Partie 6 : lien vers la présentation

https://drive.google.com/file/d/1VF2yaQw3qQThtWXc5tfOqTayu7igQ76-/view?usp=sharing

Partie 6 : Architectures Hors Normes : technologies, patterns et étude de cas

- Très haute disponibilité
- Très forte montée en charge
- Cas de la base de données
- Quelques patterns et principes
- Etude de cas : Netflix

Très haute disponibilité

- Définition de la disponibilité : temps pendant lequel le système est complètement opérationnel

- Calcul de la disponibilité :
  disponibilité =
  (nb minutes de fonctionnement sur la période) * 100
  / (nb minutes sur la période)

- on parle de HA : High Availability

Très haute disponibilité

- La méthode des 5 - 9 : 5 niveaux de disponibilité

Niveau de disponibilité	Durée maximale d'arrêt du système sur un an	Durée moyenne d'arrêt du système par jour
1 - 9 : 90 %	36,5 jours	2,4 heures
2 - 9 : 99 %	3,65 jours	14 minutes
3 - 9 : 99,9 %	8,76 heures	86 secondes
4 - 9 : 99,99 %	52,6 minutes	8,6 secondes
5 - 9 : 99,999 %	5,25 minutes	0,86 seconde

Très haute disponibilité

- coût indisponibilité : peu de communication sur le sujet

- un exemple :
  - Amazon, août 2013 : interruption totale du site et des applications mobiles pendant 15 minutes.
  - Coût estimé de l'interruption : 66 000 $ / minute soit environ 1 M$

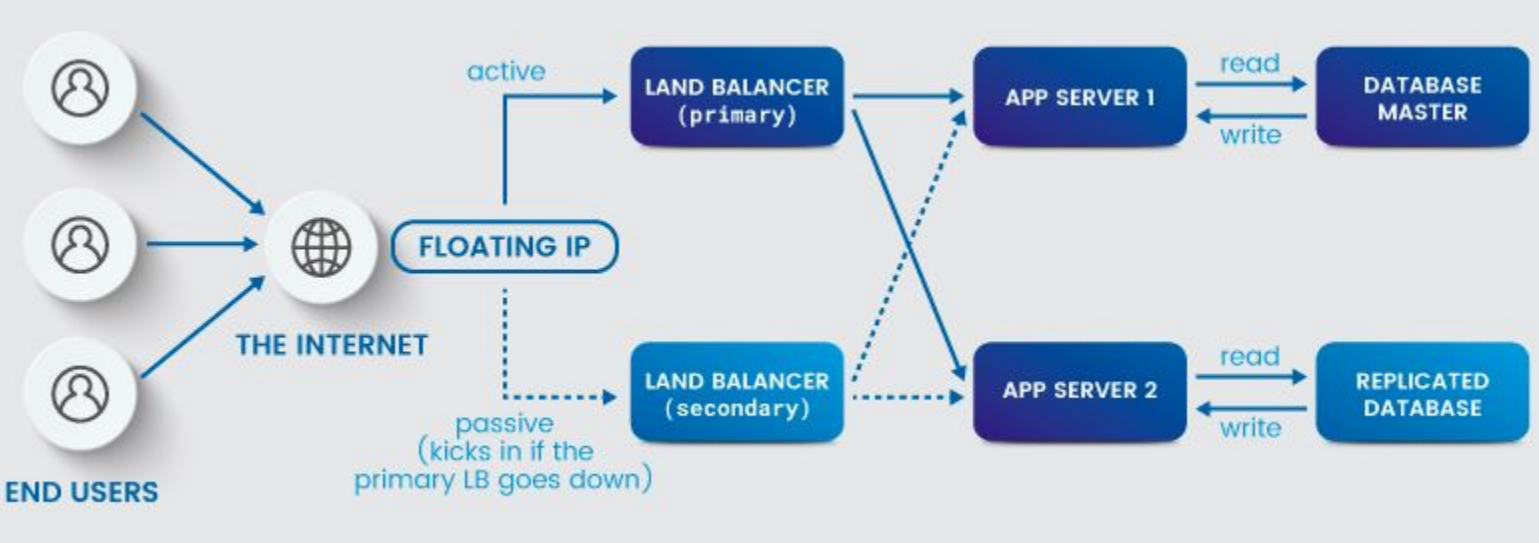
- une étude donne les chiffres suivants (ce sont des moyennes) :
  - 5 600 $ par minute d'interruption pour les grandes entreprises
  - 1 000 $ par minute d'interruption pour les entreprises de tailles moyennes
  - 450 $ par minute d'interruption pour les petites entreprises

Très haute disponibilité : les moyens à mettre en oeuvre

- Elimination des points de défaillance unique : les SPOF (Single Point Of Failure)
- Sauvegarde, récupération et réplication des données
- Clustering
- Network Load Balancing (équilibrage de charge réseau)
- Fail Over (Basculement en cas de panne)
- Redondance géographique
- Plan de reprise sur incident

