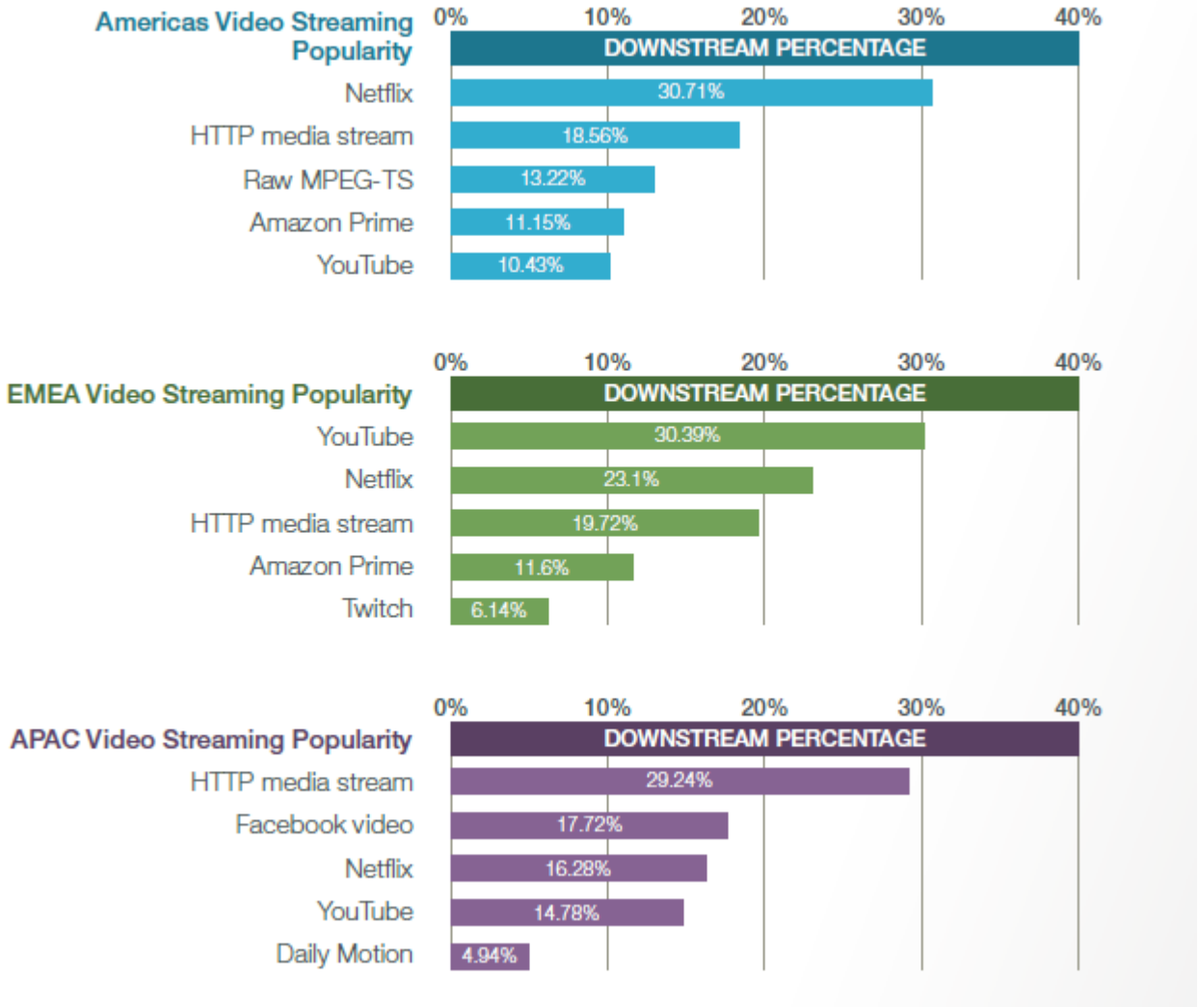


Assumptions

1. Total number of daily active users = 100 million
2. The peak daily active users, 100 million * 3 = 300 million

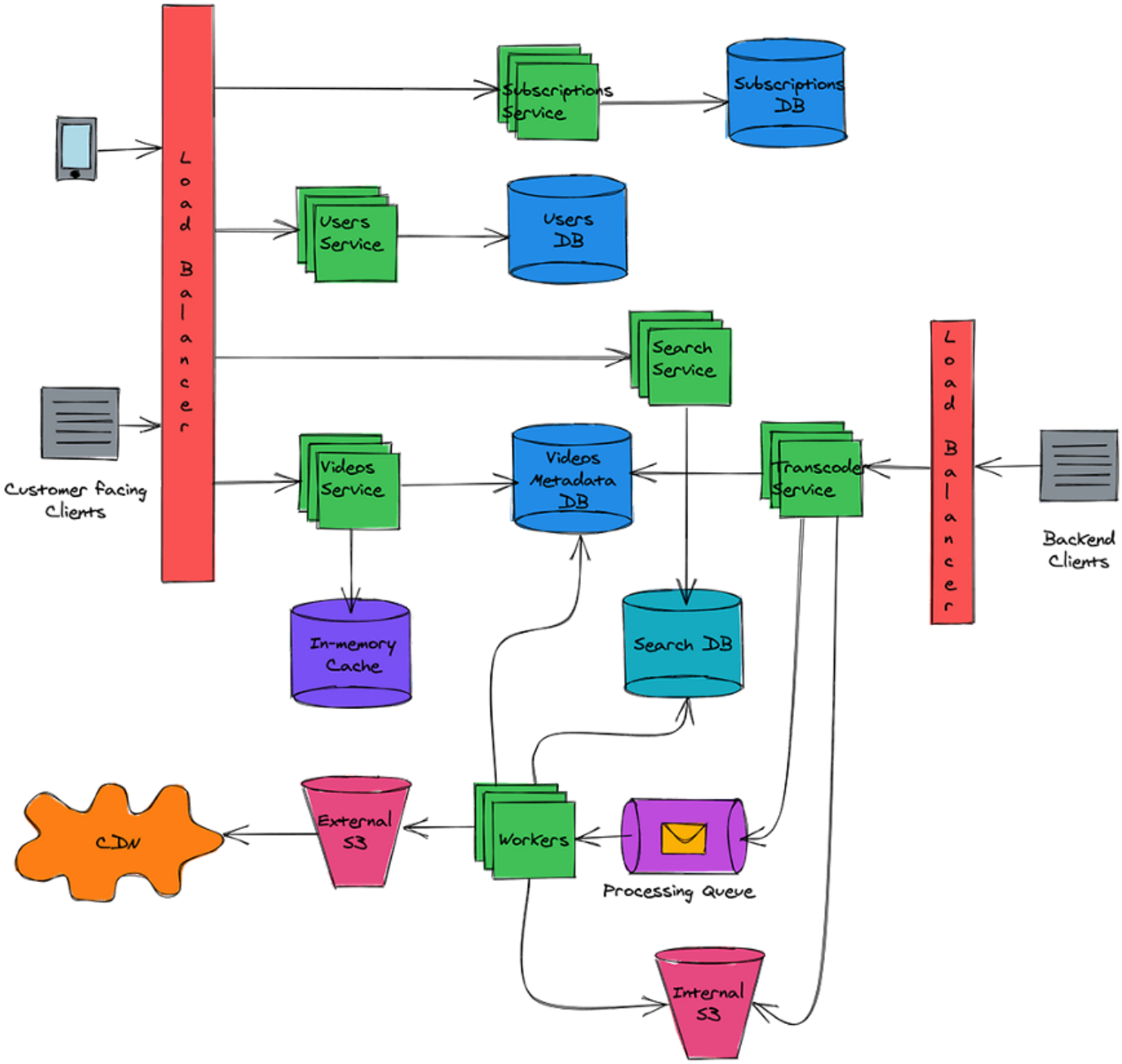
3. The max peak daily active users in 3 months, 300 million * 2 = 600 million
4. The average number of videos watched by each user per day = 5
5. The average size of one video = 500 MB
6. The average number of videos uploaded per day from the backend = 1,000
7. Total number of videos watched per day = 100 million *5 =500 million
8. Total peak video per day = 1.5 billion
9. Total max peak video per day = 3 billion
10. Total egress per day = 500 million * 500 MB = 250 PB (Peta Byte)
11. Egress bandwidth = 29.1GB/sec
12. Total ingress for upload = 1,000 * 500MB = 500GB
13. Ingress bandwidth =5.8MB/sec
14. Total Storage required in 5 years = 500 GB*5*365 = 912.5TB (please note that Netflix creates multiple formats and resolutions for each video optimized for different device types. So the storage will be more than 912.5TB.

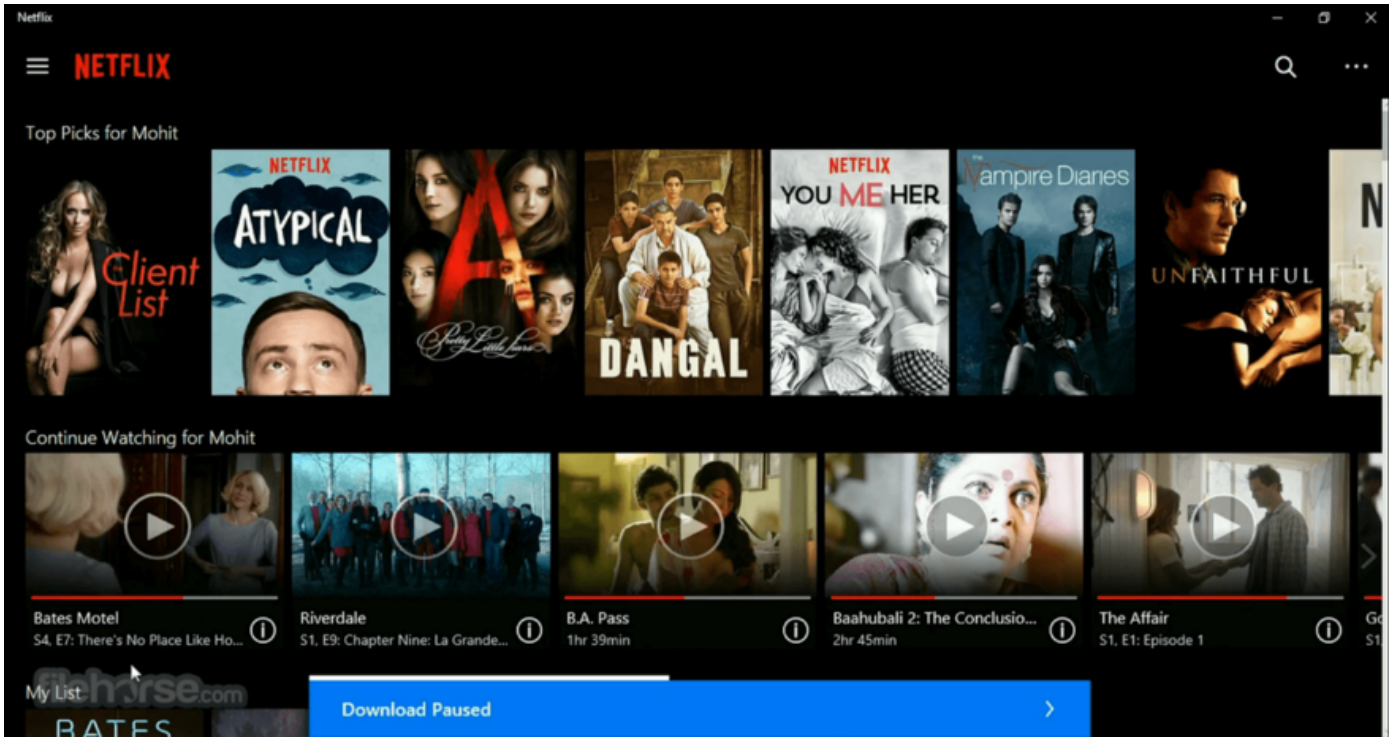




The system components

The Detailed Component Design

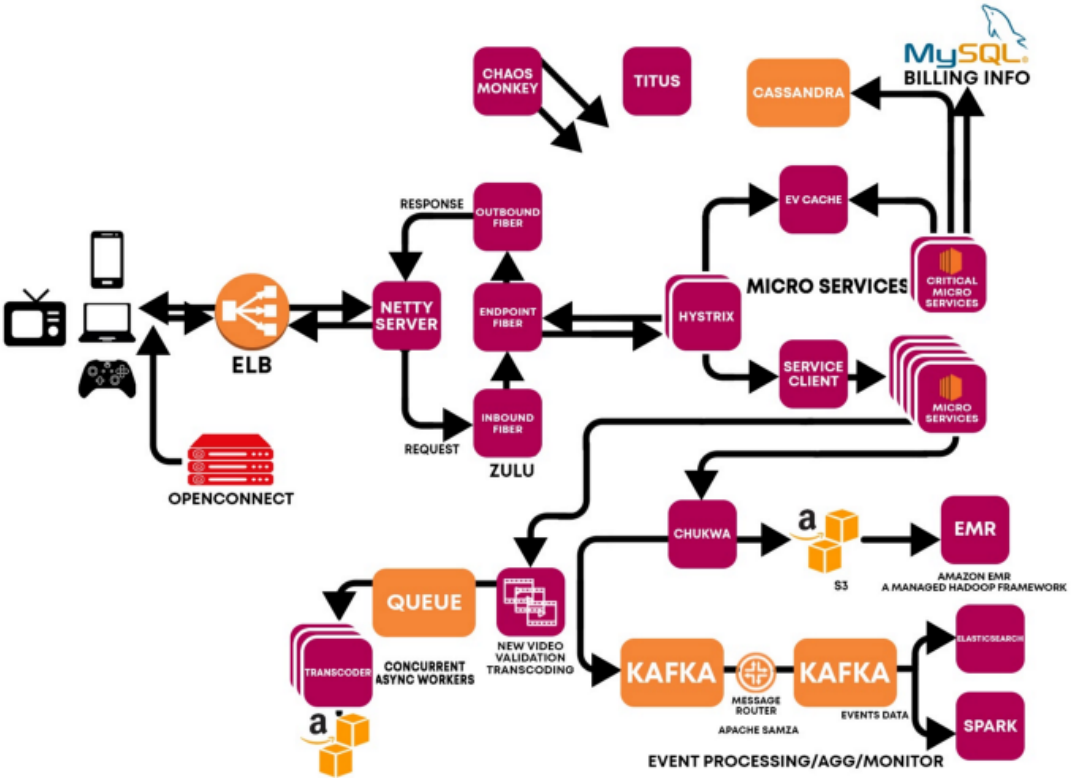

Load Balancer
Subscriptions Service
Subscriptions DB
Users Service
Users DB
Search Service
Customer facing Clients
Videos Service
Videos Metadata DB
Transcoder Service
Load Balancer
Backend Clients
In-memory Cache
Search DB
CDN
External S3
Workers
Processing Queue
Internal S3



