Member-only story

A Design Analysis of Cloud-based Microservices Architecture at Netflix

A comprehensive system design analysis of microservices architecture at Netflix to power its global video streaming services

Cao Duc Nguyen · Follow

Published in The Startup · 19 min read · May 2, 2020

1. Introduction

Netflix has been among the best online subscription-based video streaming services in the world ([12]) for many years, accounting for over 15% of the world's Internet bandwidth capacity. In 2019, Netflix already acquired over 167 million subscribers, with more than 5 million new subscribers added every quarter, and operates in more than 200 countries. More specifically, Netflix's subscribers spend over 165 million hours of watching over 4,000 films and 47,000 episodes daily. These impressive statistics show us, from an engineering perspective point of view, ***Netflix technical teams have designed such an amazing video streaming system with very high availability and scalability in order to serve their customers globally***.

However, it took the technical teams over 8 years to have their IT systems as now ([1]). In fact, the infrastructure transformation at Netflix began in August 2008 after a service outage in its own data centers shutting the entire

DVD renting services down for three days. Netflix realized that it needs a more reliable infrastructure with no single point of failure. Therefore, it has made two important decisions: migrating the IT infrastructure from its data centers to a public ***cloud*** and replacing ***monolithic*** programs with small manageable software components by ***microservices architecture***. Both decisions have directly shaped today Netflix's success.

Netflix had chosen AWS cloud ([4]) to migrate its IT infrastructure because AWS could offer highly reliable databases, large-scale cloud storage and multiple data centers around the globe. By utilizing the cloud infrastructure built and maintained by AWS, Netflix did not do the ***undifferentiated heavy lifting*** work of building data centers but focusing more on its core business of providing high quality video streaming user experience. Even though it has to rebuild the whole technology to allow it run smoothly on AWS cloud, the improvement of Netflix's scalability and service availability has gained significantly in return.

Netflix is also one of the first major drivers behind microservices architecture. Microservices targets the problems of monolith software design by encouraging *separation of concerns* ([11]) in which big programs are broken into smaller software components by modularity with data encapsulation on its own. Microservices also helps to increase the scalability via horizontal scaling and workload partitioning. By adopting microservices, Netflix engineers easily change any services which lead to faster deployments. More importantly, they can track the performance of each service and quickly isolate its issues from other running services.

In this study, I am interested in understanding Netflix's cloud architecture and its performance under different workloads and network limitations

system architecture learnt from various online sources. Then in section 3, more detailed system components will be discussed. In section 4, 5, 6, 7, I

will analyze the system with respect to the above design goals. Finally, I conclude what has been learnt from this analysis and possible next steps need to be taken for improvement.

2. Architecture

Netflix operates based on Amazon cloud computing services (AWS) and Open Connect, its in-house content delivery network ([1]). Both systems must work together seamlessly to deliver high quality video streaming services globally. From the software architecture point of view, Netflix comprises three main parts: Client, Backend and Content Delivery Network (CDN).

Client is any supported browsers on a laptop or desktop or a Netflix app on smartphones or smart TVs. Netflix develops its own iOS and Android apps to provide the best viewing experience for each and every client and device. By controlling their apps and other devices through its SDK, Netflix can adapt its streaming services transparently under certain circumstances such as slow networks or overloaded servers.

Backend includes services, databases, storages running entirely on AWS cloud. Backend basically *handles everything not involving streaming videos*. Some of the components of Backend with their corresponding AWS services are listed as follows:

- Scalable computing instances (AWS EC2)
- Scalable storage (AWS S3)
- Business logic microservices (purpose-built frameworks by Netflix)
- Scalable distributed databases (AWS DynamoDB, Cassandra)
- Big data processing and analytics jobs (AWS EMR, Hadoop, Spark, Flink, Kafka and other purpose-built tools by Netflix)

- Video processing and transcoding (purpose-built tools by Netflix)

Open Connect **CDN** is a network of servers called Open Connect Appliances (OCAs) optimized for storing and streaming large videos. These OCAs servers are placed inside internet service providers (ISPs) and internet exchange locations (IXPs) networks around the world. *OCAs are responsible for streaming video*s directly to clients.

In the following sections, I will describe a reference of Netflix cloud architecture comprising these above 3 parts. In section 2.1, an overall architecture is capable of streaming videos, called **playback architecture**, after a subscriber clicks the Play button on his or her apps. Then in section 2.2, a more detailed **microservices architecture** of Backend will be described to demonstrate how Netflix handles availability and scalability at global scale.

2.1 Playback Architecture

When subscribers click the Play button on their apps or devices, the Client will talk to both Backend on AWS and OCAs on Netflix CDN to stream videos ([7]). The following diagram illustrates how the playback process works:

