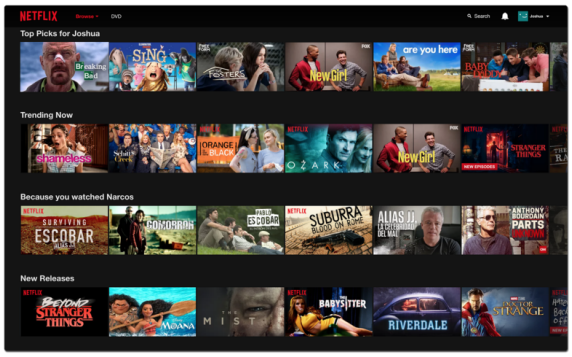
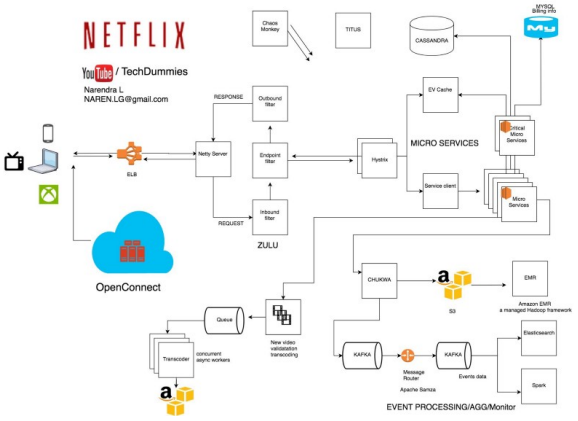


NETFLIX system design

Narendra L
Sep 8, 2018 · 15 min read

System Design:



If you don't want to read the whole article, I have a video on the same topic

Please subscribe to my channel: Tech Dummies (Thank you)

Netflix operates in two clouds: AWS and Open Connect.

Both clouds must work together seamlessly to deliver endless hours of customer-pleasing video.

Netflix has 3 main components which we are going to discuss today

- OC or netflix CDN
- backend

- client

Lets first talk about some high level working of Netflix and then jump in to these 3 components.

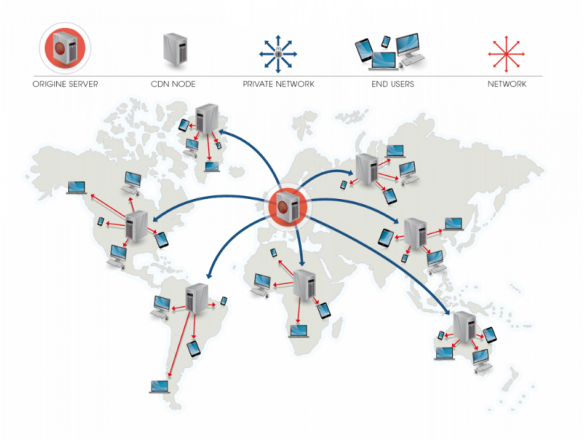
The client is the user interface on any device used to browse and play Netflix videos. TV, XBOX, laptop or mobile phone etc

Anything that doesn't involve serving video is handled in AWS.

Everything that happens after you hit play is handled by Open Connect. Open Connect is Netflix's custom global content delivery network (CDN).

Open Connect stores Netflix video in different locations throughout the world. When you press play the video streams from Open Connect, into your device, and is displayed by the client.

CDN—A content delivery network (CDN) is a system of distributed servers (network) that deliver pages and other Web content to a user, based on the geographic locations of the user, the origin of the webpage and the content delivery server.



Before explaining system design I will walk you through the high level data flow/system working of Netflix. This helps to better understand the system components

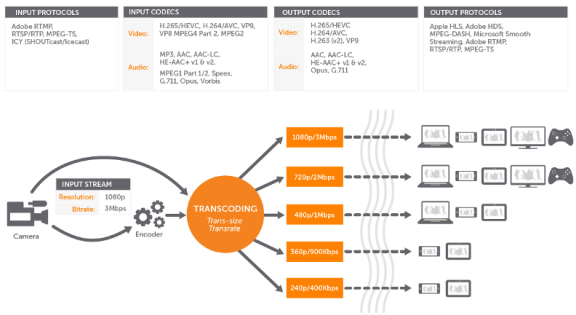
How Netflix onboard a movie/video:

Before this movie is made available to users, Netflix must convert the video into a format that works best for your device. This process is called transcoding or encoding.