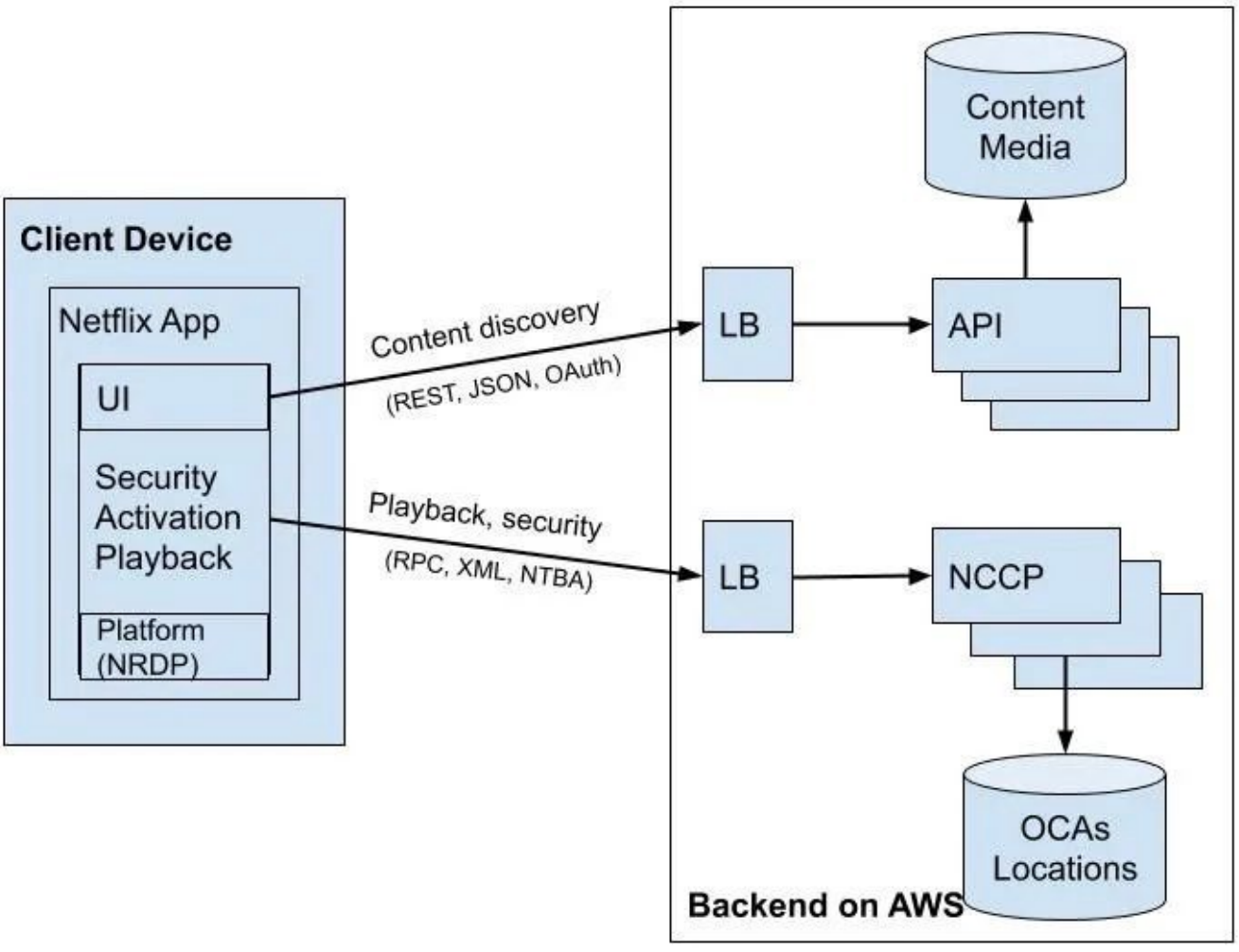
1. **Client App**

- Mobile (Apple ios, androis os, huawei OS and etc)
- Tablet (Ipad, Android tablet, windows)
- TV
- Laptop or computer

React.js is used to write the front-end because of its loading/startup speed, durability/modularity, and runtime performance.

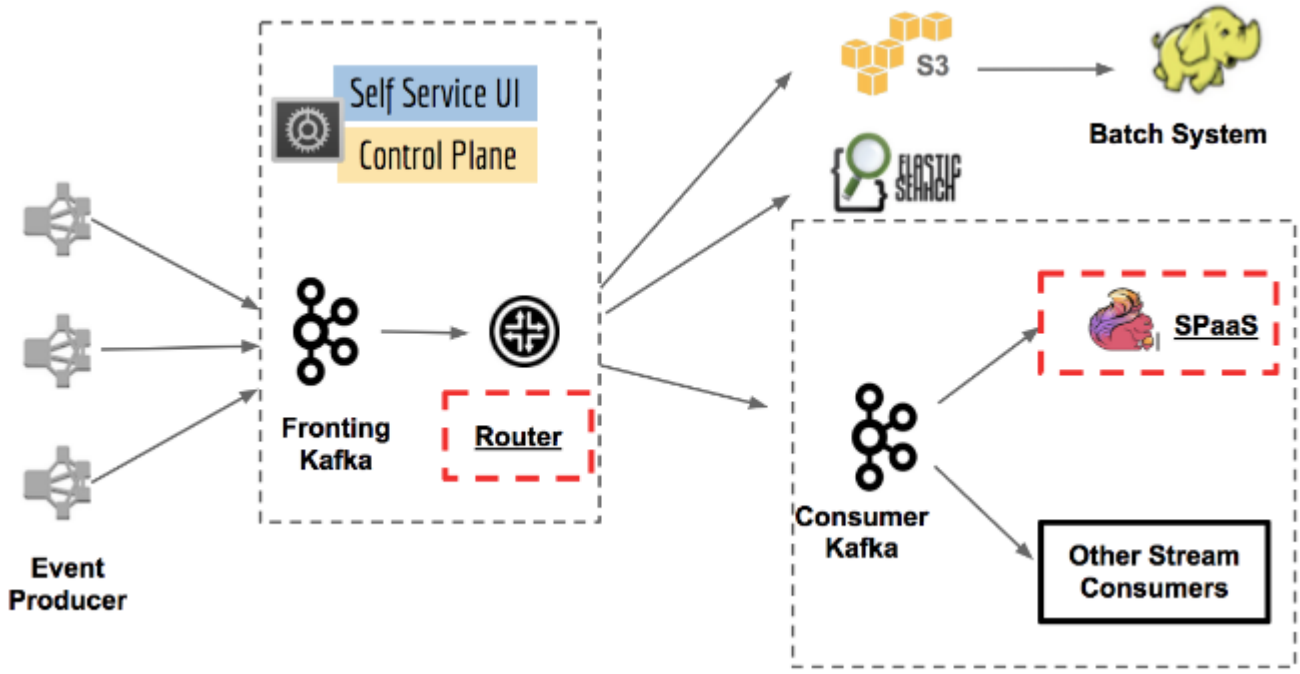

CHAOS MONKEY
TITUS
CASSANDRA
MySQL
BILLING INFO
RESPONSE
OUTBOUND FIBER
EV CACHE
MICRO SERVICES
CRITICAL MICRO SERVICES
NETTY SERVER
ENDPOINT FIBER
HYSTRIX
ELB
SERVICE CLIENT
MICRO SERVICES
INBOUND FIBER
REQUEST
ZULU
OPENCONNECT
CHUKWA
S3
EMR
AMAZON EMR
A MANAGED HADOOP FRAMEWORK
QUEUE
NEW VIDEO VALIDATION TRANSCODING
TRANSCODER
CONCURRENT ASYNC WORKERS
KAFKA
MESSAGE ROUTER
APACHE SAMZA
KAFKA
EVENTS DATA
ELASTICSEARCH
SPARK
EVENT PROCESSING/AGG/MONITOR

2. Backend


Client Device
Netflix App
UI
Security
Activation
Playback
Platform
(NRDP)
Content discovery
(REST, JSON, OAuth)
Playback, security
(RPC, XML, NTBA)
LB
LB
API
NCCP
Content
Media
OCAs
Locations
Backend on AWS

Netflix started to implement the microservices architectures in 2011 completely for their cloud-based system to manage running those workloads. It has small manageable software components on the API level, which enables and serves requests from apps and websites. Microservices rely on each other internally for requests and fetching data. There are Java, MySQL, Gluster, Apache Tomcat, Chukwa, Cassandra, KAFKA, and Hadoop to strengthen the backend system. Other than streaming videos, the backend handles everything such as processing videos, onboarding new content, network traffic management, and distribution of resources across servers worldwide. Currently, Netflix adopts AWS (Amazon Web Services).

Data processing involves all the events required after a click to the video, it takes nanoseconds to process the video and stream it to the user. There are around 600 billion events daily, resulting in 1.5PB data, and during peak hours (evening and night), there are around 8 million events per second. Events are UI activities, video viewing activities, logging errors, troubleshooting, diagnostic events, processing events, and performance events. but all these events are done using **Kafka and Apache Chukwe.**



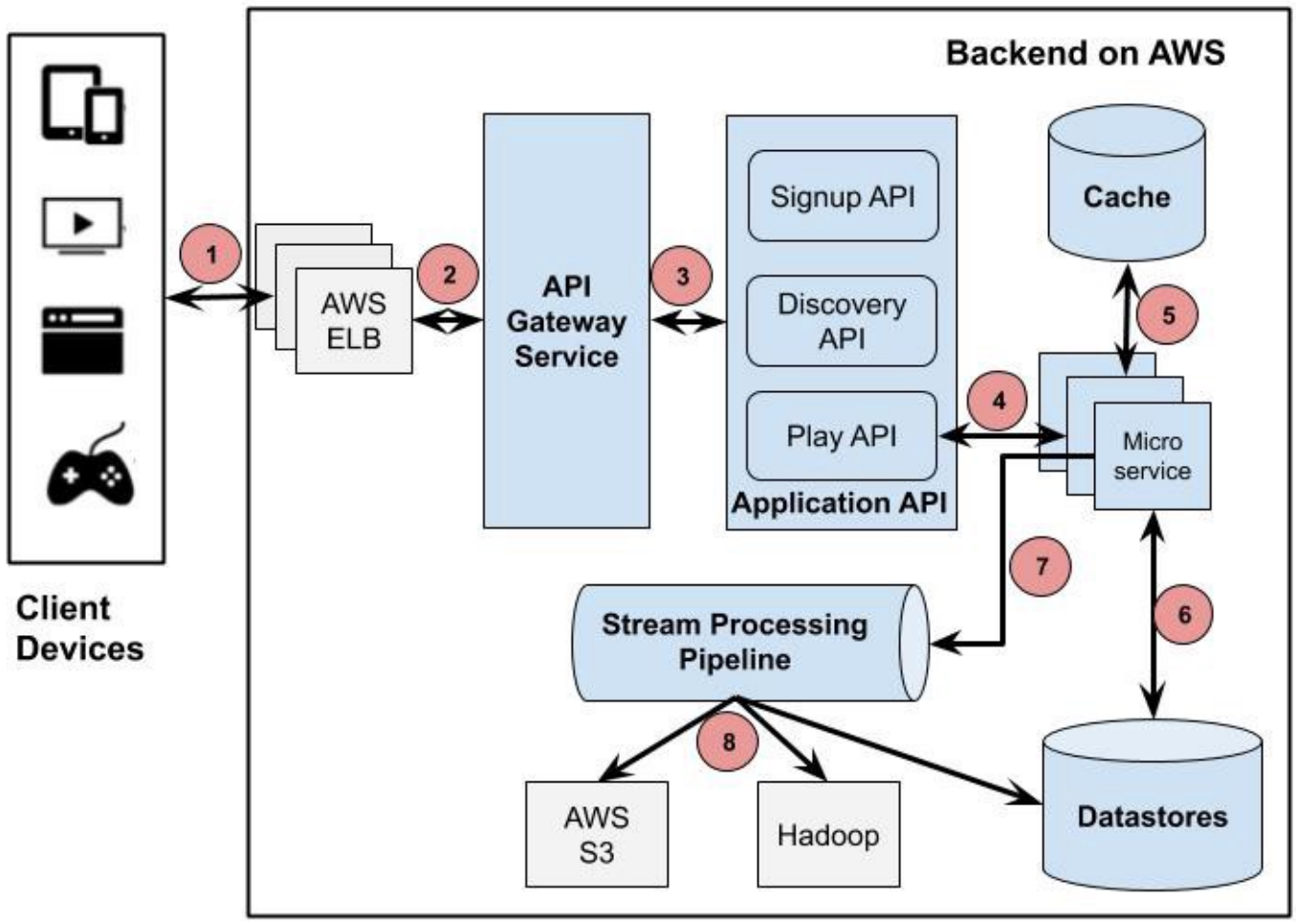
Kafka and Apache Chukwe

- to ingest the data which is produced in a different part of the system.