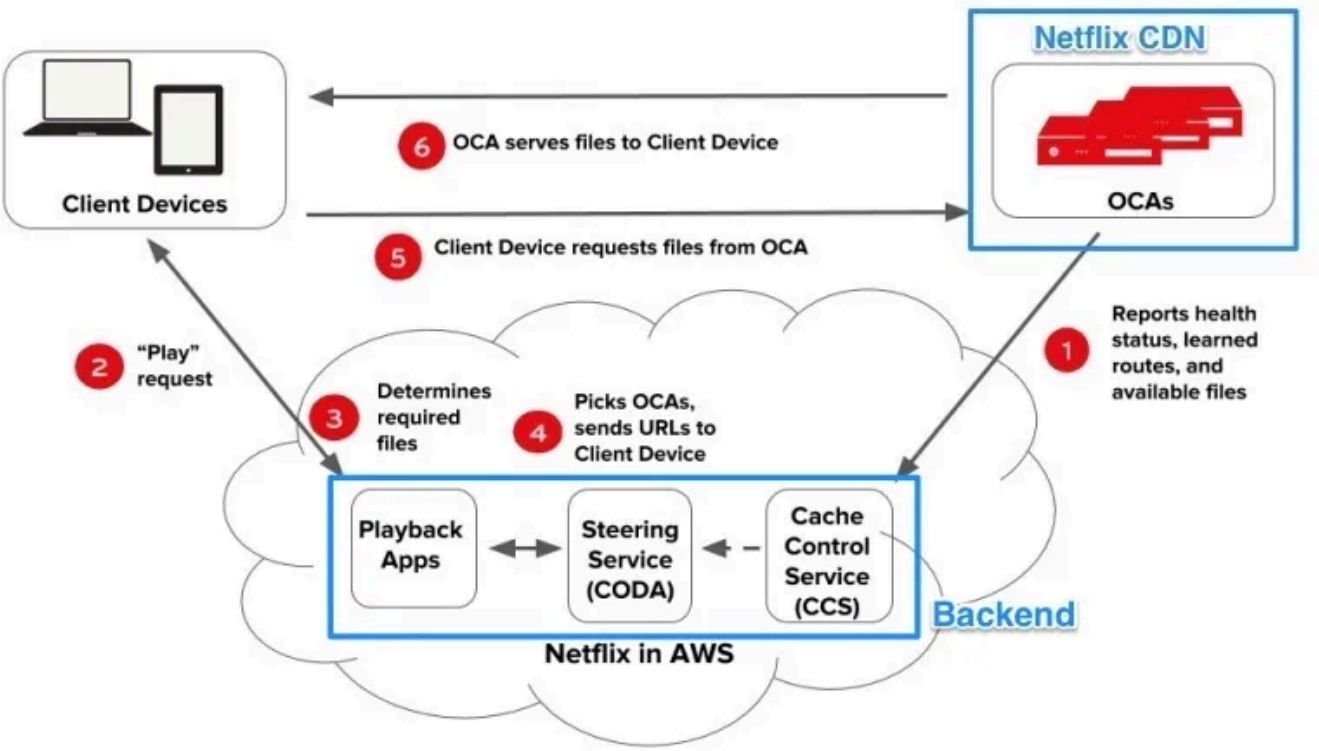

Fig 1. Playback architecture for streaming videos

1. OCAs constantly send health reports about their workload status, routability and available videos to Cache Control service running in AWS EC2 in order for Playback Apps to update the latest healthy OCAs to clients.
2. A Play request is sent from the client device to Netflix's Playback Apps service running on AWS EC2 to get URLs for streaming videos.
3. Playback Apps service must determine that Play request would be valid in order to view the particular video. Such validations would check subscriber's plan, licensing of the video in different countries, etc.
4. Playback Apps service talks to Steering service also running in AWS EC2 to get the list of appropriate OCAs servers of the requested video. Steering service uses the client's IP address and ISPs information to identify a set of suitable OCAs work best for that client.
5. From the list of 10 different OCAs servers returned by Playback Apps service, the client tests the quality of network connections to these OCAs and selects the fastest, most reliable OCA to request video files for streaming.
6. The selected OCA server accepts requests from the client and starts streaming videos.

In the above diagram, Playback Apps service, Steering service and Cache Control service run entirely in AWS cloud based on a microservices architecture. In the next section, I will describe a reference of Netflix Backend microservices architecture which increases the availability and scalability of running services.

2.2 Backend Architecture

As I have described in previous sections, Backend handles almost everything, ranging from sign up, login, billing to more complex processing tasks such as video transcoding and personalized recommendations. In

order to support both lightweight and heavy workloads running on the same underlying infrastructure, Netflix has chosen microservices architecture for their cloud based system. The diagram in Figure 2 represents a possible microservices architecture at Netflix which I have derived from several online sources ([11, 13, 14]):

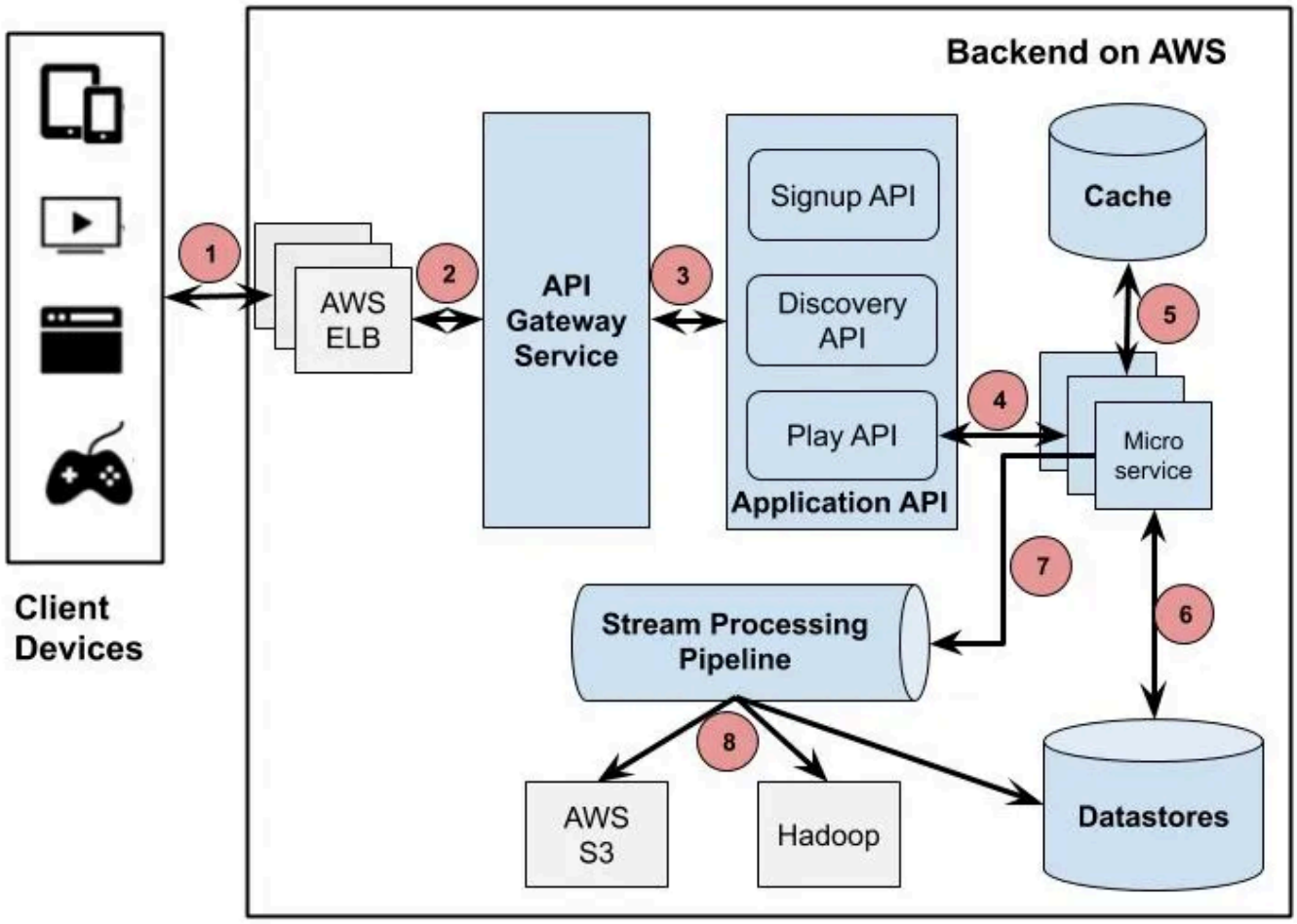

Fig 2. A reference of Backend architecture based on various sources

1. The Client sends a Play request to Backend running on AWS. That request is handled by AWS Load balancer (ELB)
2. AWS ELB will forward that request to API Gateway Service running on AWS EC2 instances. That component, named **Zuul**, is built by Netflix team to allow dynamic routing, traffic monitoring and security, resilience to failures at the edge of the cloud deployment. The request will be applied to some predefined filters corresponding to business logics, then is forwarded to Application API for further handling.

3. Application API component is the core business logic behind Netflix operations. There are several types of API corresponding to different user activities such as Signup API, Recommendation API for retrieving video recommendation. In this scenario, the forwarded request from API Gateway Service is handled by Play API.
4. Play API will call a microservice or a sequence of microservices to fulfill the request. Playback Apps service, Steering service and Cache Control service in Figure 1 can be seen as a microservice in this diagram.
5. Microservices are mostly stateless small programs and can call each other as well. To control its *cascading failure* and *enable resilience,* each microservice is isolated from the caller processes by **Hystrix**. Its result after run can be cached in a memory-based cache to allow faster access for those critical low latency requests.
6. Microservices can save to or get data from a data store during its process.
7. Microservices can send events for tracking user activities or other data to the Stream Processing Pipeline for either real-time processing of personalized recommendation or batch processing of business intelligence tasks.
8. The data coming out of the Stream Processing Pipeline can be persistent to other data stores such as AWS S3, Hadoop HDFS, Cassandra, etc.

The described architectures help us get a general understanding of how different pieces organize and work together to stream videos. However, to analyze the availability and scalability of the architectures, we need to go more into each important component to see how it performs under different workloads. That will be covered in the next section.

3. Components

In this section, I want to look into the components defined in Section 2 in order to analyze its availability and scalability. When describing each

component, I would also provide how it meets these design goals. A more in depth design analysis will be mentioned in subsequent sections with respect to the whole system.