Transcoding is the process that converts a video file from one format to another, to make videos viewable across different platforms and devices.

Whys do we need to do it? why can't we just play the source video?

The original movie/video comes in a high definition format that's many terabytes in size. Also, Netflix supports 2200 different devices. Each device has a video format that looks best on that particular device. If you're watching Netflix on an iPhone, you'll see a video that gives you the best viewing experience on the iPhone.



Netflix also creates files optimized for different network speeds. If you're watching on a fast network, you'll see the higher quality video than you would if you're watching over a slow network. And also depends on your Netflix plan. that said Netflix does create approx 1,200 files for every movie !!!!

That's a lot of files and processing to do transcoding Now we have all the files we need to stream it. OC Open connect comes in to picture, OC is Netflix own CDN no third-party CDN

Advantages of OC

- Less expensive
- Better quality
- More Scalable

So once the videos are transcoded these files are pushed to all of the OC servers.

Now the second scenario:

When the user loads Netflix app All requests are handled by the server in AWS Eg: Login, recommendations, home page, users history, billing, customer support etc. Now you want to watch a video when you click the play button of the Video. Your app automatically figures out the best OC server, best format and best bitrate for you and then the video is streamed from a nearby Open Connect Appliance (OCA) in the Open Connect CDN.

The Netflix apps are so intelligent that they constantly check for best streaming server and bitrate for you and switches between formats and servers to give the best viewing experience for you.

Now what Netflix does is with all of your searches, viewing, location, device, reviews and likes data on AWS it uses Hadoop | Machine learning models to recommend new movies which you might like…

And this cycle goes on and on

Netflix supports 2200 different devices, including Smart TV, Adroid, IOS, gaming consoles, web apps etc

All these apps are written in platform-specific code.

Netflix Likes React JS:

React was influenced by a number of factors, most notably: 1) startup speed, 2) runtime performance, and 3) modularity

ELB:

Netflix uses Amazons Elastic Load Balancer (ELB) service to route traffic to our front-end services. ELB's are set up such that load is

balanced across zones first, then instances. This is because the ELB is a two-tier load balancing scheme.

- The first tier consists of basic DNS based round robin load balancing. This gets a client to an ELB endpoint in the cloud that is in one of the zones that your ELB is configured to use.
- The second tier of the ELB service is an array of load balancer instances (provisioned directly by AWS), which does round-robin load balancing over our own instances that are behind it in the same zone.

ZUUL:

The Netty handlers on the front and back of the filters are mainly responsible for handling the network protocol, web server, connection management and proxying work. With those inner workings abstracted away, the filters do all of the heavy lifting.

The inbound filters run before proxying the request and can be used for authentication, routing, or decorating the request.

The endpoint filters can either be used to return a static response or proxy the request to the backend service (or origin as we call it).

The outbound filters run after a response has been returned and can be used for things like gzipping, metrics, or adding/removing custom headers.

Features:

- Supports http2
- mutual TLS
- Adaptive retries
- Concurrency protection for the origin

It helps in Easy routing based on query params, URL, path. The main use case is for routing traffic to a specific test or staging cluster.

Advantages: ??

- Services needing to shard their traffic create routing rules that map certain paths or prefixes to separate origins
- Developers onboard new services by creating a route that maps a new hostname to their new origin
- Developers run load tests by routing a percentage of existing traffic to a small cluster and ensuring applications will degrade gracefully under load
- Teams refactoring applications migrate to a new origin slowly by creating rules mapping traffic gradually, one path at a time
- Teams test changes (canary testing) by sending a small percentage of traffic to an instrumented cluster running the new build
- If teams need to test changes requiring multiple consecutive requests on their new build, they run sticky canary tests that route the same users to their new build for brief periods of time
- Security teams create rules that reject “bad” requests based on path or header rules across all Zuul clusters