- Apache Chukwe is an open-source data collection system for collecting logs or events from a distributed system. It is built on top of HDFS and Map-reduce framework. It comes with Hadoop’s scalability and robustness features. Also, it includes a lot of powerful and flexible toolkits to display, monitor, and analyze the result. Chukwe collects the events from different parts of the system and from Chukwe you can do monitoring, analysis or you can use the dashboard to view the events. Chukwe writes the event in the Hadoop file sequence format (S3). After that Big Data team processes these S3 Hadoop files and writes Hive in Parquet data format. This process is called batch processing which basically scans the whole data at the hourly or daily frequency. To upload online events to EMR/S3, Chukwa also provide traffic to the Kafka (main gate in real-time data processing). Kafka is responsible for moving data from fronting Kafka to various sinks: S3, Elasticsearch, and secondary Kafka. Routing of these messages is done using the **Apache Samja** framework. Traffic sent by the Chukwe can be full or filtered streams so sometimes you may have to apply further filtering on the Kafka streams. That is the reason we consider the router to take from one Kafka topic to a different Kafka topic.

Elastic Search

- Netflix is running approximately **150** clusters of elastic search and **3, 500** hosts with instances.
- Netflix is using elastic search for data visualization, customer support, and for some error detection in the system. For example, if a customer is unable to play the video then the customer care executive will resolve this issue using elastic search. The playback team go to the elastic search and search for the user to know why the video is not playing on the user’s device. They get to know all the information and events happening for that particular user. They get to know what caused the error in the video stream. Elastic search is also used by the admin to keep track of some information. It is also used to keep track of resource usage and to detect signup or login problems.



Backend- Services

- **User and authentication service** (Mainly responsible for user authentication and profiles. The data will be in a relational database like MySQL or PostgreSQL. SO, RDBMS is a suitable choice.
- **Subscription Management service** (Manage the subscription of the users. Since data processing by this service is highly transactional in nature, RDBMS is a suitable choice.
- **Videos Service** (Surfacing videos to end-users. This service store videos metadata in an RDBMS like MySQL or PostgreSQL. For quick response time, this service would implement a write-around cache using an in-memory cache like Redis or Memcached.
- **TransCoder Service** (Checking the quality of uploaded videos, compressing the videos with different codecs, creating different resolutions of the video. Once a video is uploaded to the Transcoder service, it will upload the same to an internal distributed

storage like an S3 bucket and add an entry to the database. Kafka or RabbitMQ process a message in the queue. So that, the backend workers would consume messages from the queue, download the video from the internal S3 bucket and transcode it to different formats. Once transcoding is complete, the worker would upload the video to an external S3 bucket and update the status of the video in the database as active for viewing by end-users. The worker would also add an entry of the video metadata in the search database which supports full-text search. This would enable the end-users to search for videos using their title or summary. Then, the video from an external S3 bucket would also be cached over a CDN for reducing latency improving playback performance.

- **Global Search Service** (enable end-users to search for a video using metadata like title, summary, etc. Elasticsearch or Solr can be used to support full-text searching because they are stored in Elastic Search DB that enables users to search for movies, series by title, or any meta-data associated with the video. This service also ranks the results based on recency, reviews, recommendations, and popularity for a better user experience. Also, the elastic search can track down users' events in cases of failure. Then the customer care team uses elastic search to resolve issues.

3. Cloud

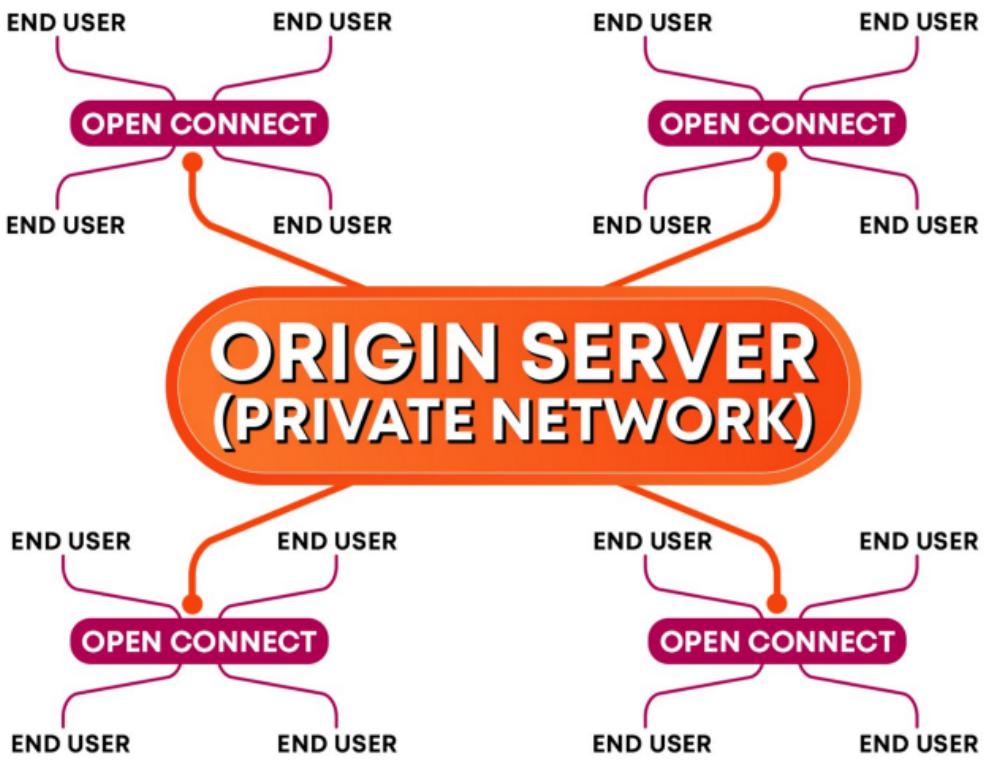
- Netflix migrated its IT infrastructure to a public cloud. The cloud service used is AWS and Open connect (Netflix's custom CDN). These 2 cloud services work parallelly for video processing and delivering content to end-users.

4. CDN (Content Delivery Network)

It is a network of servers distributed globally. When you play the video, the video displayed on your device is streamed from this component. This significantly reduces the response time as the video is streamed from the server nearest to your location.

- CDNs replicate content in multiple places. There's a chance of videos being closer to the user and with fewer hops.
- CDN machines make heavy use of caching, so even though the video is not found from the server, it also can be obtained from the cache

- Less popular videos (1–20 views per day) that are not cached by CDNs



5. Open connect

It is Netflix's in-house or custom global CDN responsible for the storage and delivery of movies and TV shows to Netflix users globally. We hit the play button, the video is streamed from open connect stored in different locations in the world. If the video is present there, the client app can access easily. If the video is not present there, Netflix has to process that video from S3 AWS, then open connect will stream that video to your client app.

Netflix Open Connect

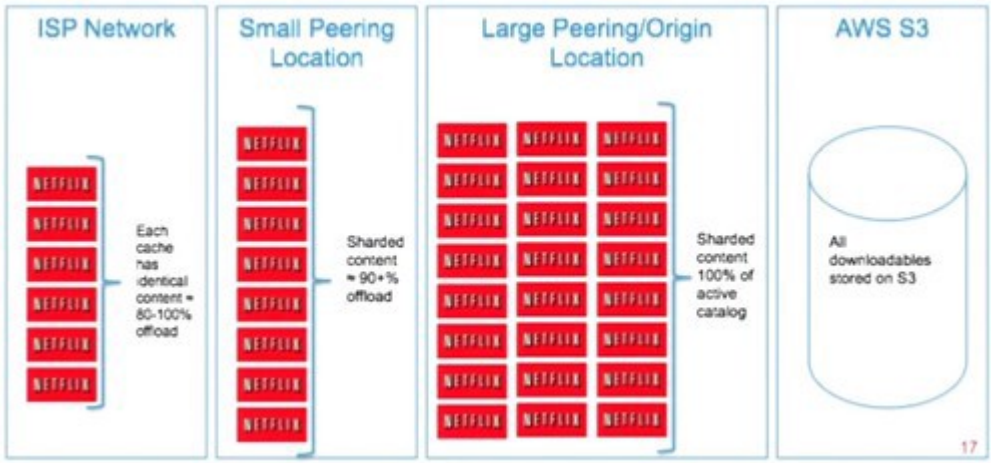
The following diagram shows the typical workflow for a new ISP partner engagement: To begin the engagement process, you…

openconnect.netflix.com

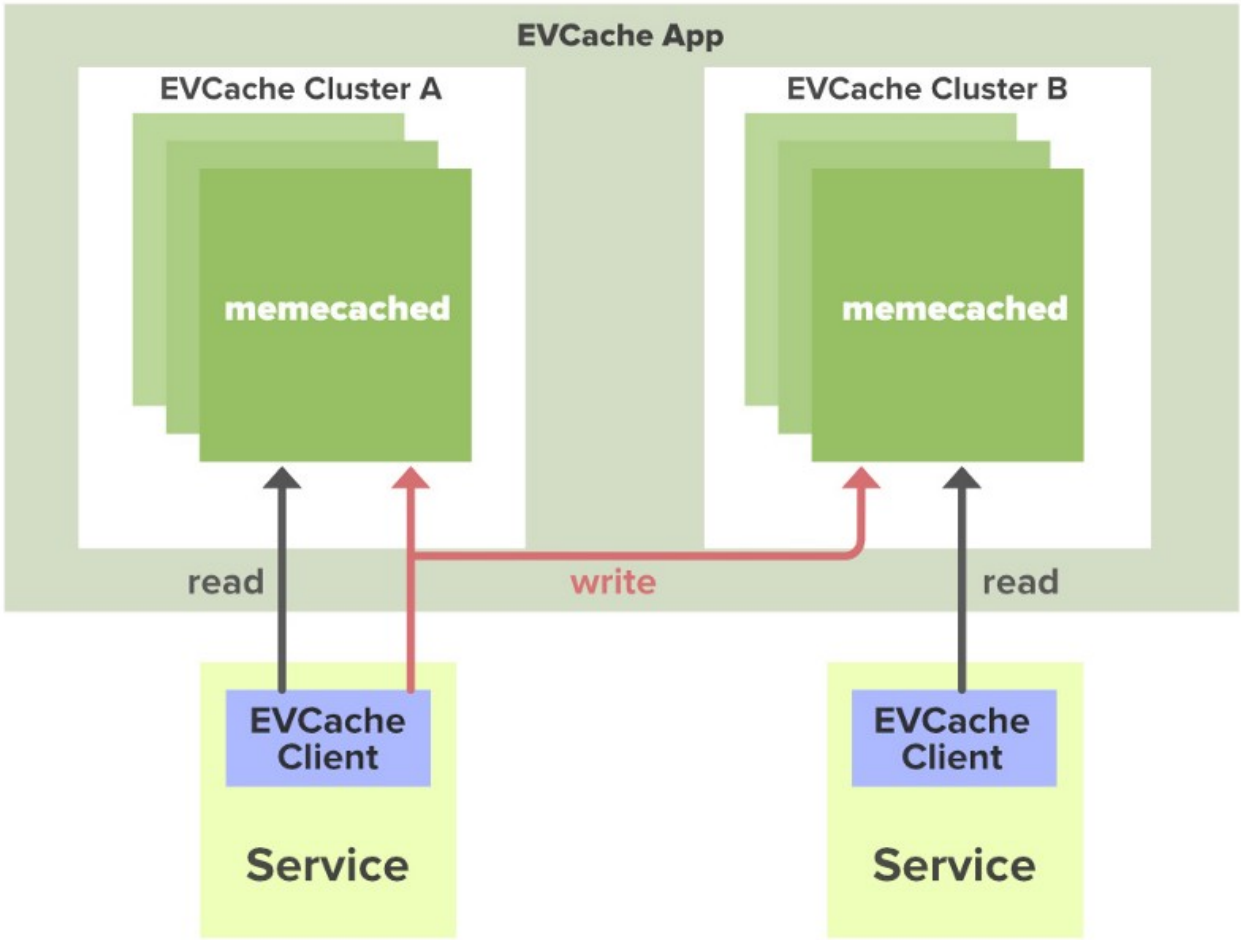
6. Caching

Redis and Memcached cache the data from the database in key-value pair because it can reduce the number of hits to the database. From the client-server, before access to the database, the system will check if data exists in the cache. If it exists then we bypass a

database trip. However, if data is not present in the cache, we will hit the database, get the data and populate the same in the cache too. So, the subsequent requests won’t hit the database. This caching strategy is known as write-around caching. The least recently used (LRU) eviction policy for caching data because will discard caching data that is oldest fetched.



- EV cache is actually a wrapper around Memcached



Netflix has deployed a lot of clusters in a number of AWS EC2 instances and these clusters have so many nodes of Memcached and they also have cache clients. The data is shared across the cluster within the same zone and multiple copies of cache are stored in sharded nodes. Every time when write happens to the client all the nodes in all the clusters are updated but when the read happens to the cache, it is only sent to the nearest cluster (not all the cluster and nodes) and its nodes. In case, a node is not available then read from a different available node. This approach increases performance, availability, and reliability.

7. Scalability

- **Horizontal Scaling** — add more application servers behind the load balancer to increase the capacity o the service.
- **Database replication** — Use the relational database in Master-slave configuration where write will happen to master and reads from the slave. This will improve the performance of reading queries as they won't be stopped due to write locks on rows.

There is a slight replication lag (a few milliseconds) as data is written to master DB then propagated to slave DB.