3.1 Client

Netflix technical teams have put a lot of effort into developing faster and smarter client applications running on either laptops, desktops or mobile devices. Even on some smart TVs in which Netflix does not build a specialized client, Netflix still controls its performance via the provided SDK. In fact, any device environment needs to install Netflix Ready Device Platform (NRDP) in order to enable the best possible Netflix viewing experience. A typical client structural component ([11]) is illustrated in Figure 3.

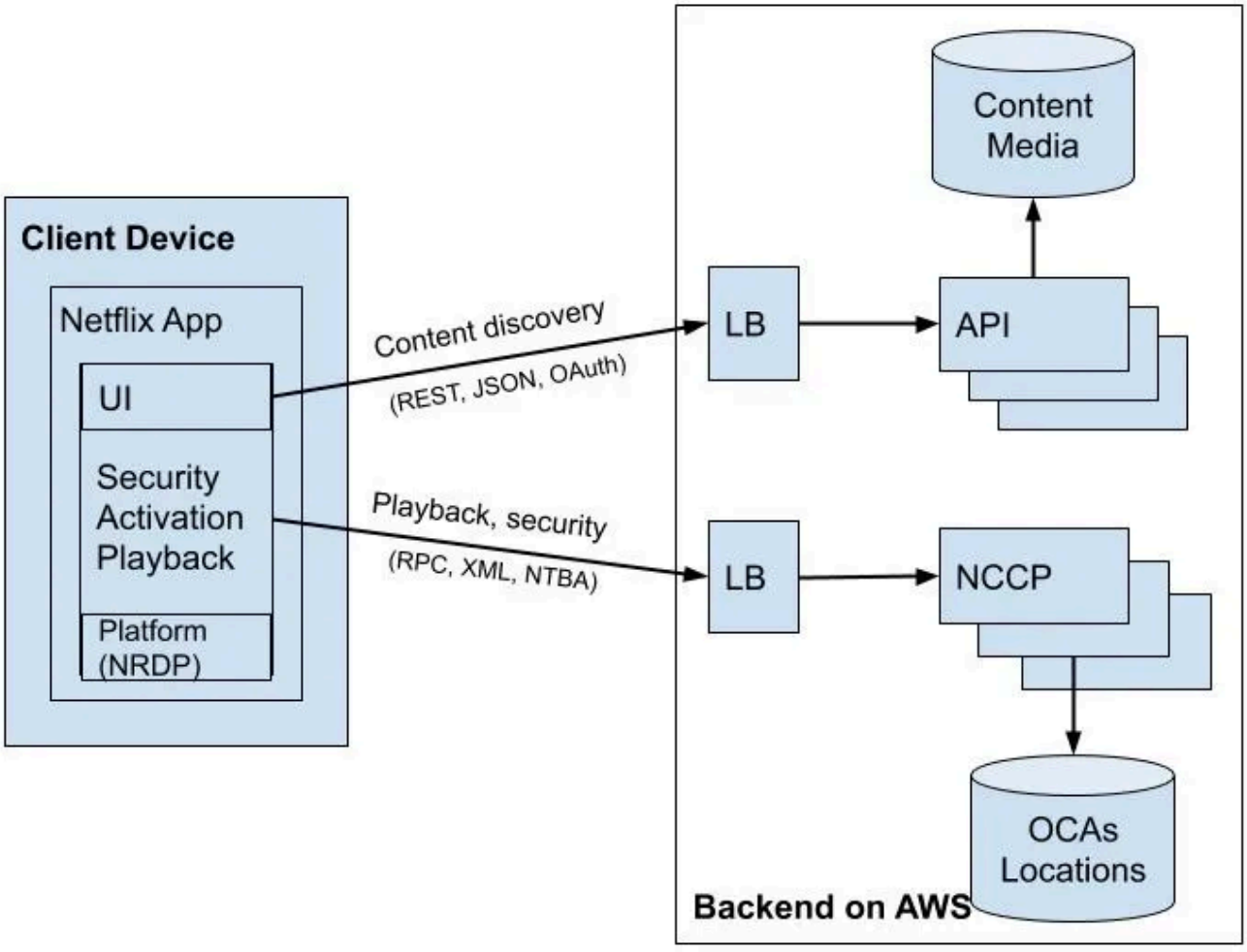

Fig 3. Client App Component

- Client Apps separate 2 types of connections to Backend for content discovery and playback. Client uses NTBA protocol ([15]) for Playback requests to ensure more security over its OCA servers locations and to remove the latency caused by a SSL/TLS handshake for new connections.
- While streaming videos, Client App intelligently lowers the video quality or switches to different OCA servers ([1]) if network connections are overloaded or have errors. Even if the connected OCA is overloaded or failed, Client App can easily change to another OCA server for better viewing experience. All this could be achieved because the Netflix Platform SDK on Client keeps tracking the latest healthy OCAs retrieved from Playback Apps service (Figure 1)

3.2 Backend

3.2.1 API Gateway Service

API Gateway Service component communicates with AWS Load Balancers to resolve all requests from clients. This component can be deployed to multiple AWS EC2 instances across different regions to increase Netflix service availability. The diagram in Figure 4 represents an open-sourced **Zuul**, an implementation of API Gateway created by Netflix team.