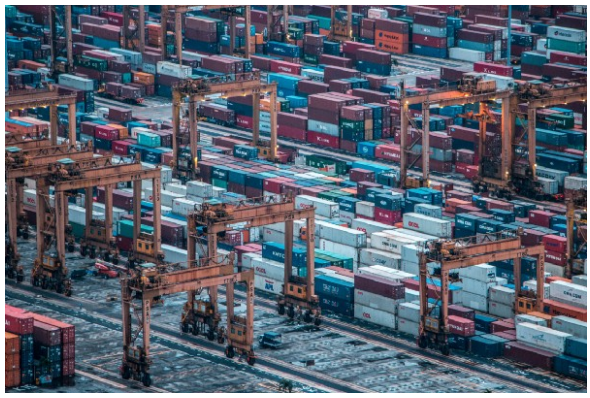
Hystrix:

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries. which

helps in

1. Stop cascading failures
2. Real-time monitoring of configurations changes
3. Concurrency aware request caching
4. Automated batching through request collapsing

IE. If a microservice is failing then return the default response and wait until it recovers.



"aerial photo of shipping container lot" by A J on Unsplash

Microservices:

How APIS gets the response?

Netflix uses MicroServices architecture to power all of the APIs needed for applications and Web apps. Each API calls the other micro-services for required data and then responds with the complete response

This setup works, but is it reliable?

1. We can use Hysterix which I already explained
2. We separate critical services

Critical Microservices:

What Netflix does is, they identify few services as critical (so that at last user can see recommended hit and play, in case of cascaded service failure) and these micro-services works without many dependencies to other services !!

Stateless Services:

One of the major design goals of the Netflix architecture's is stateless services.

These services are designed such that any service instance can serve any request in a timely fashion and so if a server fails it's not a big deal. In the failure, case requests can be routed to another service instance and we can automatically spin up a new node to replace it.

"orange and beige marble countertop" by Chris Leipelt on Unsplash

EVCache:

When a node goes down all the cache goes down along with it. so the performance hit until all the data is cached. so what Netflix did is they came up with EVcache. It is a wrapper around Memcached but it is sharded so multiple copies of cache is stored in sharded nodes. So every time the write happens, all the shards are updated too...

When cache reads happens, read from nearest cache or nodes, but when a node is not available, read from a different available node. It handles 30 million request a day and linear scalability with milliseconds latency.

SSDs for Caching:

Storing large amounts of data in volatile memory (RAM) is expensive. Modern disk technologies based on SSD are providing fast access to data but at a much lower cost when compared to RAM. Hence, we wanted to move part of the data out of memory without sacrificing availability or performance. The cost to store 1 TB of data on SSD is much lower than storing the same amount of RAM.

Database: (EC2 deployed MySQL)

EC2 MySQL was ultimately the choice for the billing/user info use case, Netflix built MySQL using the InnoDB engine large ec2 instances. They even had master-master like setup with "Synchronous replication protocol" was used to enable the write operations on the primary node to be considered completed. Only after both the local and remote writes have been confirmed.

As a result, the loss of a single node is guaranteed to have no data loss. This would impact the write latency, but that was well within the SLAs.

Read replica set up in local, as well as cross-region, not only met high availability requirements, but also helped with scalability.

The read traffic from ETL jobs was diverted to the read replica, sparing the primary database from heavy ETL batch processing. In case of the primary MySQL database failure, a failover is performed to the secondary node that was being replicated in synchronous mode. Once secondary node takes over the primary role, the route53 DNS entry for database host is changed to point to the new primary.

Cassandra: -> 500 nodes 50 clusters

Cassandra is a free and open-source distributed wide column store NoSQL database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure

At Netflix as userbase started to grow more there has been a massive increase in viewing history data.

Initial days it was fine, but not for long

So Netflix Redesigned data storage arch with two main goals in mind:

- Smaller Storage Footprint.
- Consistent Read/Write Performance as viewing per member grows.

So the solution: Compress the old rows!! Data was divided in to two types

Live Viewing History (LiveVH): Small number of recent viewing records with frequent updates. The data is stored in uncompressed form as in the simple design detailed above.

Compressed Viewing History (CompressedVH): Large number of older viewing records with rare updates. The data is compressed to reduce storage footprint. Compressed viewing history is stored in a single column per row key.

Kafka to chukwa for distribute system monitoring

Push all the netflix events to processing pipelines

~500 billion events and ~1.3 PB per day

~8 million events and ~24 GB per second during peak hours

What kind of events??

- Video viewing activities
- UI activities
- Error logs
- Performance events
- Troubleshooting & diagnostic events

Apache Chukwa is an open source data collection system for monitoring large distributed systems. Apache Chukwa is built on top of the Hadoop Distributed File System (HDFS) and Map/Reduce framework and inherits Hadoop's scalability and robustness.