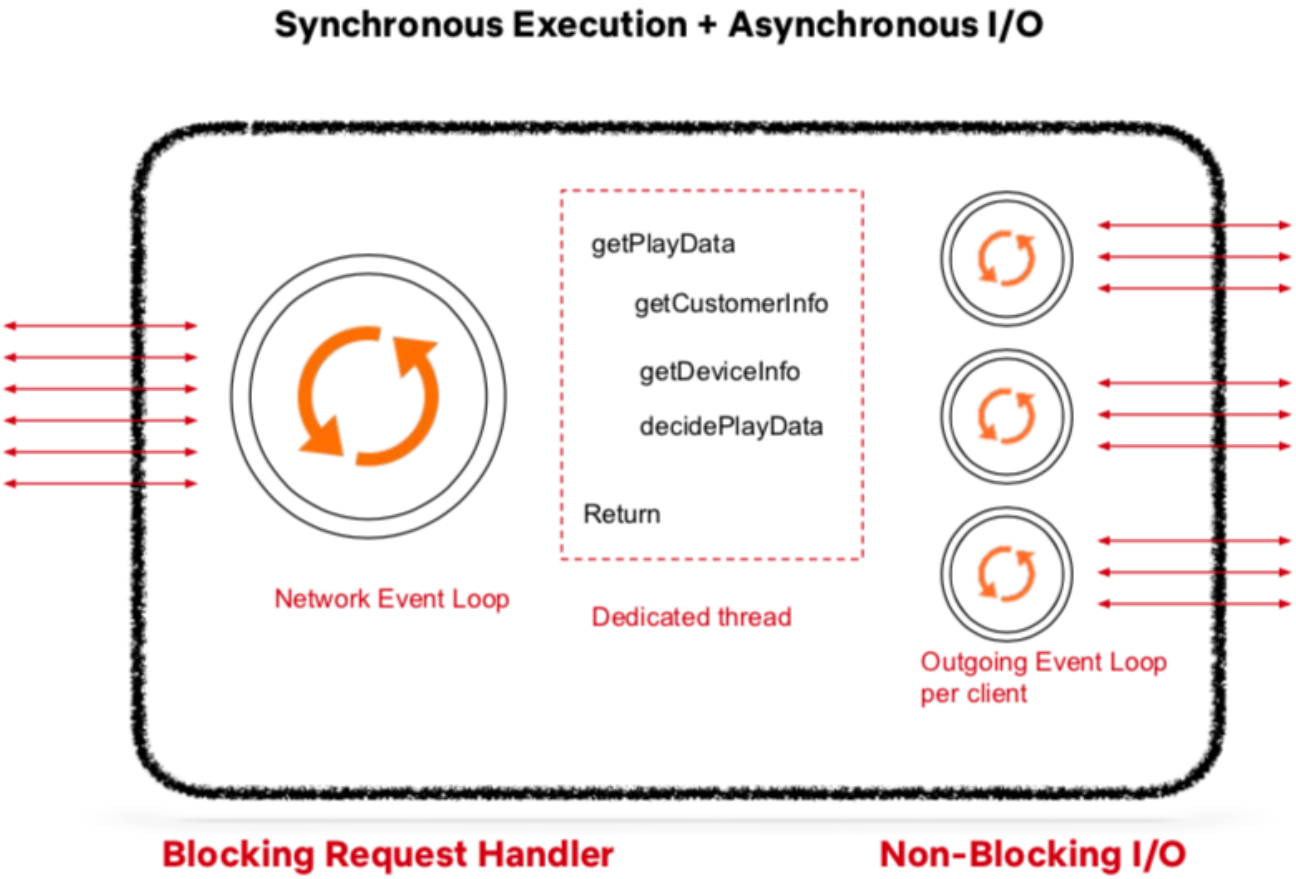
- **Database sharding** — distribute data to multiple servers to perform read/write operations efficiently. we can shard the video metadata database using video_id. our hash function will map each video_id to a random server where we can store the video metadata.
- **cache sharding** — We can distribute our cache to multiple servers. Redis has out-of-box support for partitioning the data across multiple Redis instances. Using consistent Hashing for distributing data will ensure that load is equally distributed if one instance goes away.
- **Search database sharding** — Elasticsearch comes with native support for sharding and replication. Sharding helps in improving the query runtime by running them in parallel against multiple shards.

8. Security

- **HTTPS** — encrypting the traffic between client and server over HTTPS. this will ensure that no one in the middle is able to see the data especially passwords
- **Authentication** — Each API request post-log-in, will do authentication by checking the validity of auth_token in the authorization HTTP header. This ensures that the requests are legitimate.

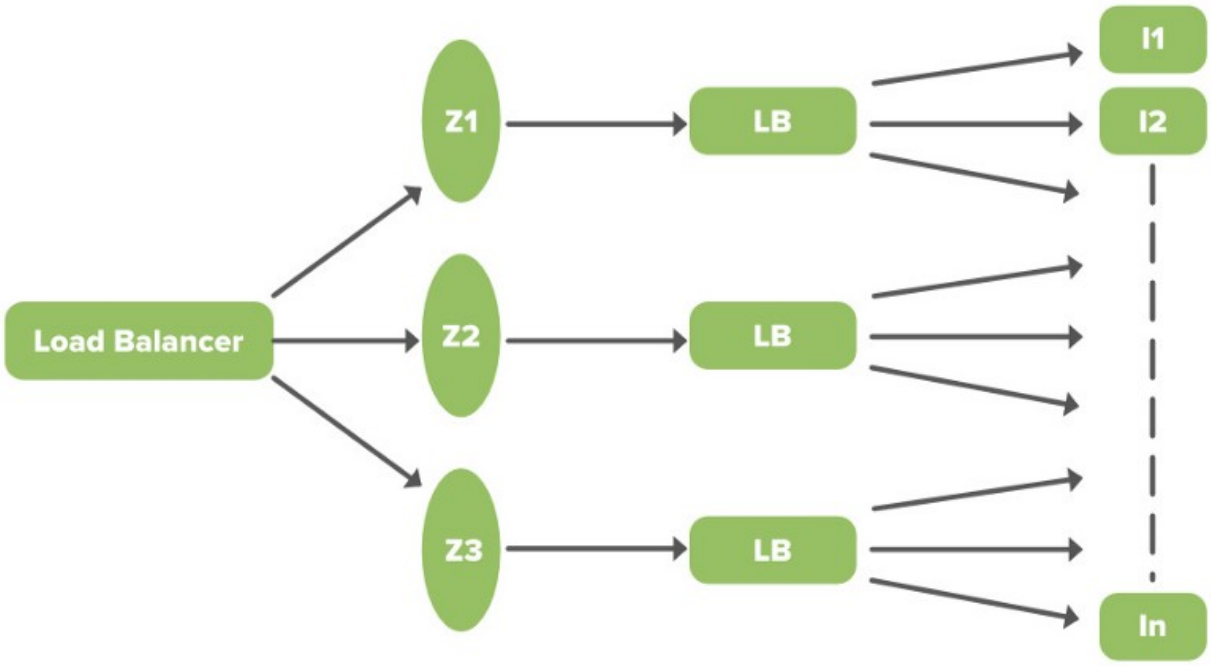
9. Resiliency

- **Replication** — Replicate database servers in a master-slave configuration. If one of the nodes does down, others takes place and service and continues functioning as expected.
- **Queuing** — Using while processing the uploaded videos.



10. Load Balancing

- Multiple servers behind a load balancer, there would be redundancy. The load balancer will continuously do a health check on servers behind it. If any server dies, the load balancer will stop forwarding the traffic to it and remove it from the cluster. This ensures the requests don't fail due to an unresponsive server.



The load balancer is responsible for routing the traffic to frontend services. **Elastic load balancing**, ELB performs a 2-tier load-balancing scheme where the load is balanced over zones first and then instances (servers).

- The First-tier consist of basic DNS-based Round Robin Balancing. When the request lands on the first load balancing, it is balanced across one of the zones (using round-robin) that your ELB is configured to use.
- The second tier is an array of load balancer instances and it performs the Round Robin Balancing technique to distribute the request across the instances that are behind it in the same zone.

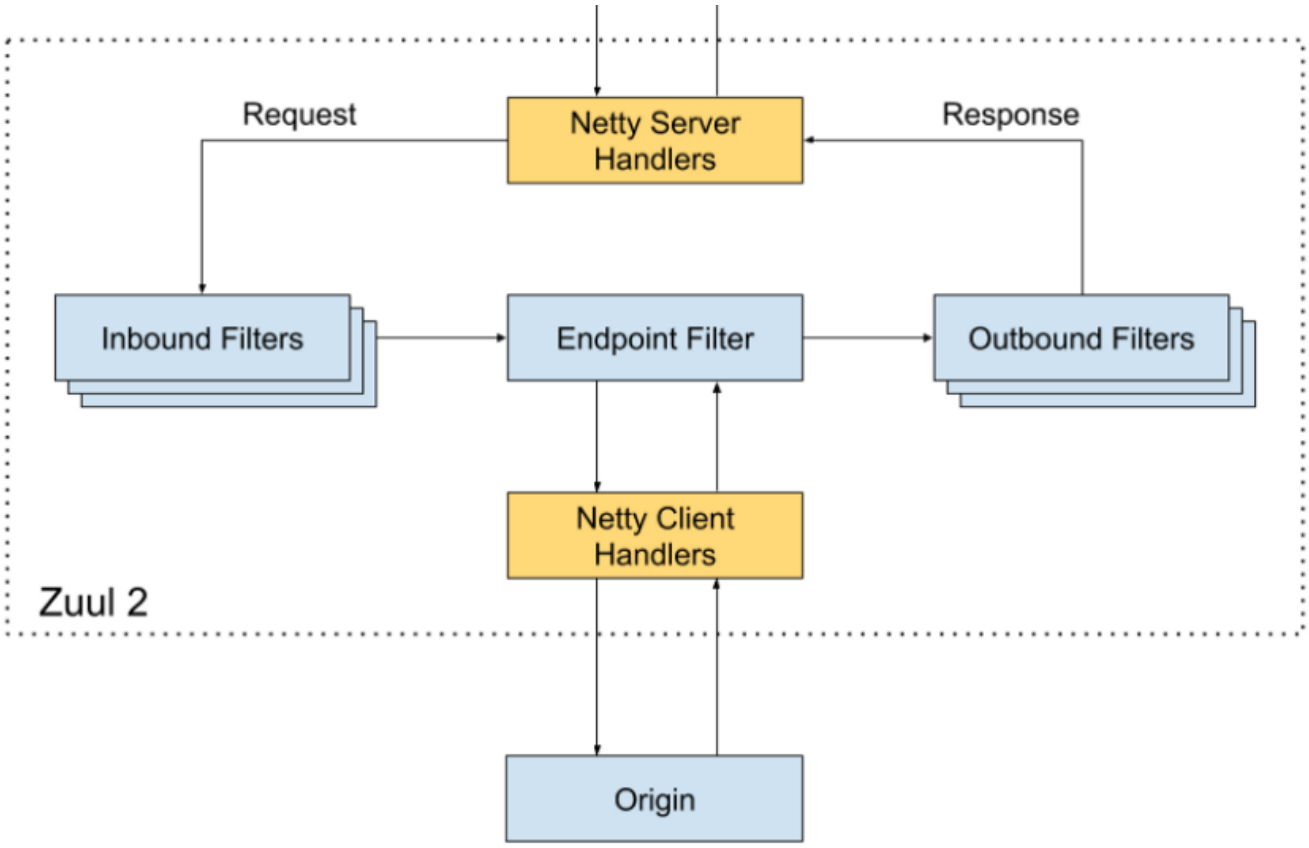
11. Geo-redundancy

- Deploy exact replica of our services in data centers across multiple geographical locations. The traffic could still be served from the remaining data centers.

12. ZUUL

provides dynamic routing, monitoring, resiliency, and security. It provides easy routing based on query params, URL paths.

- **Netty server** takes the responsibility to handle the network protocol, web server, connection management, and proxying work. When the request will hit the Netty server, it will proxy the request to the inbound filter.
- **The inbound filter** is responsible for authentication, routing, or decorating the request. Then it forwards the request to the endpoint filter.
- **Endpoint filter** is used to return a static response or to forward the request to the backend service (or origin as we call it). Once it receives the response from the backend service, it sends the request to the outbound filter.
- **The outbound filter** is used for zipping the content, calculating the metrics, or adding/removing custom headers. After that, the response is sent back to the Netty server and then it is received by the client.



Advantages:

- You can create some rules and **shard the traffic** by distributing the different parts of the traffic to different servers.