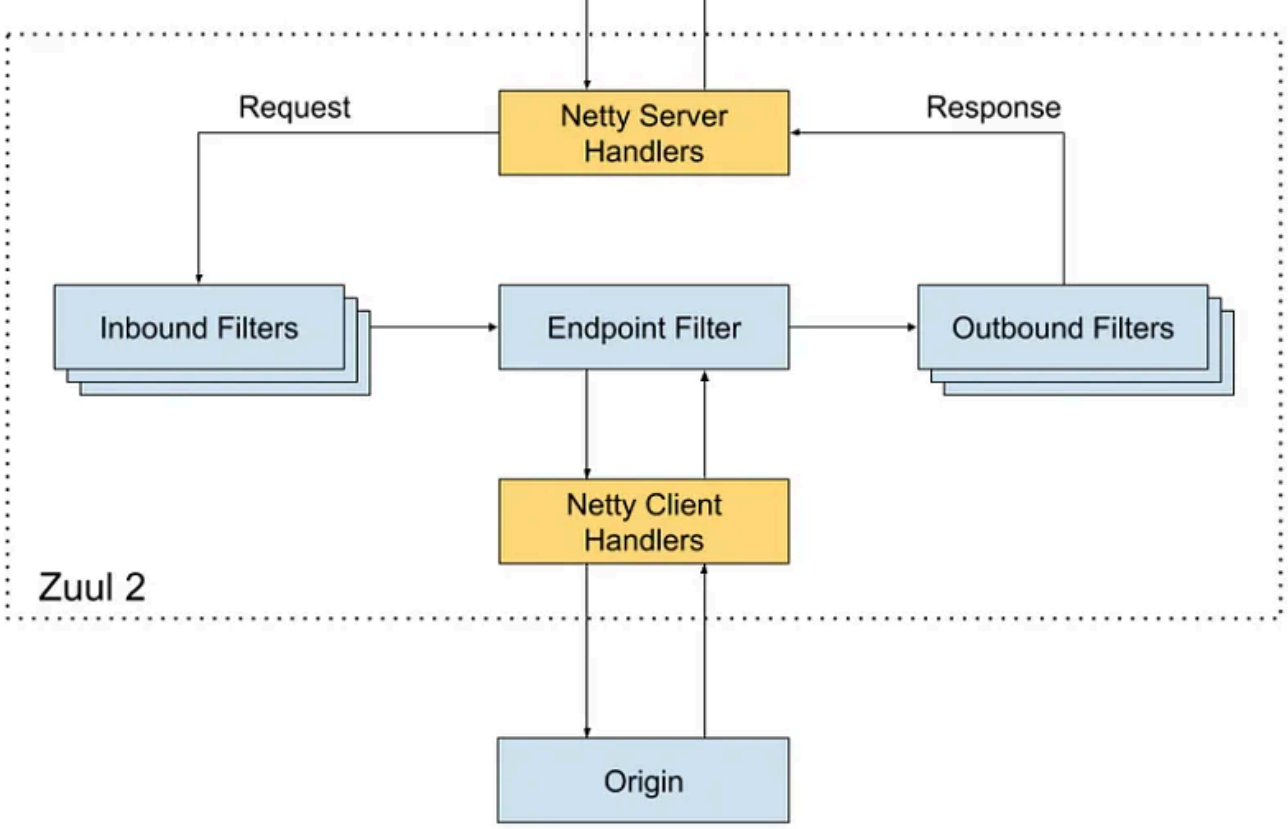

Fig 4. Zuul Gateway Service Component

- Inbound Filters can be used for authentication, routing and decorating the request.
- Endpoint Filter can be used to return static resources or route the request to appropriate Origin or Application API for further processing.
- Outbound Filters can be used for tracking metrics, decorating the response to the user or adding custom headers.
- Zuul is able to discover new Application API by integrating with the Service discovery **Eureka**
- Zuul is used extensively for routing traffic for different purposes such as onboarding new application API, load tests, routing to different service endpoints under huge workloads.

3.2.2 Application API

Application API plays a role of an orchestration layer ([18]) to the Netflix microservices. The API provides a logic of composing calls to underlying microservices in the order needed, with the additional data from other data stores to construct appropriate responses. Netflix team has spent a lot of

time designing the Application API component since it is correspondent to Netflix core business functionalities. It also needs to be scalable, highly available under high request volume. Currently, the Application APIs are defined under three categories: ***Signup API*** for non-member requests such as sign-up, billing, free trial, etc., ***Discovery API*** for search, recommendation requests and ***Play API*** for streaming, view licensing requests. A detailed structural component diagram of Application API is provided in Figure 5.

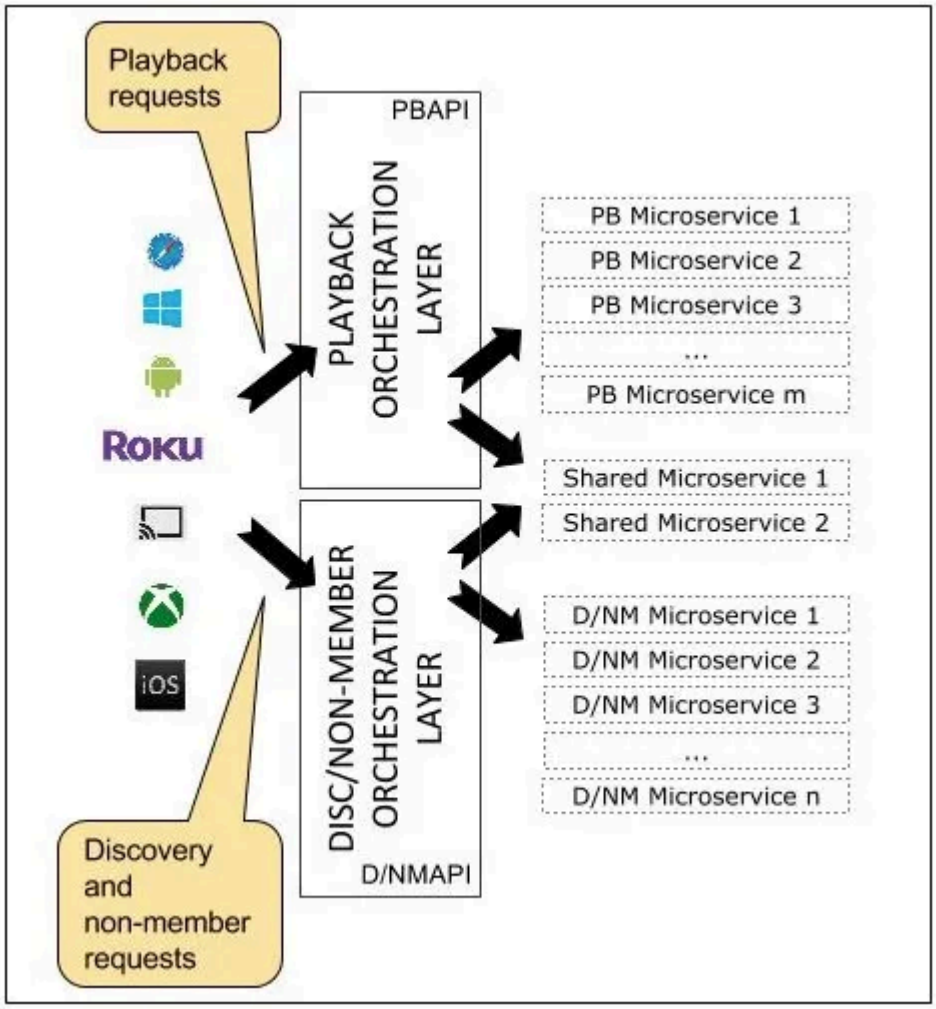

Fig 5. Separation of Play and Discovery Application API

- In a recent update of Play API implementation, the network protocol between Play API and microservices is gRPC/HTTP2 which "*allowed RPC methods and entities to be defined via Protocol Buffers, and client libraries/SDKs automatically generated in a variety of languages*" ([13]). The change allows Application API to integrate appropriately with auto-generated clients via bi-directional communication and to "minimize code reuse across service boundaries".

- Application API also provides a common resilient mechanism based on Hystrix commands to protect its underlying microservices.

Since Application API has to deal with huge volumes of requests and construct appropriate responses, its internal processing needs to run highly in parallel. Netflix team has found a combination of synchronous execution and asynchronous I/O ([13]) is the right approach to go.