Apache Chukwa also includes a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collected data.

The kafka routing service is responsible for moving data from fronting Kafka to various sinks: S3, Elasticsearch, and secondary Kafka.

Routing is done using apache Samza

When Chukwa sends traffic to Kafka, it can deliver full or filtered streams. Sometimes, we need to apply further filtering on the Kafka streams written from Chukwa. That is why we have the router to consume from one Kafka topic and produce to a different Kafka topic.

Elastic search:

We have seen explosive growth in Elastic search adoption within Netflix for the last two years. There are ~150 clusters totaling ~3,500 instances hosting ~1.3 PB of data. The vast majority of the data is injected via our data pipeline.

How its used at Netflix

Say when a customer tried to play a video and he couldn't, he calls the customer care now how customer care guys can debug whats happening? So Playback team uses Elastic search to drill down the problem and also to understand how widespread is the problem.

- To see Signup or login problems
- To keep track of resource usage

AWS Application Auto Scaling feature—TITUS

Titus is a container management platform that provides scalable and reliable container execution and cloud-native integration with Amazon AWS.

Titus was built internally at Netflix and is used in production to power Netflix streaming, recommendation, and content systems.

It also has scheduling support for service applications. Mainly used to scale docker images, It talks to AWS auto scale service using AWS API gateways to scale dockers on AWS.

AWS auto scale can scale instances, Titus will scale instances and also dockers based on the traffic conditions. As Netflix has many micro services on docker.

For example, as people on the east coast of the U.S. return home from work and turn on Netflix, services automatically scale up to meet this demand. Scaling dynamically with demand rather than static sizing helps ensure that services can automatically meet a variety of traffic patterns without service owners needing to size and plan their desired capacity. Additionally, dynamic scaling enables cloud resources that are not needed to be used for other purposes, such as encoding new content.

This design centered around the AWS Auto Scaling engine being able to compute the desired capacity for a Titus service, relay that capacity information to Titus, and for Titus to adjust capacity by launching new or terminating existing containers. There were several advantages to this approach. First, Titus was able to leverage the same proven auto scaling engine that powers AWS rather than having to build our own. Second, Titus users would get to use the same Target Tracking and Step Scaling policies that they were familiar with from EC2. Third, applications would be able to scale on both their own metrics, such as request per second or container CPU utilization, by publishing them to CloudWatch as well as AWS-specific metrics, such as SQS queue depth. Fourth, Titus users would benefit from the new auto scaling features and improvements that AWS introduces.

The key challenge was enabling the AWS Auto Scaling engine to call the Titus control plane running in Netflix's AWS accounts. To address this, we leveraged AWS API Gateway, a service which provides an accessible API "front door" that AWS can call and a backend that could call Titus. API Gateway exposes a common API for AWS to use to adjust resource capacity and get capacity status while allowing for pluggable backend implementations of the resources being scaled, such as services on Titus. When an auto scaling policy is configured on a Titus service, Titus creates a new scalable target with the AWS Auto Scaling engine.

Media processing while onboarding and later

Validating the video: The first thing Netflix does is spend a lot of time validating the video. It looks for digital artifacts, color changes, or missing frames that may have been caused by previous transcoding attempts or data transmission problems.

The video is rejected if any problems are found.

After the video is validated, it's fed into what Netflix calls the media pipeline.

A pipeline is simply a series of steps data is put through to make it ready for use, much like an assembly line in a factory. More than 70 different pieces of software have a hand in creating every video.

It's not practical to process a single multi-terabyte sized file, so the first step of the pipeline is to break the video into lots of smaller chunks.

The video chunks are then put through the pipeline so they can be encoded in parallel. In parallel simply means the chunks are processed at the same time

Archer:

Archer is an easy to use MapReduce style platform for media processing that uses containers so that users can bring their OS-level dependencies. Common media processing steps such as mounting video frames are handled by the platform. Developers write three functions: split, map and collect; and they can use any programming language. Archer is explicitly built for simple media processing at scale, and this means the platform is aware of media formats and gives white glove treatment for popular media formats.

For example, a PRORES video frame is a first class object in Archer and splitting a video source into shot based chunks [1] is supported out of the box (a shot is a fragment of the video where the camera doesn't move).