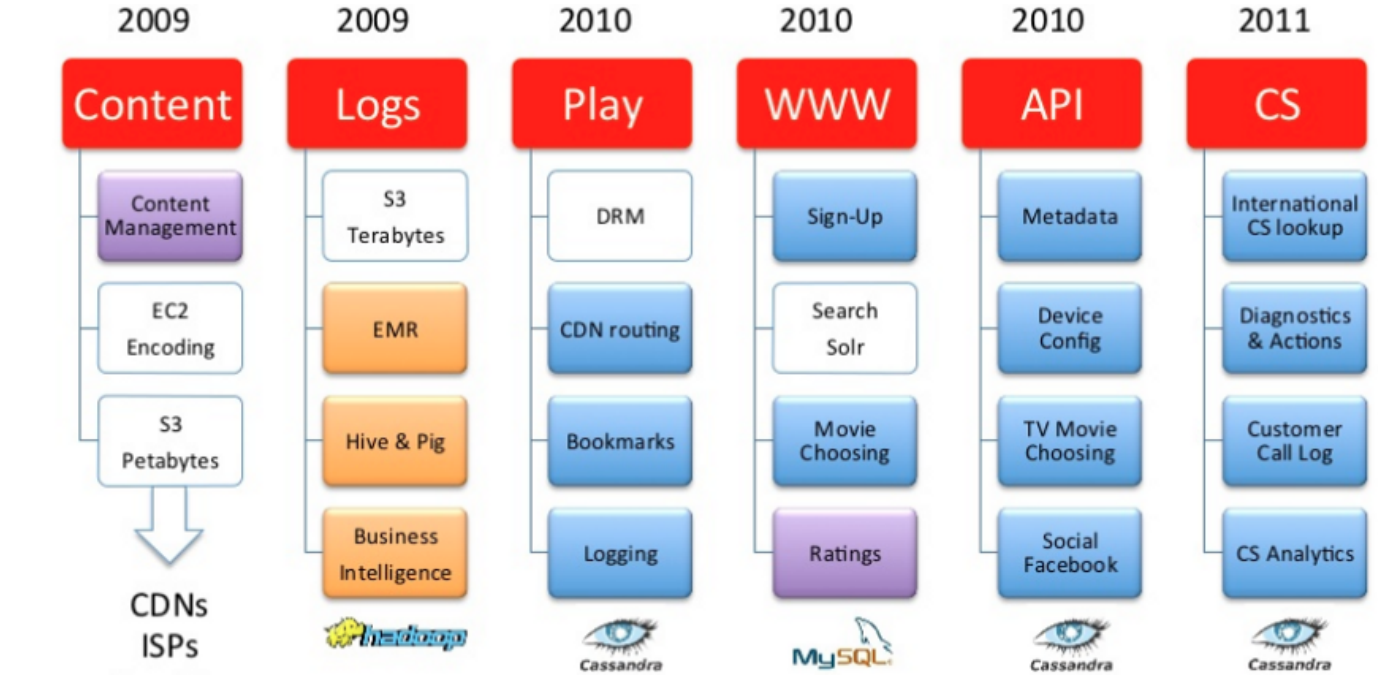
- Developers can also do **load testing** on newly deployed clusters in some machines. They can route some existing traffic on these clusters and check how much load a specific server can bear.
- You can also **test new services**. When you upgrade the service and you want to check how it behaves with the real-time API requests, in that case, you can deploy the particular service on one server and you can redirect some part of the traffic to the new service to check the service in real-time.
- We can also **filter the bad request** by setting the custom rules at the endpoint filter or firewall.

13. Hystrix

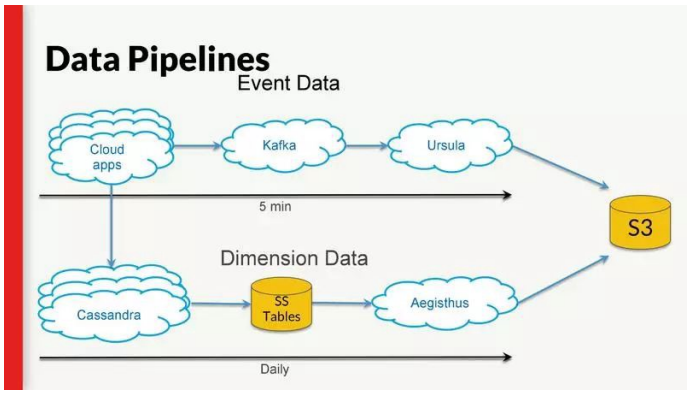
In a complex distributed system a server may rely on the response of another server. Dependencies among these servers can create latency and the entire system may stop working if one of the servers will inevitably fail at some point. To solve this problem we can isolate the host application from these external failures. Hystrix library is designed to do this job. It helps you to control the interactions between these distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, remote system, and 3rd party libraries. The library helps in.

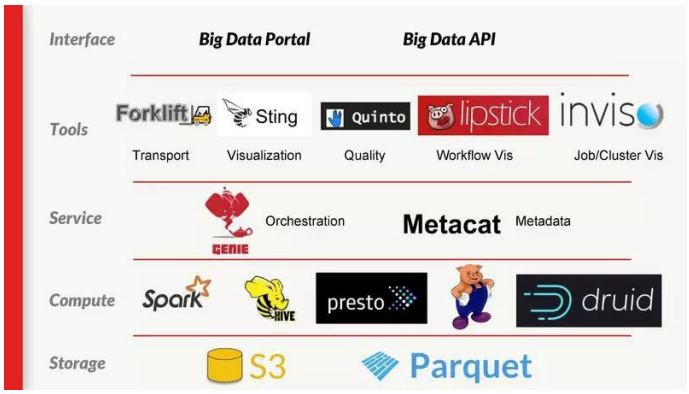
- Stop cascading failures in a complex distributed system.
- control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
- Fail fast and rapidly recover.
- Fallback and gracefully degrade when possible.
- Enable near real-time monitoring, alerting, and operational control.
- Concurrency-aware request caching. Automated batching through request collapsing

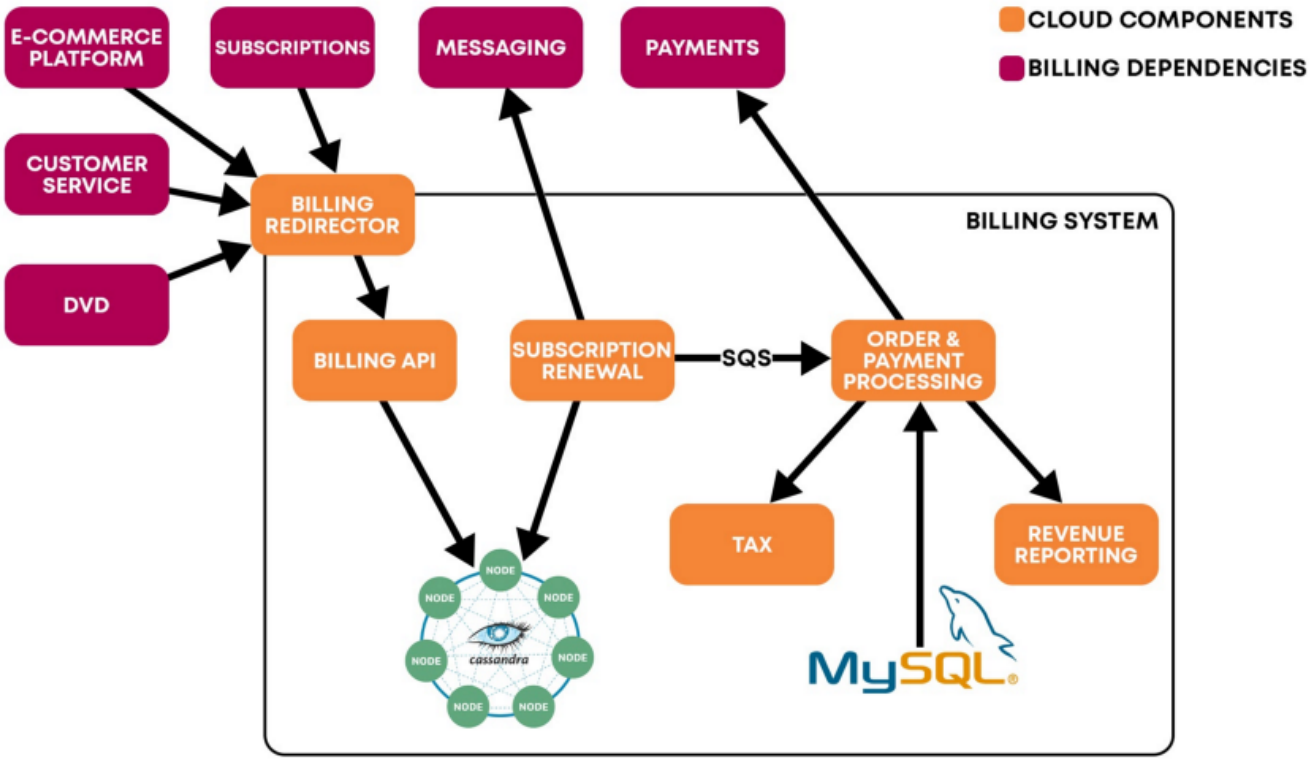
Databases Components



Netflix uses different DB to store different types of files, such as SQL and NoSQL for different purposes.

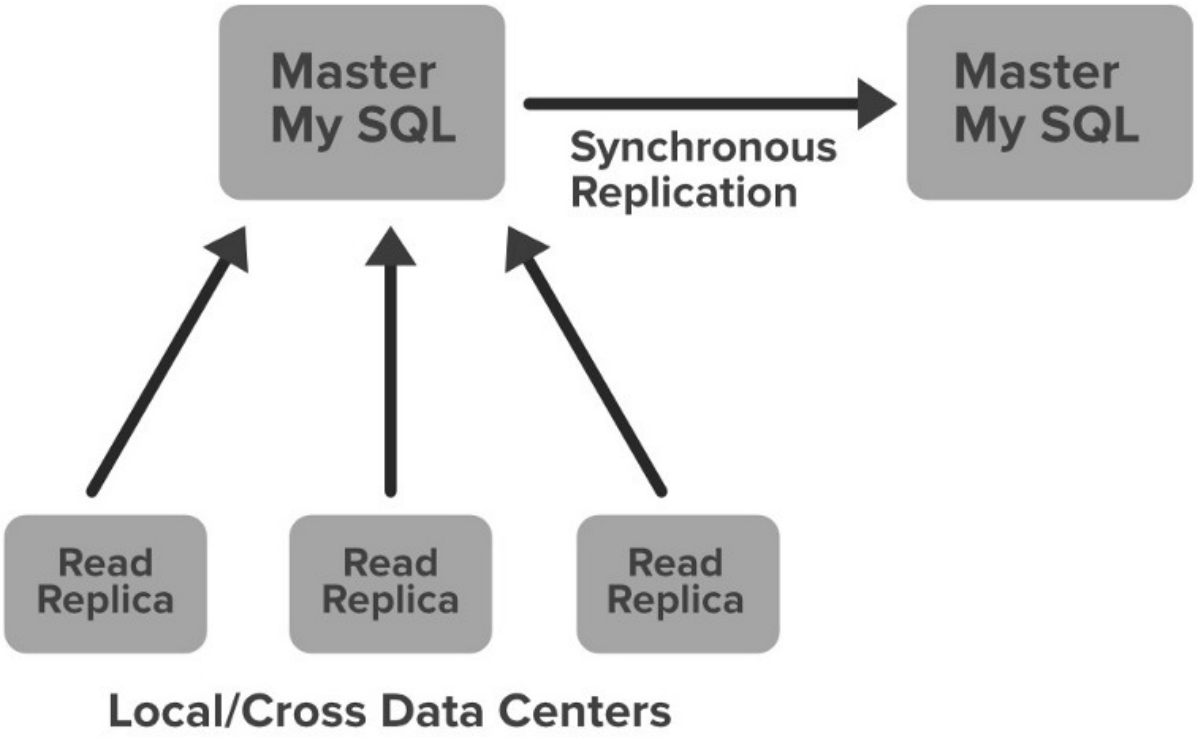


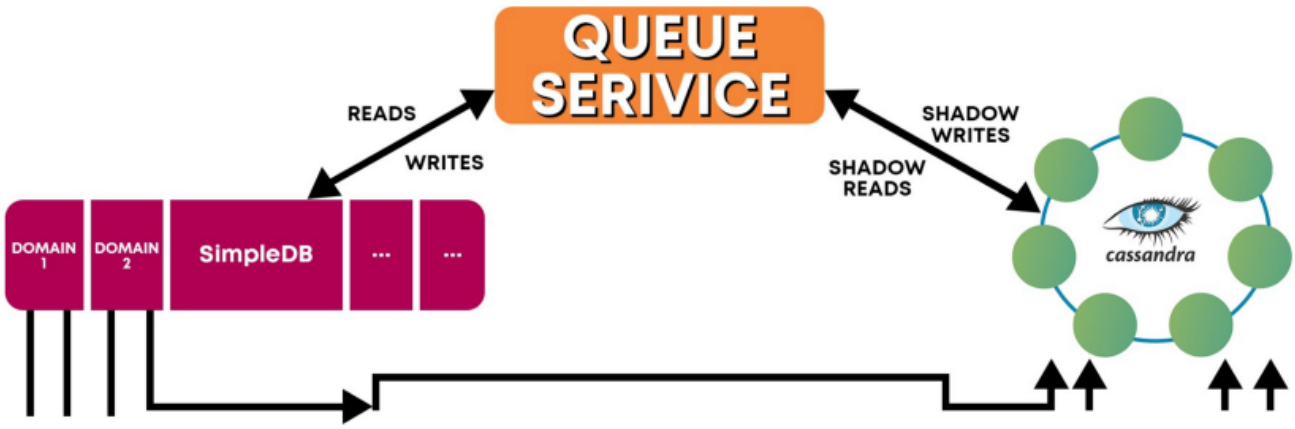




1. **MySQL**

- manage movies titles, billing, and transaction purpose because it needs ACID compliance
- AWS EC2 deployed MySQL to store the data
- Netflix has a master-master setup for MySQL because MySQL is built using the InnoDB engine over large AWS EC2 instances.
- The setup follows the "**Synchronous replication protocol**". Replicate the data is done synchronously, which states that there is a master-master relationship between nodes, and any write operation on the primary node will be considered as done only if that data is synchronized by both local and remote nodes to ensure high availability. Read queries are not handled by the primary node, it's handled by replicas, only write queries are handled by master DB. In case of failover, the secondary node will take up as the master node and will handle the write query well.