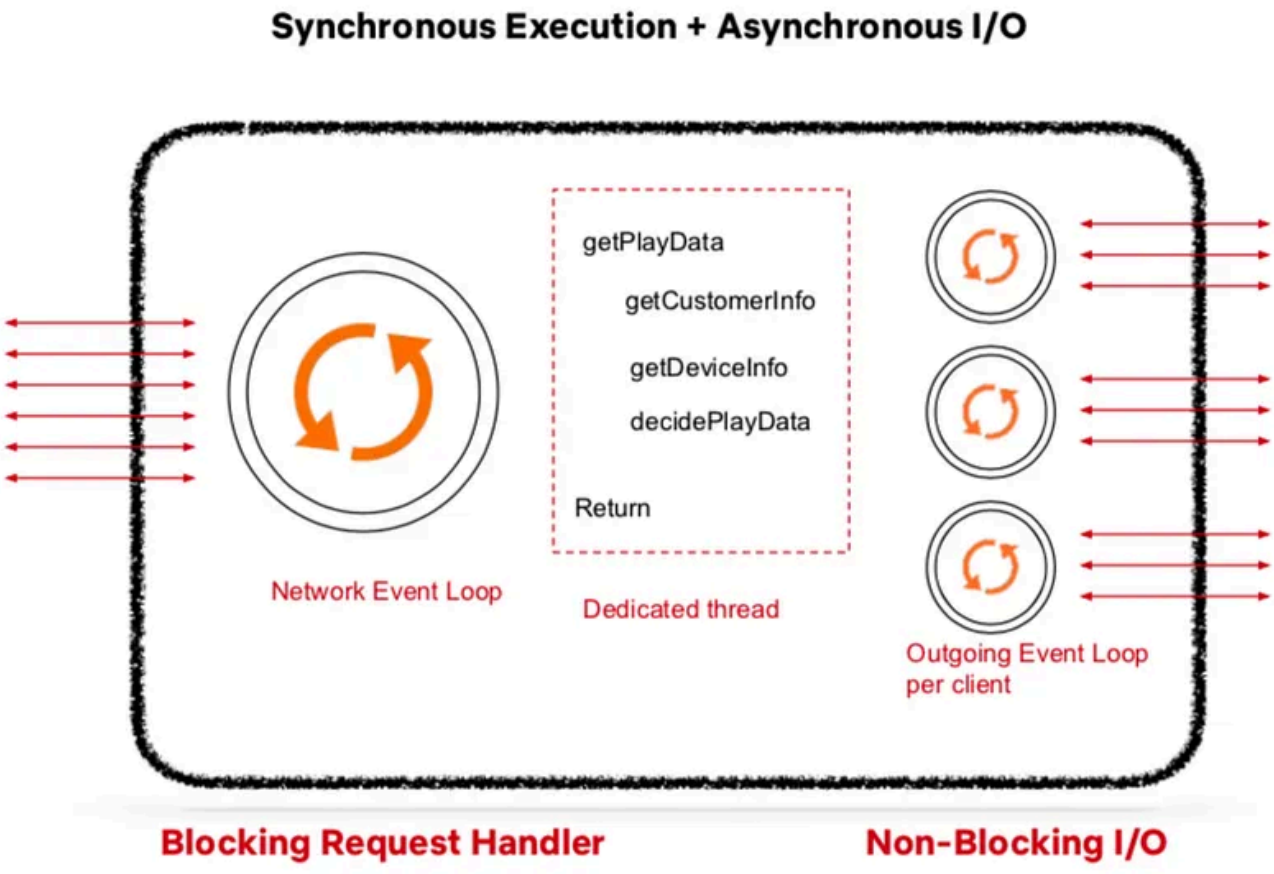

Fig 6. Synchronous Execution & Asynchronous I/O of Application API

- Each request from API Gateway Service will be placed into Application API's Network Event Loop for processing
- Each request will be blocked by a dedicated thread handler which places Hystrix commands such as getCustomerInfo, getDeviceInfo, etc. into the Outgoing Event Loop. This Outgoing Event Loop is set up per client and runs with non-blocking I/O. Once the calling microservices finish or timeout, the dedicated thread would construct corresponding responses.

3.2.3 Microservices

> *From Martin Fowler's definition, "microservices are a suite of small services, each running in its own process and communicate with lightweight mechanisms…". These small programs are independently deployable or upgradable with respect to others and have their own encapsulated data.*

An implementation of the microservice component at Netflix ([11]) is illustrated in Figure 7.

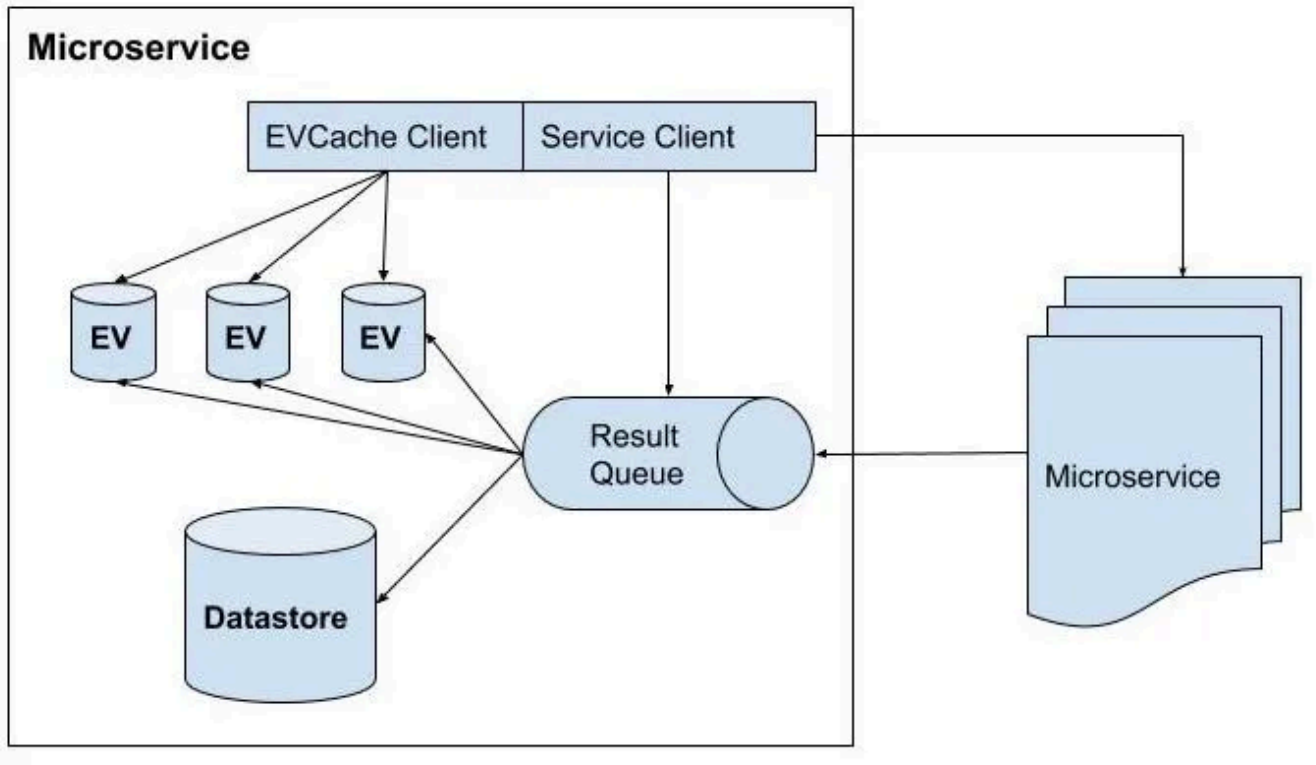

Fig 7. Structural component of a microservice

- A microservice can work on its own or call other microservices via REST or gRPC.
- The implementation of microservice can be similar to that of Application API as described in Figure 6 in which the requests would be put into the Network Event Loop and results from other called microservices are placed into the result queue in asynchronous non-blocking I/O.
- Each microservice can have its own datastore and some in-memory cache stores of recent results. **EVCache** is a primary choice for caching of microservices at Netflix.

3.2.4 Data Stores

When migrating their infrastructure to AWS cloud, Netflix made use of different data stores (Figure 8), both SQL and NoSQL, for different purposes ([6]).