1. detecting dead pixels caused by defective digital camera
2. machine learning (ML) to tag audio
3. QC for subtitles

SPARK Usage for the movierecommendation

Spark is used for content recommendations and personalization. A majority of the machine learning pipelines for member personalization run atop large managed Spark clusters. These models form the basis of the recommender system that backs the various personalized canvases you see on the Netflix app including, title relevance ranking, row selection & sorting, and artwork personalization among others.

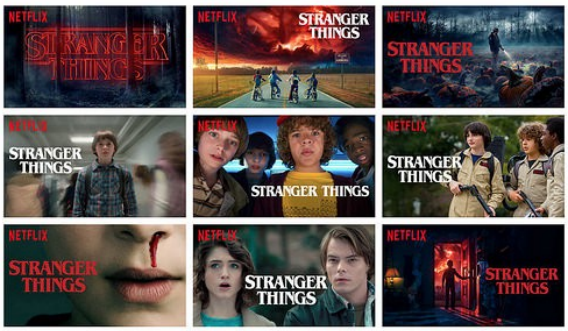
Netflix personalizes artwork just for you

Here's a great example of how Netflix entices you to watch more videos using its data analytics capabilities.

When browsing around looking for something to watch on Netflix, have you noticed there's always an image displayed for each video? That's called the header image.

The header image is meant to intrigue you, to draw you into selecting a video. The idea is the more compelling the header image, the more likely you are to watch a video. And the more videos you watch, the less likely you are to unsubscribe from Netflix.

Here's an example of different header images for Stranger Things:



You might be surprised to learn the image shown for each video is selected specifically for you. Not everyone sees the same image.

Everyone used to see the same header image. Here's how it worked. Members were shown at a random one picture from a group of options, like the pictures in the above Stranger Things collage. Netflix counted every time the video was watched, recording which picture was displayed when the video was selected.

For our Stranger Things example, let's say when the group picture in the center was shown, Stranger Things was watched 1,000 times. For all the other pictures, it was watched only once each.

Since the group picture was the best at getting members to watch, Netflix would make it the header image for Stranger Things forever.

This is called being data-driven. Netflix is known for being a data-driven company. Data is gathered—in this case, the number of views associated with each picture—and used to make the best decisions possible—in this case, which header image to select.

Clever, but can you imagine doing better? Yes, by using more data. That's the theme of the future—solving problems by learning from data.

You and I are likely very different people. Do you think we are motivated by the same kind of header image? Probably not. We have different tastes. We have different preferences.

Netflix knows this too. That's why Netflix now personalizes all the images they show you. Netflix tries to select the artwork highlighting the most relevant aspect of a video to you. How do they do that?

Remember, Netflix records and counts everything you do on their site. They know which kind of movies you like best, which actors you like the most, and so on.

How Netflix's Recommendations System Works

https://beta.vu.nl/nl/Images/wer...

Whenever you access the Netflix service, our recommendations system strives to help you find a show or movie to enjoy with minimal effort. We estimate the likelihood that you will watch a particular title in our catalog based on a number of factors including:

• your interactions with our service (such as your viewing history and how you rated other titles),

• other members with similar tastes and preferences on our service (more info here), and

• information about the titles, such as their genre, categories, actors, release year, etc.

In addition to knowing what you have watched on Netflix, to best personalize the recommendations we also look at things like:

• the time of day you watch,

• the devices you are watching Netflix on, and

• how long you watch.

Collaborative filtering The Collaborative Filtering (CF) algorithms are based on the idea that if two clients have similar rating history then they will behave similarly in the future (Breese, Heckerman, and Kadie, 1998). If, for example, there are two very likely users and one of them watches a movie and rates it with a good score, then it is a good indication that the second user will have a similar pattern

Content-based filtering The Content-based filtering (CB) aims to recommend items or movies that are alike to movies the user has liked before. The main difference between this approach and the CF is that CB offers the recommendation based not only in similarity by rating, but it is more about the information from the products (Aggarwal, 2016), i.e., the movie title, the year, the actors, the genre. In order to implement this methodology, it is necessary to possess information describing each item, and some sort of user profile describing what the user likes is also desirable. The task is to learn the user preferences, and then locate or recommend items that are “similar” to the user preferences

Hybrid filtering The hybrid methods are characterized by combining CF and CB techniques