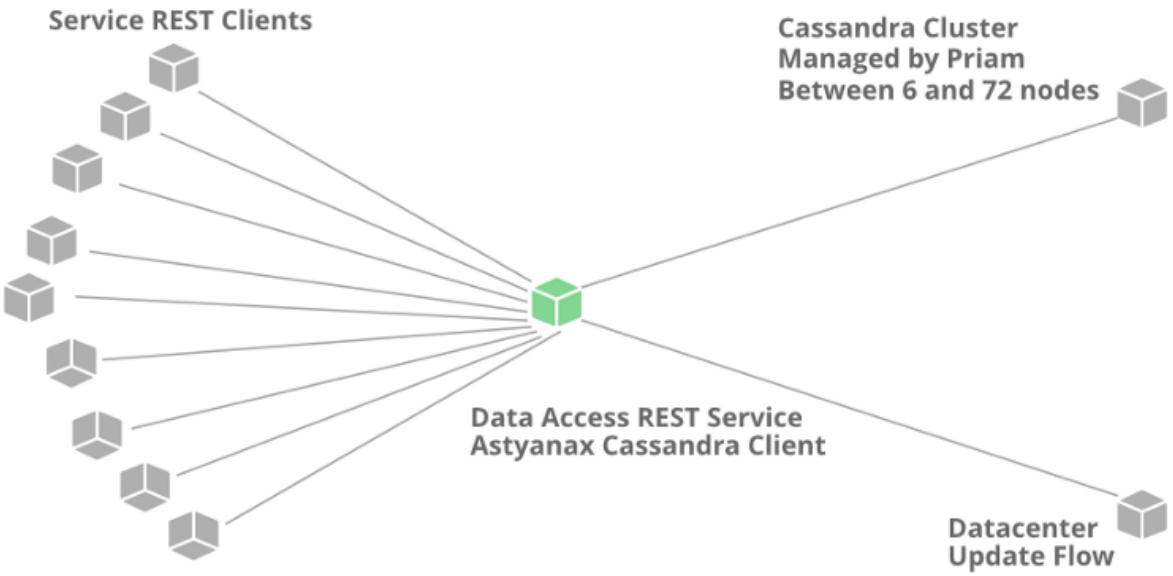


2. Cassandra (NoSQL)

- Cassandra is an open source distributed column-based NoSQL DB that enables the storage of a large amount of data over servers. Netflix uses this because it requires such

DB to store user history. It enables the handling of large amounts of reading requests efficiently and optimizes the latency for large read requests. As the user base grew, it became difficult to store many rows of data, and it is costly and slow. So, Netflix was designed based on the time frame and recent use for the new DB.

- When Netflix started acquiring more users, the viewing history data for each member also started increasing.
- Smaller Storage Footprint.
- Consistent Read/Write Performance as viewing per member grows (viewing history data write to read ratio is about 9:1 in Cassandra).
- **Total Denormalized Data Model**
- Over 50 Cassandra Clusters
- Over 500 Nodes
- Over 30TB of daily backups
- Biggest cluster 72 nodes.
- 1 cluster over 250K writes/s
- Initially, the viewing history was stored in Cassandra in a single row. When the users started increasing on Netflix the row sizes as well as the overall data size increased. This resulted in high storage, more operational cost, and slow performance of the application. The solution to this problem was to compress the old rows…



- LiveVH (Live viewing history) — only recent data with frequent updates, smaller numbers of rows are stored in an uncompressed form that is used for many operations like analysis, recommendations to the user after performing ETL (extracting, transforming, and loading).
- CompressedVH(compressed viewing history) — old data of browsing history and viewed by users is stored after compressing with occasional updates. Storage size is also decreased, and only one column per row was stored.

Database Schema

Users	
PK	**user_id**
	name
	email
	password
	created_at
	updated_at
	last_login_at

Profiles	
PK	**profile_id**
FK	user_id
	name
	created_at
	updated_at

Subscriptions	
PK	**subscription_id**
FK	user_id
FK	plan_id
	valid_till
	created_at
	updated_at

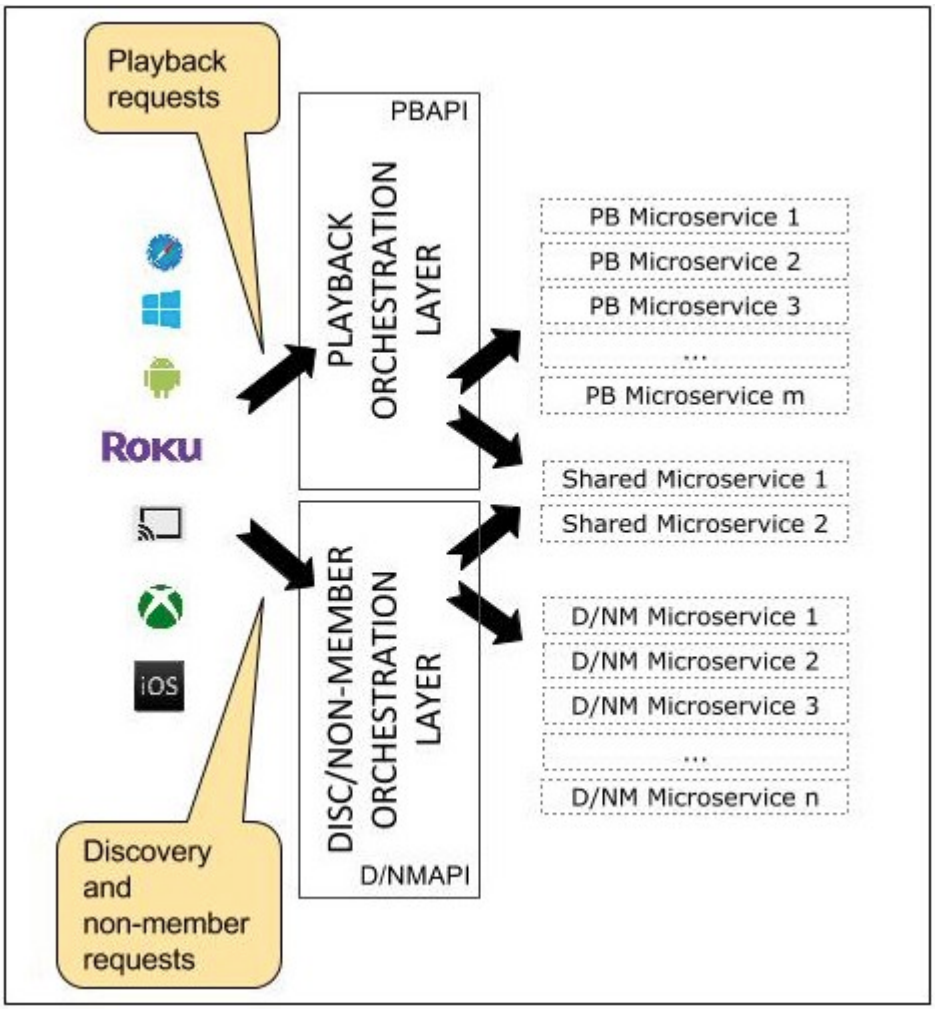
Videos	
PK	**video_id**
	title
	summary
	url
	length
	censor_rating
	created_at
	updated_at

Views	
PK	**view_id**
FK	user_id
FK	video_id
	watched_till
	updated_at

Plans	
PK	**plan_id**
	name
	charges

APIs

- REST APIs is used



User-sign-up

Request:

```
POST /api/v1/users
X-API-Key: api_key
{
  name:
  email:
  password:
}
```

Post HTTP method for this API because we create a resource or create a new entry in the DB. X-API-key which is passed to the HTTP header is the API key to identify different clients and do rate limiting.

Response:

```
201 Created
{
  message:
}
```

HTTP response code 201 tells that the user has signed up successfully. Other possible HTTP codes for failure cases.

```
400 Bad Request
409 Conflict
500 Internal Server Error
```

User-Sign-in

Request:

```
POST /api/v1/users/session
X-API-Key: api_key
{
  email:
  password:
}
```

Response:

```
200 OK
{
  auth_token:
}
```

API should return an auth_token which can be passed in the header for successive API calls which require authentication. auth_token can be generated using JWT.

JWT.IO - JSON Web Tokens Introduction

NEW: get the JWT Handbook for free and learn JWTs in depth! JSON Web

User-sign-out

Request:

```
DELETE /api/v1/users/session
X-API-Key: api_key
Authorization: auth_token
```

Delete HTTP method is used because it will delete a row entry in the DB which means we are terminating a session.

Response:

```
200 OK
```

HTTP response code 200 indicates successful sign-out.

Subscribe

Request:

```
POST /api/v1/subscription
X-API-Key: api_key
Authorization: auth_token
```

The post HTTP method is used because it creates a new subscription. We would be passing auth-token in the Authorization header for authenticating the user.

Response:

```
201 Created
{
  subscription_id:
  plan_name:
  valid_till:
}
```

201 HTTP response code is given along with subcription_id, plan_name, and valid_till to render these in the User interface.

Possible failure HTTP response codes:

```
401 Unauthorized
400 Bad request
```

Unsubscribe

Request:

```
DELETE /api/v1/subscription
X-API-Key: api_key
Authorization: auth_token
```

Delete HTTP method is used because we are canceling the subscription, means we will delete a row entry from the subscription DB.

Response:

```
200 OK
```

HTTP code `200` means it is done successfully.

Get videos

Request:

```
GET /api/v1/videos?page_id=<page_id>
X-API-Key: api_key
Authorization: auth_token
```

This API is used to render the home page of the application once logged in. This API contains recommended video which is determined by machine learning models. `page_id` would be used for pagination in API and `next_page_id` would be used for requesting results from the next page.

Response:

```
200 OK
{
  page_id:
  next_page_id:
  videos: [
    {
      id:
      title:
      summary:
      url:
      watched_till:
    },...
  ]
}
```

HTTP status code `200` mean successful operation.

Some failure status codes :

```
401 Unauthorized
500 Bad request
429 Too many requests
```

HTTP status code `429` mean users hit the rate limit and should wait some time to make requests again to avoid Denial of service (DOS) attacks.

Search API

Request:

```
GET /api/v1/search?q=<query>&page_id=<page_id>
X-API-Key: api_key
Authorization: auth_token
```

Search a video by title

Response:

```
200 OK
{
  page_id:
  next_page_id:
  videos: [
    {
      id:
      title:
      summary:
      url:
      watched_till:
    },...
  ]
}
```

200 HTTP response codes mean successful operations. then list down id, title, summary, URL, and watched_till. might be found none too.

Get video

Request:

```
GET /api/v1/videos/:video_id
X-API-Key: api_key
Authorization: auth_token
```

Play a particular video

Response:

```
200 OK
{
  id:
  title:
  summary:
  url:
  watched_till:
}
```

200 HTTP response code mean matched

Some failure HTTP status codes:

```
401 Unauthorized
404 Video not found
429 Too many requests
500 Internal server error
```

Upload API

Request:

```
POST /api/v1/videos
X-API-Key: api_key
Authorization: auth_token
{
  title:
  summary:
  censor_rating:
```

```
  video_contents:
}
```

Upload video from backend

Response:

```
202 Accepted
{
  video_url:
}
```

HTTP status code 202, which means that the video has been enqueued for asynchronous processing and quality checks. The results of the processing can send to users via email or other alerting mechanisms.

Some HTTP failure scenarios:

```
401 Unauthorized
400 Bad request
500 Internal server error
```

Update watched till timestamp

Request:

```
PUT /api/v1/videos/:video_id/watched_till
X-API-Key: api_key
Authorization: auth_token
{
  watched_till:
}
```

PUT HTTP method is used because we replace the other data with other row entries in the same database table or we are updating a resource on the server. This API would be used to

update the timestamp till the user has watched a particular video.

Response:

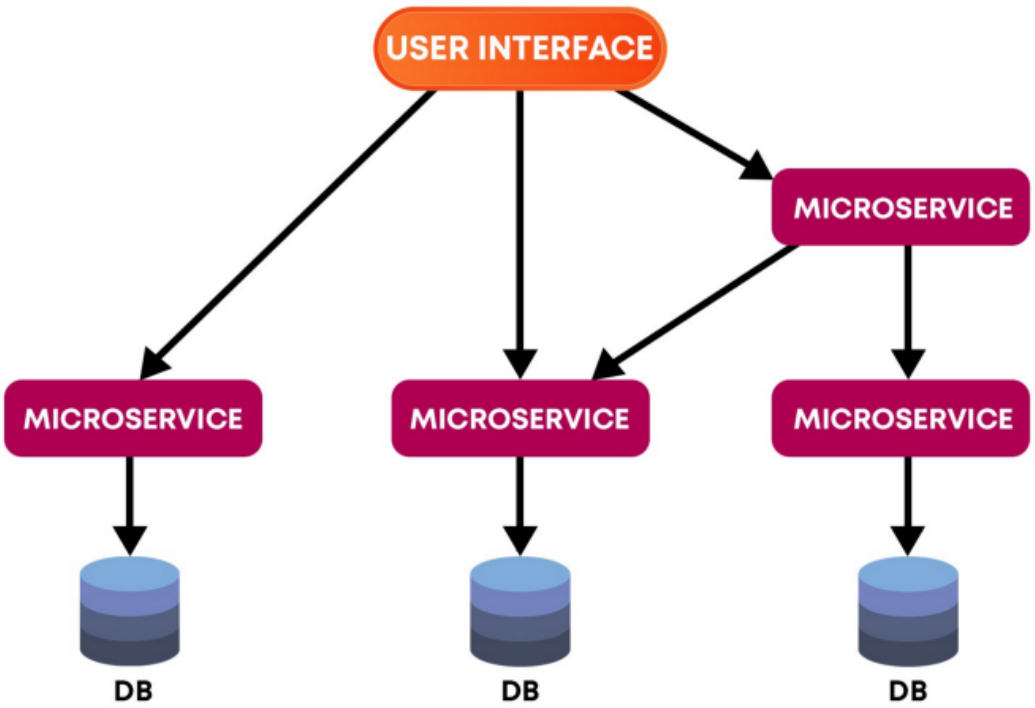
```
200 OK
```

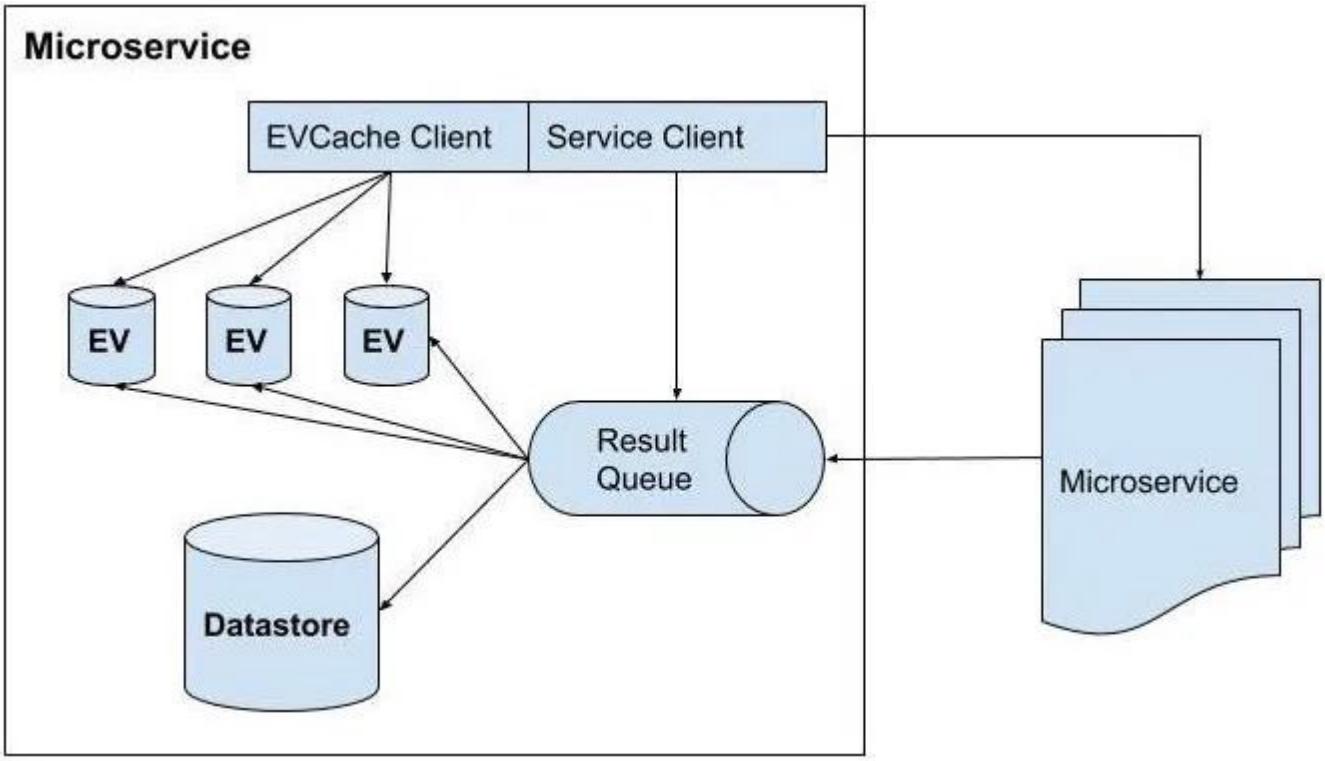
HTTP status code 200 means successful operation.

Other failure HTTP status codes:

```
401 Unauthorized
400 Bad request
500 Internal server error
```

Microservice Architecture

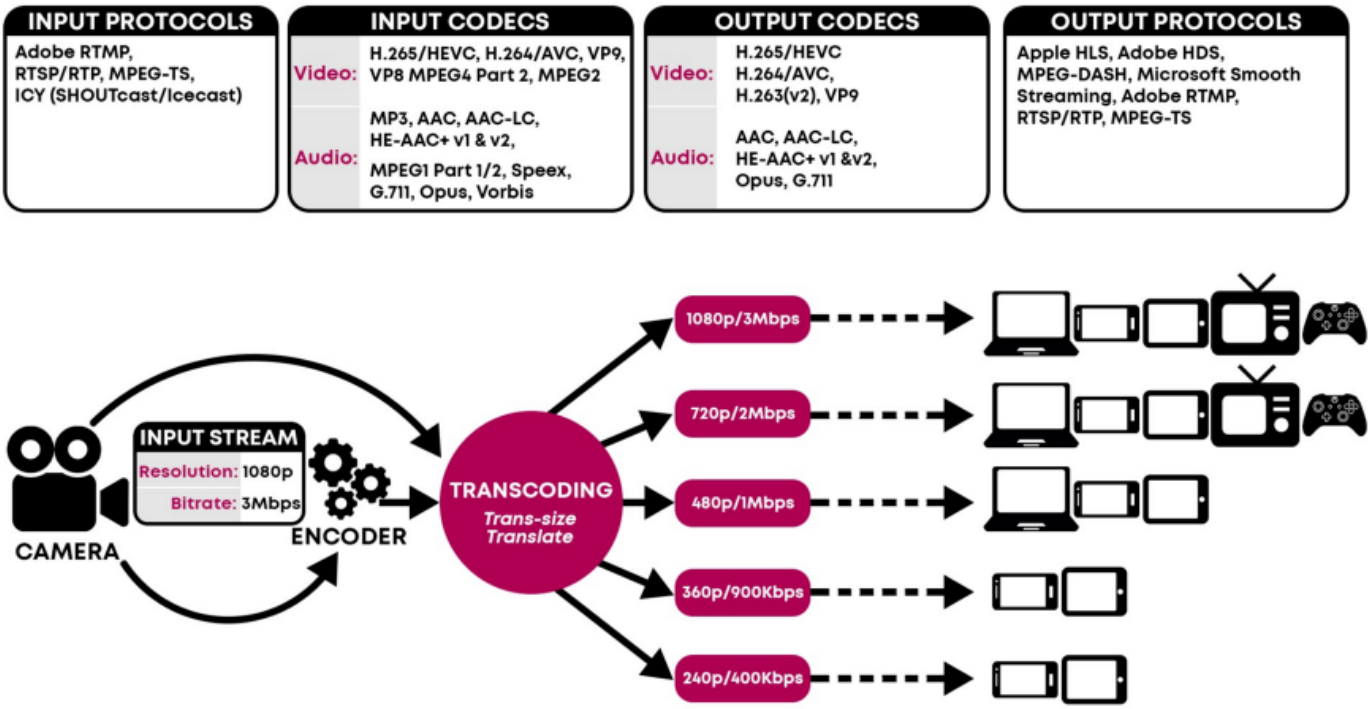




Microservices are used for faster deployments as any change in services can be done easily. The performance of each service can be tracked and if there is any issue, then it can be quickly isolated from other running services.

1. Critical services — to serve the users who interact with them very frequently the service. These services are kept independent of other services so that for any fail-over, users can continue to perform basic operations.

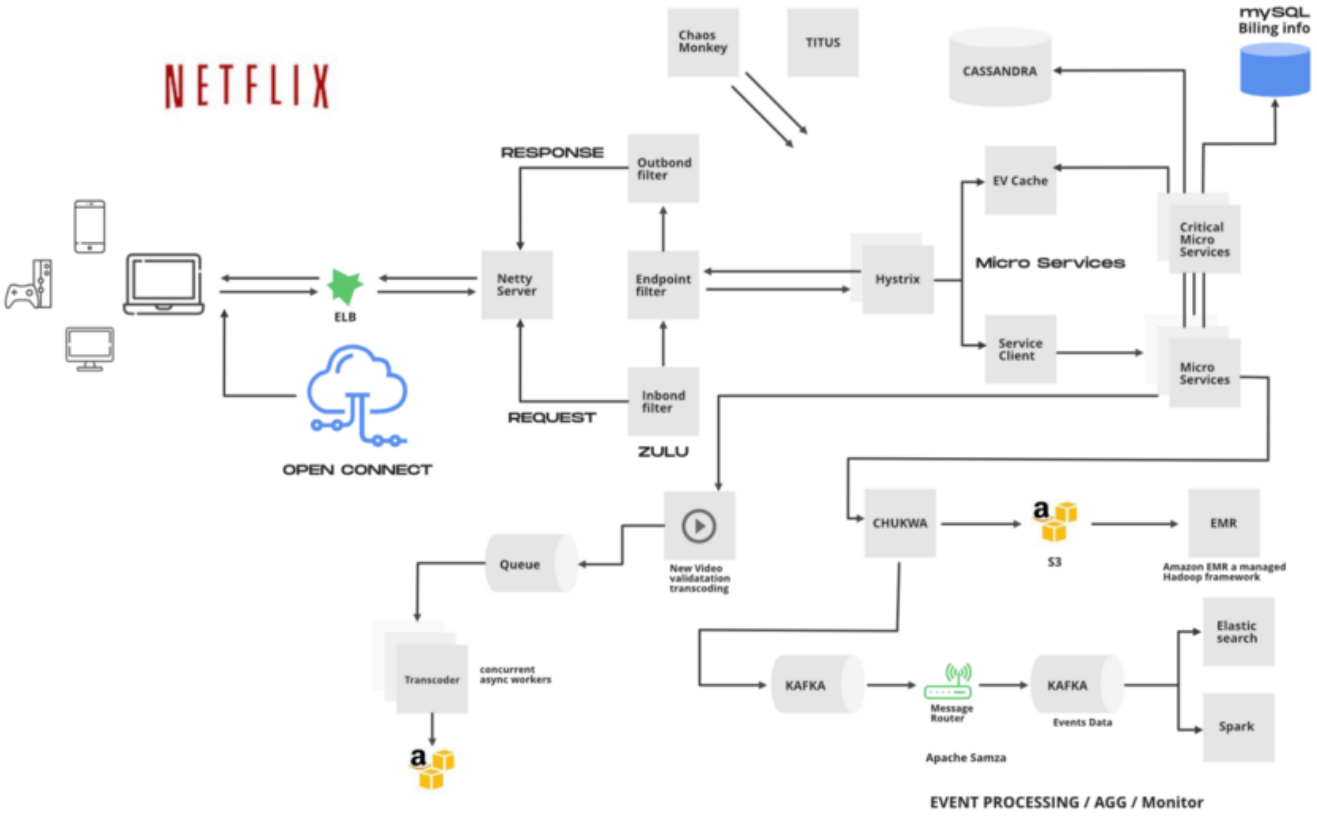
2. Stateless services — to serve API requests to clients, in such a way that they continue to work with others instances even if any server fails. This ensures high availability. For example, REST API server users the most.