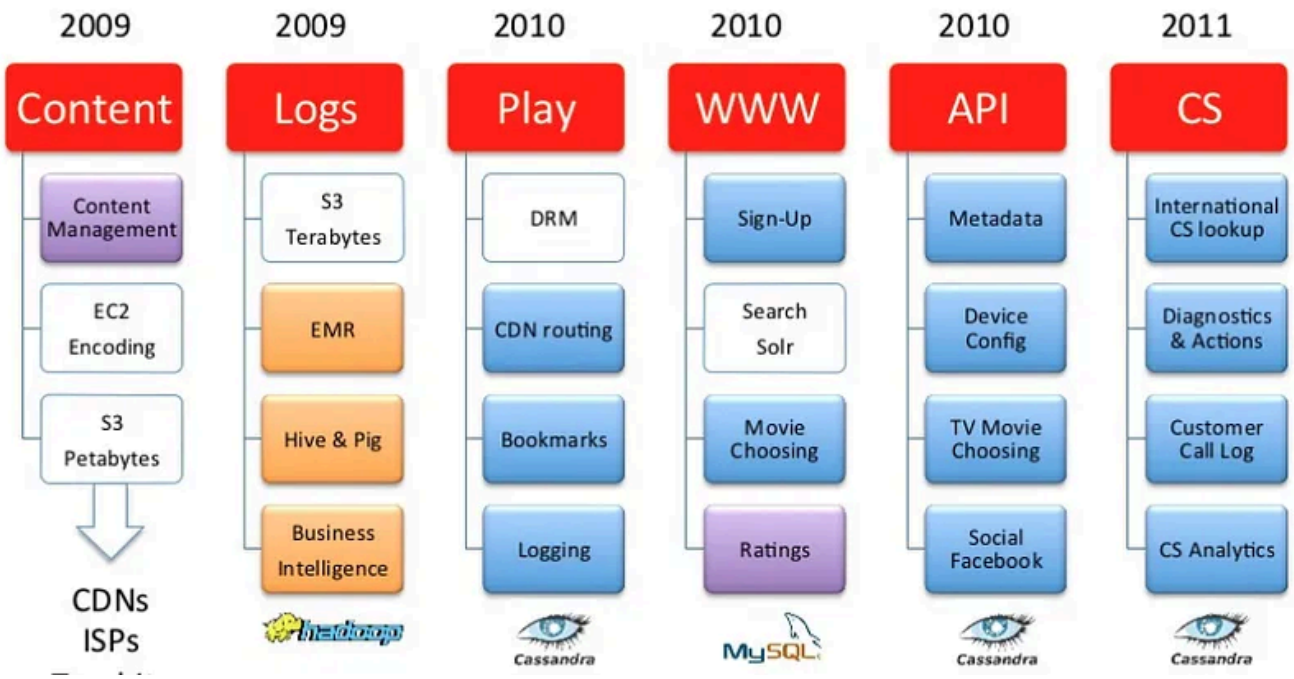

Fig 8. Netflix Data Stores deployed on AWS

- MySQL databases are used for movie title management and transactional/billing purposes.
- Hadoop is used for big data processing based on user logs
- ElasticSearch has powered searching titles for Netflix apps
- Cassandra is a distributed column-based NoSQL data store to handle large amounts of read requests with no single point of failure. To optimize the latency over large write requests, Cassandra is used because of its eventually consistent ability.

3.2.5 Stream Processing Pipeline

Stream Processing Data Pipeline ([14, 3]) has become Netflix's data backbone of business analytics and personalized recommendation tasks. It is responsible for producing, collecting, processing, aggregating, and moving all microservice events to other data processors in near real-time. Figure 9 shows various pieces of the platform.

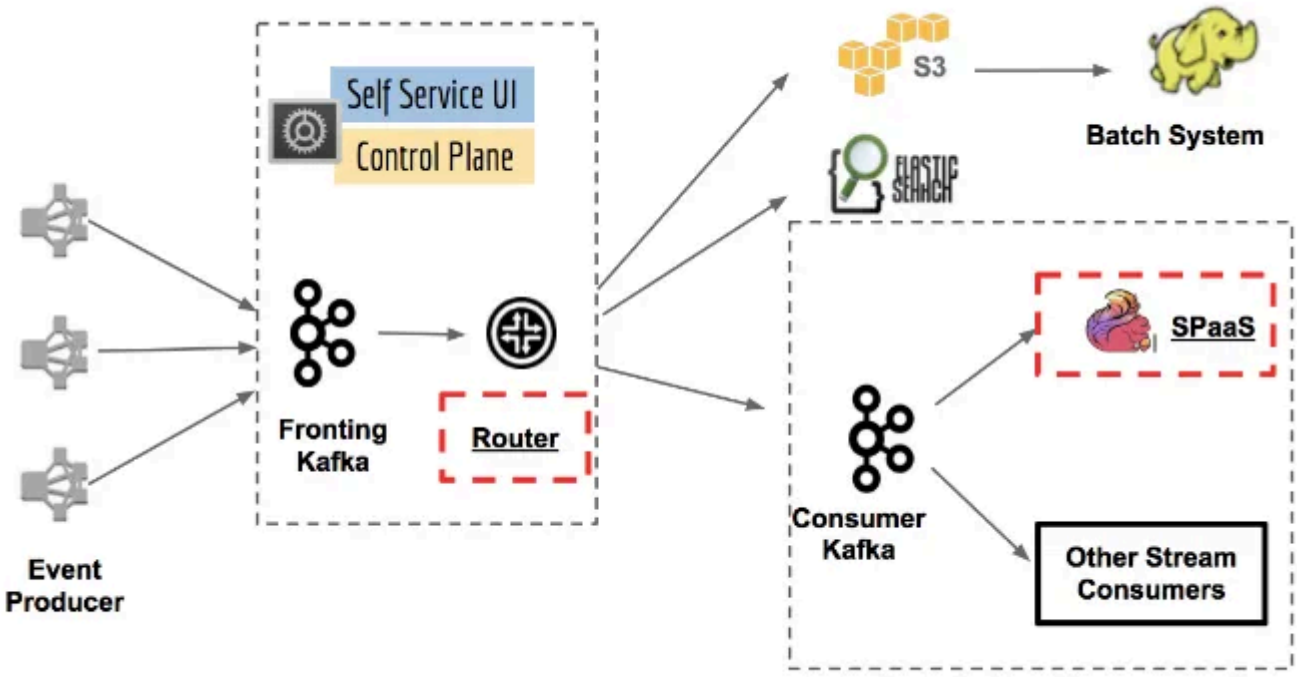

Fig 9. Keystone Stream Processing Platform at Netflix

- The stream processing platform has processed trillions of events and petabytes of data per day. It will also automatically scale as the number of subscribers increases.
- The **Router** module enables routing to different data sinks or applications while **Kafka** is responsible for routing messages as well as buffering for downstream systems.
- Stream Processing as a Service (**SPaaS**) allows data engineers to build and monitor their custom managed stream processing applications while the platform would take care of the scalability and operations.

3.3 Open Connect

Open Connect is a global content delivery network (CDN) responsible for storing and delivering Netflix TV shows and movies to their subscribers world-wide. Netflix had built and operated Open Connect efficiently by bringing the content that people want to watch as close as possible to where they want to watch it. In order to localize traffic of watching Netflix videos to the customers' network, Netflix has been in partnership with Internet Service Providers (ISPs) and Internet Exchange Points (IXs or IXPs) around

the world to deploy specialized devices called Open Connect Appliances (OCAs) inside their network ([7]).

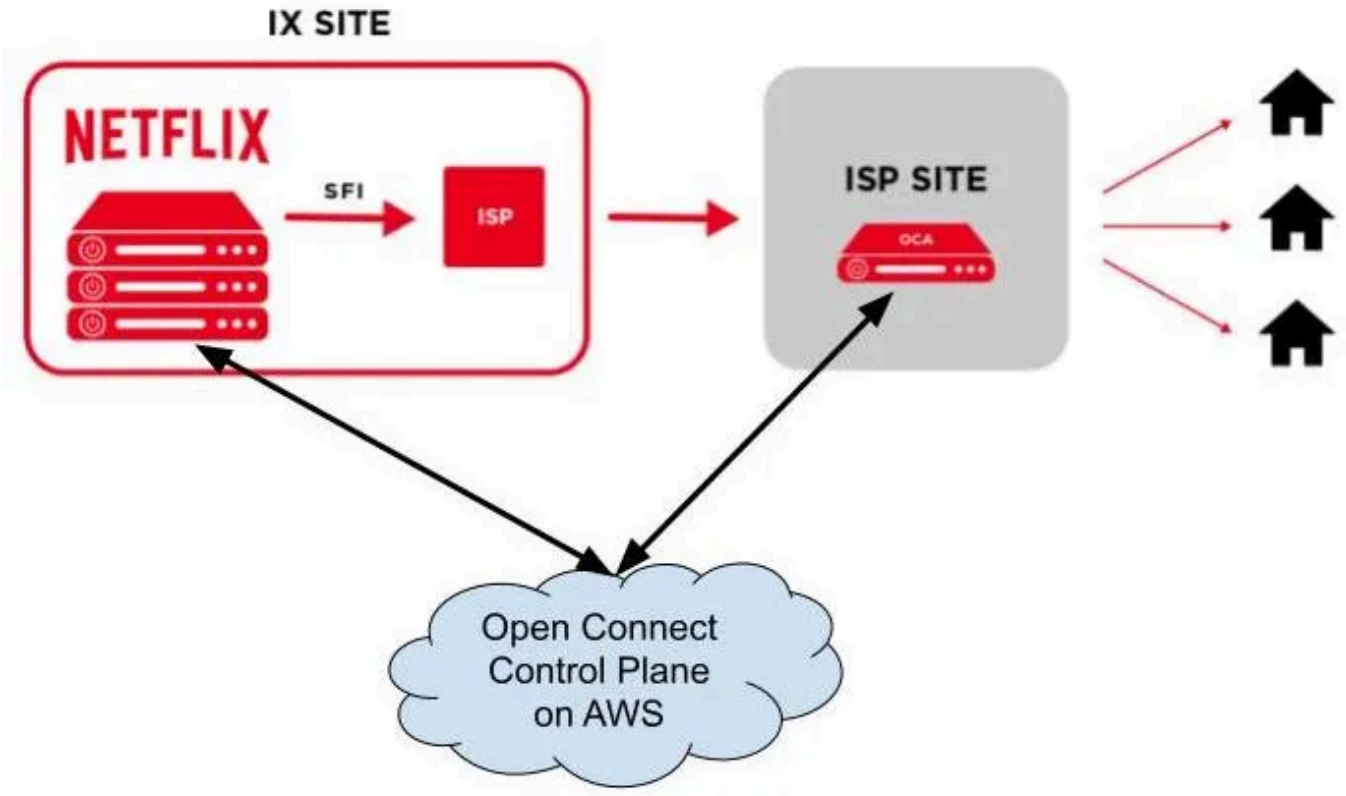

Fig 10. Deployment of OCAs to IXs or ISPs sites

OCAs are servers optimized for storing and streaming large video files from IXs or ISPs sites directly to subscribers' homes. These servers periodically report health metrics optimal routes they learned from IXP/ISP networks and what videos they store on their SSD disks to Open Connect Control Plane services on AWS. In return, the control plane services would take such data to direct client devices automatically to the most optimal OCAs given the file availability, server health and network proximity to the clients.

The control plane services also control filling behaviour of adding new files or updating files on OCAs nightly. The filling behaviours ([8,9]) are illustrated in Figure 11.

- When new video files have been transcoded successfully and stored on AWS S3, the control plane services on AWS will transfer these files to OCAs servers on IXP sites. These OCAs servers will apply **cache fill** to

transfer these files to OCAs servers on ISPs sites under their sub networks.

- When an OCA server has successfully stored the video files, it will be able to start the **peer fill** to copy these files to other OCAs servers within the same site if needed.
- Between 2 different sites which can see each other IP addresses, the OCAs can apply the **tier fill** process instead of a regular cache fill.

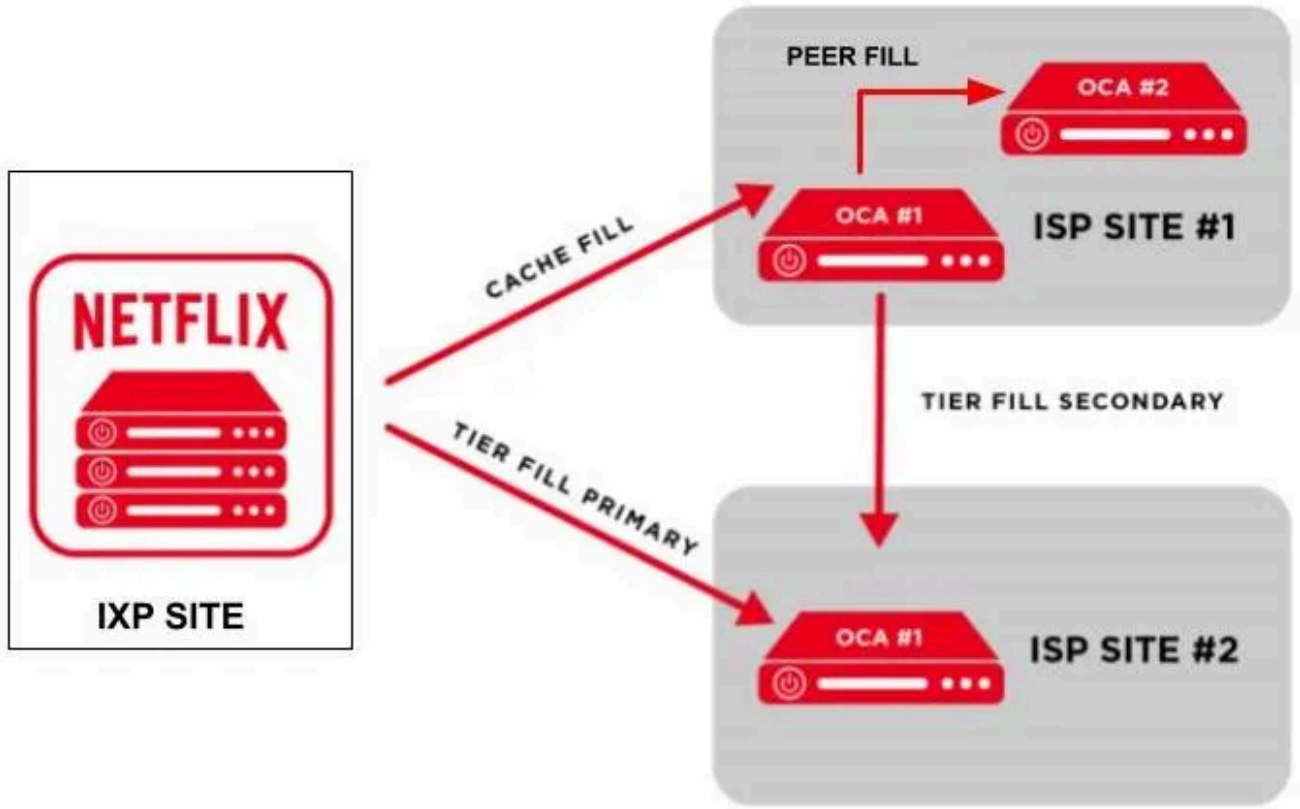

Fig 11. Fill patterns among OCAs

4. Design Goals

In previous sections, I have described in detail the cloud architecture and its components powering Netflix's video streaming business. In this section and the subsequent sections, I would like to go deeper into analyzing this design architecture. I start with the list of most important design goals as follows:

- Ensure **high availability** for streaming services at global scale.
- Tackle network failures and system outages by **resilience**.
- Minimize streaming **latency** for every supported device under different network conditions.
- Support **scalability** upon high request volume.

In the subsections, I am going to analyze the availability of the streaming service and its corresponding optimal latency. Section 6 looks at more in depth analysis about resilience mechanisms such as **Chaos Engineering** while Section 7 covers scalability of the streaming services.

4.1 High Availability

By definition, availability of a system is measured in terms of how many times a response would be fulfilled for a request within a period of time, *without guarantee that it contains the most recent version of the information*. In our system design, the availability of streaming services depends on both the availability of Backend services and OCAs servers keeping the streaming video files.