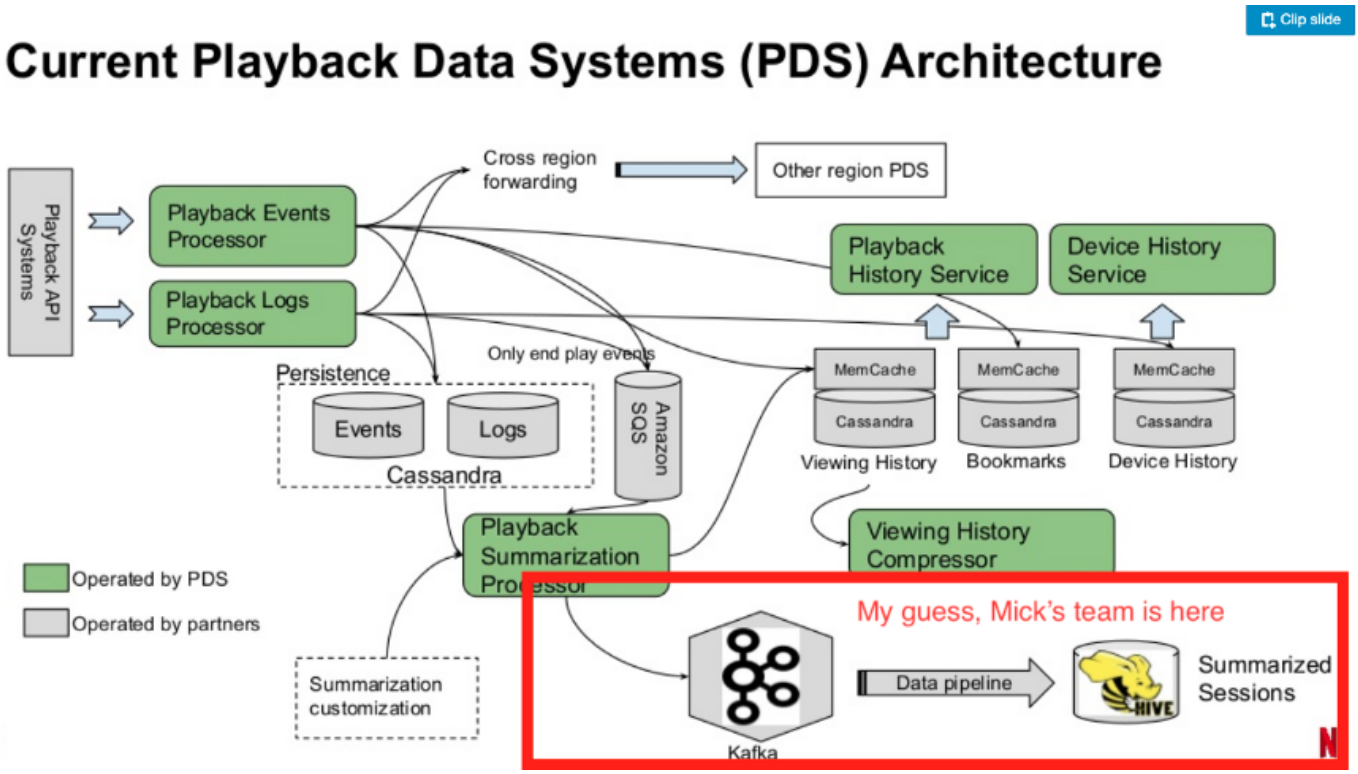
Onboarding Content

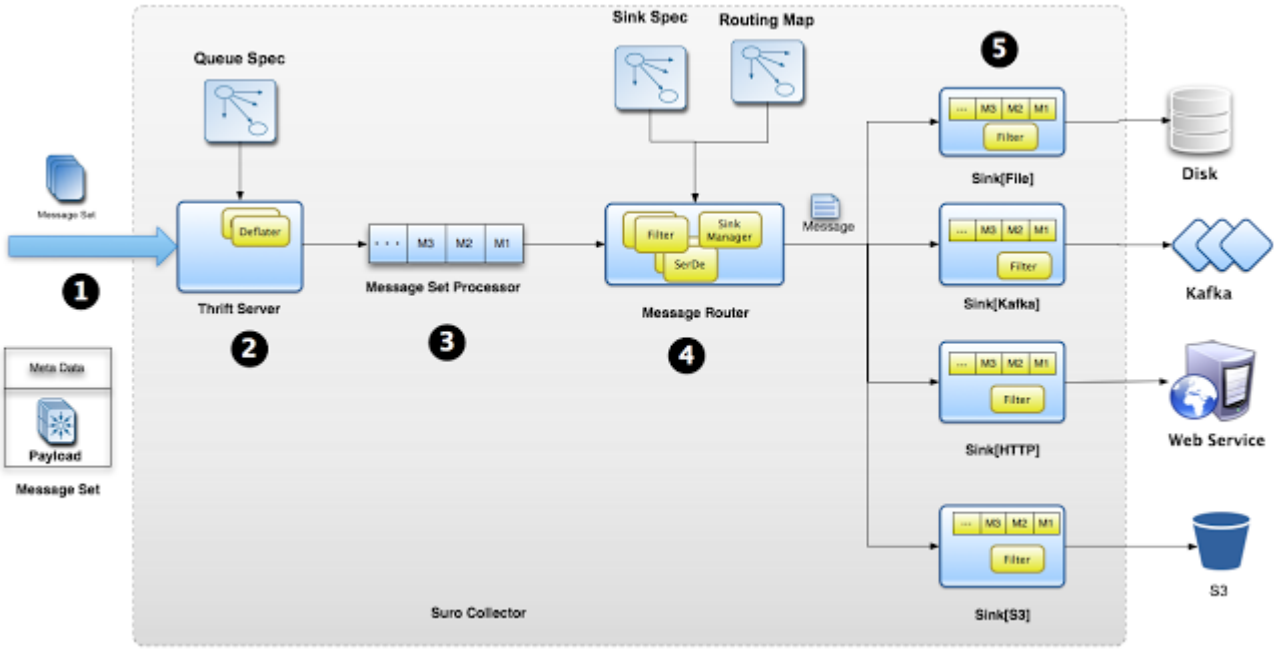
Content is a movie or show in video format. The processing unit is input protocols, input codecs, output codecs, and output protocols to serve the various devices and varying network speeds. The quality of a video is good when you are watching the video at high network speed. Netflix creates multiple replicas (approximate 1100–1200) for the same movie with different resolutions. Netflix breaks the original video into different smaller chunks and uses parallel workers in AWS to convert the chunks into different formats. Those process units are to help encoding or transcoding which is to convert the video from one format to another like changing resolutions, aspect ratio, reducing the file size, etc. After transcoding, once we have multiple copies of the files for the same movie, these files are transferred to the open connect server.

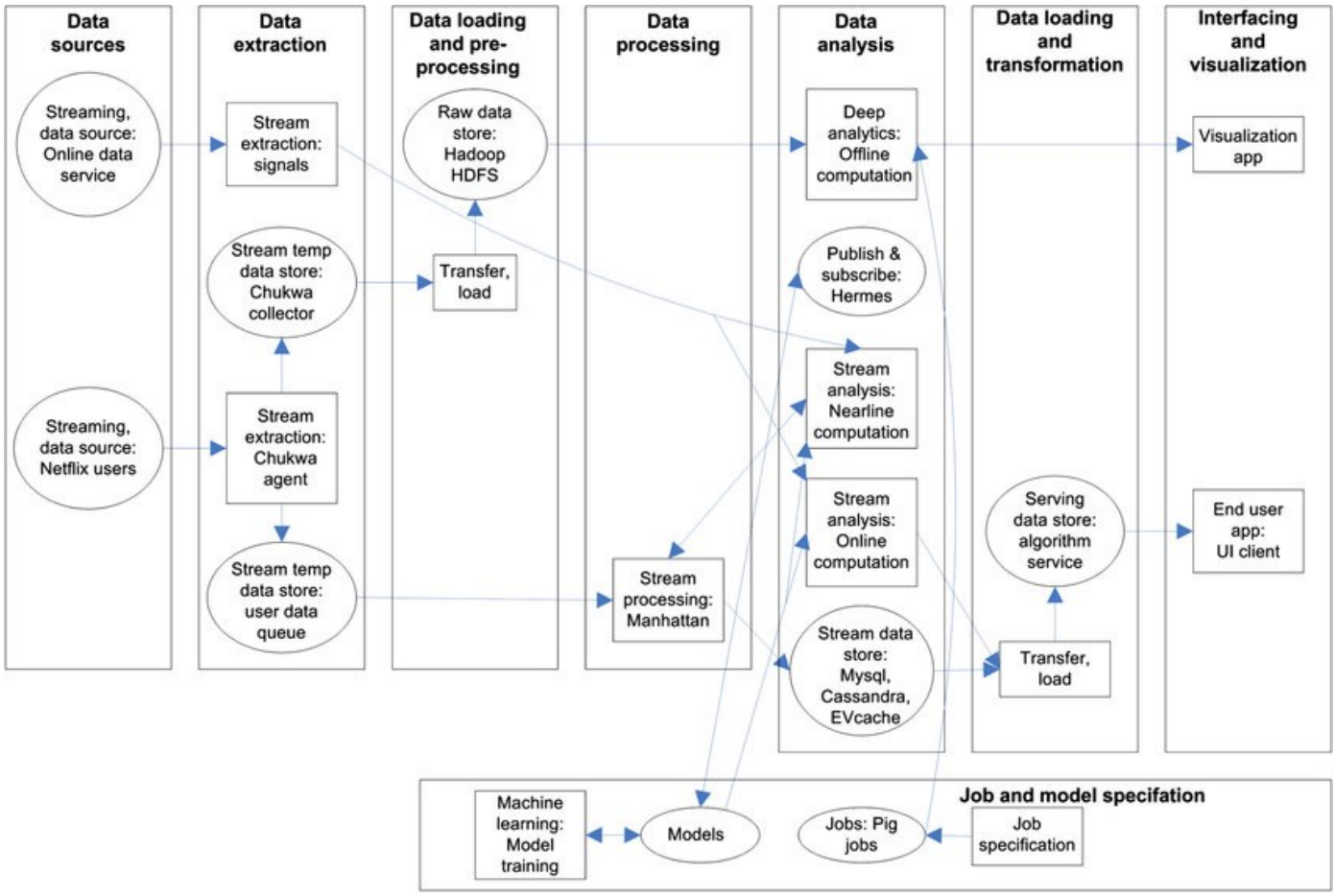
The high-level system architecture



Data Architecture





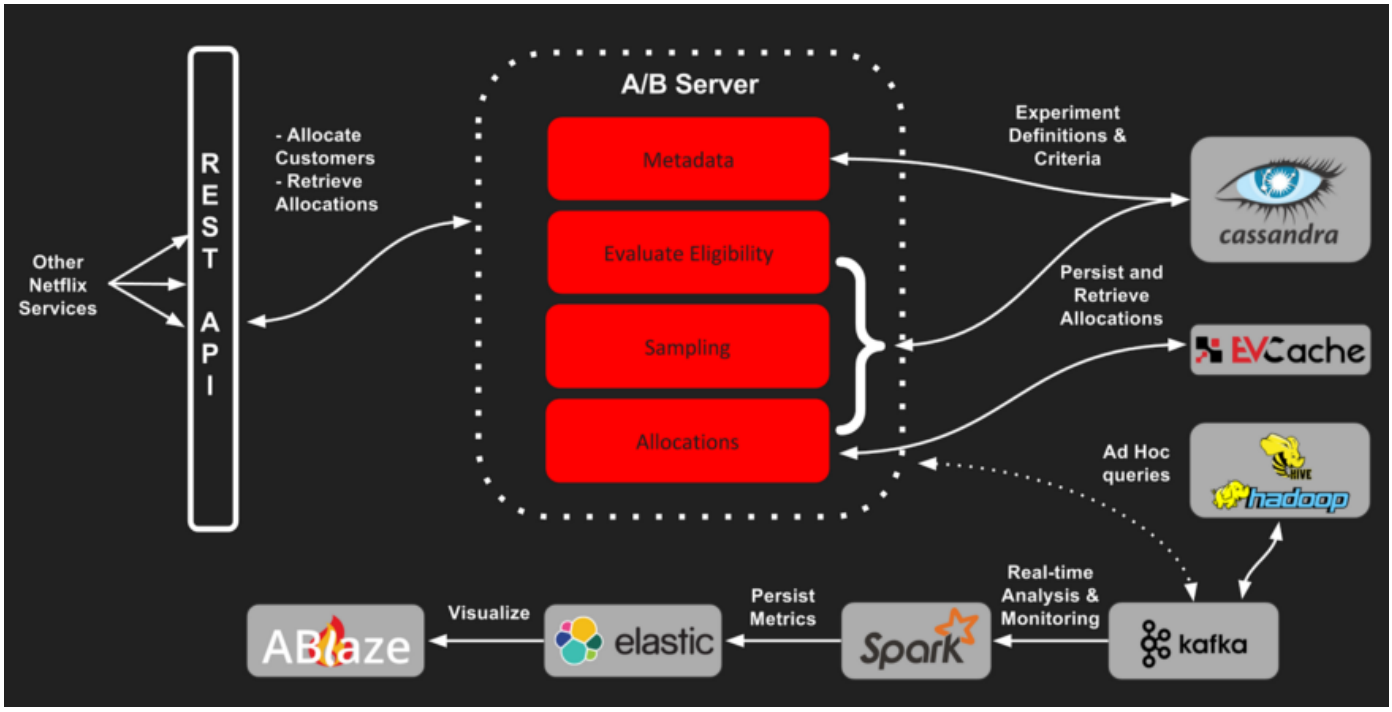


Movie Recommendation

- Apache Spark and machine learning is used for movie recommendation. When you load the front page you watch, there are multiple rows of different kinds of movies.
- (Artwork) These images are called header images (thumbnail). Netflix wants maximum clicks for the videos from the users and these clicks are dependent on the header images. Netflix has to choose the right compelling header image for a specific video. To do that Netflix creates multiple artworks for a specific movie and they display these images to the users randomly. For the same movie, images can be different for different users. Based on your preferences and viewing history Netflix predicts what kind of movies you like best or which actors you like the most in a movie. According to users' tastes, the images will be displayed to them.
- Netflix analyzes the data and decides what kind of rows or what kind of movies should be displayed to you. This is based on the user's historical data and preferences. Also, Netflix performs sorting of the movies and calculates the relevance ranking of these movies available on their platform. A majority of the machine learning pipelines are run on these large spark clusters. These pipelines are then used to do row selection, sorting, title relevance ranking, and artwork personalization among others. When you open the Netflix front page you might have noticed the images for each video.
- Now, Netflix calculates the number of clicks a certain image receives. If the clicks for the center image of the movie are 1, 500 times and the other images have fewer clicks then Netflix will make the center image as a header image for the movie Good Will Hunting forever. This is called **data-driven** and Netflix performs the data analytics with this approach. To make the right decision data is calculated based on the number of views associated with each picture.
- User interaction with the service (viewing history and how user rated other titles)
- Other members with similar tastes and preferences.
- Metadata information from the previously watched videos for a user such as titles, genre, categories, actors, release year, etc.
- The device of the user, at what time a user is more active, and for how long a user is active.
- 2 algorithms

1. Collaborative filtering: The idea of this filtering is that if two users have similar rating history then they will behave similarly in the future. For example, consider there are two-person. One person has liked the movie and rated the movie with a good score. Now, there is a good chance that the other person will also have a similar pattern and he/she will do the same thing which the first person has done.

2. Content-based filtering: The idea is to filter those videos which are similar to the video a user has liked before. Content-based filtering is highly dependents on the information from the products such as movie title, release year, actors, the genre. So to implement this filtering it's important to know the information describing each item and some sort of user profile describing what the user likes is also desirable.



Further Resources:

1. https://towardsdatascience.com/deep-dive-into-netflixs-recommender-system-341806ae3b48
2. https://netflixtechblog.com/announcing-suro-backbone-of-netflixs-data-pipeline-5c660ca917b6
3. https://keypointt.com/2020-05-16-Netflix-playback-dive-deep/
4. https://www.nexsoftsys.com/articles/how-netflix-backend-system-operates.html
5. https://elatov.github.io/2021/02/distributed-systems-design-netflix/
6. https://developpaper.com/design-and-analysis-of-netflix-microservice-architecture/
7. https://uxdesign.cc/netflix-system-design-ef5802426ad4
8. https://netflixtechblog.com/tagged/cloud-architecture
9. https://www.codingninjas.com/blog/2020/12/04/learn-what-is-rest-api-in-10-minutes/
10. https://about.netflix.com/en/news/how-netflix-works-with-isps-around-the-globe-to-deliver-a-great-viewing-experience
11. https://www.infoq.com/news/2019/01/netflix-evolution-architecture/
12. https://www.codekarle.com/system-design/netflix-system-design.html
13. http://highscalability.com/blog/2015/11/9/a-360-degree-view-of-the-entire-netflix-stack.html