The goal of Backend services is to get the list of most healthy OCAs proximity to a specific client, either from cache or by execution of some microservices. Therefore, its availability depends on different components involving the Playback request: load balancers (AWS ELB), proxy servers (API Gateway Service), Play API, execution of microservices, cache stores (EVCache) and data stores (Cassandra):

> *Load balancers can improve the availability by routing traffic to different proxy servers to help prevent overloading workloads.*
>
> *Play API controls the execution of microservices with timeout via Hystrix commands which could help to prevent cascading failures to further services.*

> *Microservices can respond to Play AI with data in cache in case the call to outside services or data stores takes more time than expected.*
>
> *Cache is replicated for faster access.*

When receiving the list of OCAs servers from Backend, the client probes the network to these OCAs and chooses the best OCAs to connect to. If that OCA is overloaded or failed during the streaming process, then the client switches to another good one or the Platform SDK would request other OCAs. Therefore its availability is highly correlated with the availability of all OCAs available in its ISPs or IXPs.

The high availability of Netflix streaming services comes at the cost of complex multi-region AWS operations and services as well as the redundancy of OCAs servers.

4.2 Low Latency

The latency of streaming services depends mostly on *how fast Play API can resolve the list of healthy OCAs and how well the connection of a client to the chosen OCA server*.

As I have described in the Application API component section, *Play API does not wait for a microservice's execution forever since it uses Hystrix commands to control how long it would like to wait for the result before it gets the not-up-to date data from the cache*. Doing so could control the acceptable latency as well as stop the cascading failures to further services.

The client would immediately switch to other nearby OCAs servers with most reliable network connection if there is a network failure to the current selected OCA server or that server is overloaded. It can also lower the video quality to match with the network quality in case it finds out a degradation in network connection.

5. Tradeoffs

In the above described system design, there are two prominent trade-offs have been carefully implemented:

- **Low latency over consistency**
- **High availability over consistency**

Latency over Consistency trade-off is built into the architecture design of Backend services. Play API can get stale data from EVCache stores or from eventually consistent data stores like Cassandra.

Similarly, Availability over Consistency trade-off would prefer constructing responses in acceptable latency without requiring the execution of microservices on latest data in data stores like Cassandra.

There is also a not-quite-relevant trade-off between Scalability and Performance ([21]). In this trade-off, *improving scalability by increasing the number instances to process more workloads may cause the system running under its expected increasing performance*. This could be a problem with those design architectures in which the workloads are not load ***well balanced*** among available workers. However, Netflix has resolved this trade-off with AWS auto scaling. We come back to this resolve in more detail in Section 7.

6. Resilience

Designing a cloud system capable of self-recover from failures or outages has been the long goal at Netflix from the start day of migration to AWS cloud. Some common failures that the system have been addressed as follows:

> *A failure in resolving service dependencies.*
>
> *A failure of executing a microservice would cause cascading failures to other services.*

> *A failure of connecting to an API due to overloading.*
>
> *A failure of connecting to an instances or servers such as OCAs.*

To detect and resolve these failures, the API Gateway Service Zuul ([20]) has built-in features such as adaptive retries, limiting concurrent calls to Application API. In return, *the Application API uses Hystrix commands to time-out calls to microservices, to stop cascading failures and isolate points of failures from others.*

Netflix technical teams are also famous for their chaos engineering practices. The idea is to inject pseudo-randomly errors into production environments and build solutions to automatically detect, isolate and recover from such failures. The errors can be adding delays to responses of executing microservices, killing services, stopping servers or instances, and even bringing down the whole infrastructure of a region([5]). *By purposefully introducing realistic production failures into a monitored environment with tools to detect and resolve such failures, Netflix can uncover such weaknesses quickly before they cause bigger problems.*

7. Scalability

In this section, I will analyze the scalability of Netflix streaming services by covering *horizontal scaling, parallel execution and database partitioning*. The other parts such as caching and load balancing also help increasing scalability have been mentioned in Section 4.

First, the horizontal scaling of EC2 instances at Netflix is provided by AWS Auto Scaling Service. This AWS service automatically spins up more elastic instances if the request volume increases and turns off unused ones. More specifically, on top of thousands of these instances, Netflix has built **Titus** ([17]), an open source ***container management platform***, to run about 3 million containers per week. Also, any component of our architecture *Figure 2* can be deployed inside a container. Moreover, Titus allows containers to run on multi-regions across different continents around the world.

Second, the implementation of an Application API or a microservice in *Section 3.2.2* also increases the scalability by allowing parallel execution of tasks on the Network Event Loop and the asynchronous Outgoing Event Loop.

Lastly, the wide column stores such as Cassandra and key-value object stores like ElasticSearch also offer *high availability and high scalability with no single point of failure*.

8. Conclusion

The study has described the whole cloud architecture of streaming services at Netflix. It also analyzed different design goals in terms of availability, latency, scalability and resilience to network failures or system outages. In short, Netflix's cloud architecture, proven by their production system to serve millions of subscribers running on thousands of virtual servers, has demonstrated a ***high availability with optimal latency, strong scalability through integration with AWS cloud services and resilience capability to network failures and system outages at global scale.*** Most of the derived architecture and components are learnt through available online trusted resources. Even though there are not many direct resources describing the internal implementation of microservices as well as the tools and systems to monitor their performance, **this study can serve as a reference implementation of how a typical production system should be built.**

References

1. **Netflix: What Happens When You Press Play**? By Todd Hoff on Dec 11, 2017. Link
2. **High Quality Video Encoding at Scale.** By Anne Aaron and David Ronca on HighScalability. Dec 9, 2015. Link
3. **Building and Scaling Data Lineage at Netflix to Improve Data Infrastructure Reliability, and Efficiency.** By Di Lin, Girish Lingappa,

Jitender Aswani on The Netflix Tech Blog. Mar 25, 2019. Link

4. **Ten years on: How Netflix completed a historic cloud migration with AWS.** By Tom Macaulay on Computerworld. Sep 10, 2018. Link
5. **The Netflix Simian Army.** By Yury Izrailevsky and Ariel Tseitlin on The Netflix Tech Blog. Link
6. **Globally Cloud Distributed Applications at Netflix.** By Adrian Cockcroft. Oct 2012. Link
7. **Open Connect Overview.** By Netflix. Link
8. **Open Connect Deployment Guide.** By Netflix. Link
9. **Netflix and Fill.** By Michael Costello and Ellen Livengood. Aug 11, 2016. Link
10. **Automating Operations of a Global CDN.** By Robert Fernandes at Strange Loop. Sep 14, 2019. Link
11. **Mastering Chaos — A Netflix Guide to Microservices.** By Josh Evans at QCon. Dec 07, 2016. Link
12. **Netflix Revenue and Usage Statistics.** By Mansoor Iqbal on BusinessofApps. March 6, 2020. Link
13. **Netflix Play API — Why we build an Evolutionary Architecture.** By Suudhan Rangarajan at QCon 2018. Dec 12, 2018. Link
14. **Keystone Real-time Stream Processing Platform.** By Zhenzhong Xu on The Netflix Tech Blog. Sep 10, 2018. Link
15. **Netflix Releases Open Source Message Security Layer.** By Chris Swan on InfoQ. Nov 24th, 2014. Link
16. **Netflix Open Source.** Link
17. **Titus, the Netflix container management platform, is now open source.** By Amit Joshi and others. Link
18. **Engineering Trade-Offs and The Netflix API Re-Architecture.** By Katharina Probst and Justin Becker on The Netflix Tech Blog. Aug 23,