Très haute disponibilité : les moyens à mettre en oeuvre

Exemple (simplifié)



Très haute disponibilité : les moyens à mettre en oeuvre

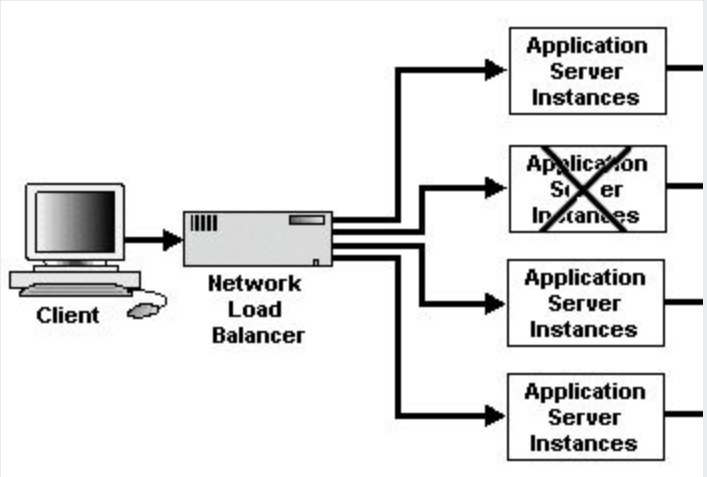
- Elimination des points de défaillance unique : les SPOF (Single Point Of Failure) :

  La disponibilité globale d'un système dépend de l'élément "le plus faible" du système

- Sauvegarde, récupération et réplication des données
  - stratégie de sauvegarde et de récupération
  - réplication pour démarrage d'un système secondaire en cas de défaillance du système primaire

Très haute disponibilité : les moyens à mettre en oeuvre

- Clustering



- Network Load Balancing (équilibrage de charge réseau)

Très haute disponibilité : les moyens à mettre en oeuvre

- Fail Over (Basculement en cas de panne)
  - Un système secondaire prêt à démarrer dans le cas où le système primaire est hors ligne, que ce soit en raison d'une panne ou d'un arrêt planifié.
  - "basculement à froid” : le serveur secondaire n'est démarré qu'après l'arrêt complet du serveur primaire.
  - “basculement à chaud" : tous les serveurs fonctionnent simultanément et que la charge est entièrement dirigée vers un seul serveur à un moment donné.

- Redondance géographique
  - Répartir les composants (load balancer, serveurs, bases de données) sur plusieurs sites distants complètement indépendants.

Très haute disponibilité : les moyens à mettre en oeuvre

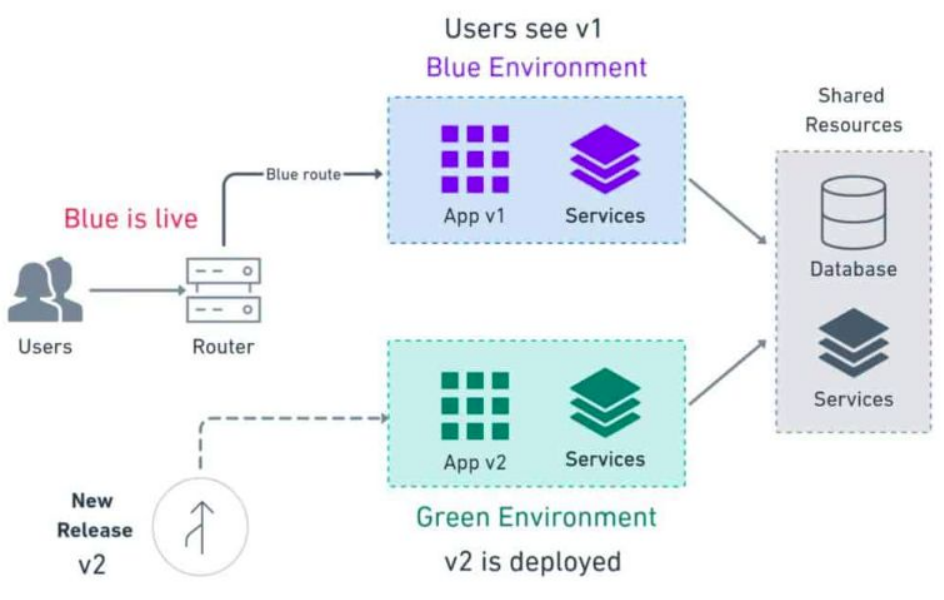
- Plan de reprise sur incident ou plan de reprise d'activité (PRA)

  Le plan de reprise d'activité (PRA) d'une entreprise constitue l'ensemble des « procédures documentées lui permettant de rétablir et de reprendre ses activités en s'appuyant sur des mesures temporaires adoptées pour répondre aux exigences métier habituelles après un incident ».

  Le plan de reprise d'activité comprend les tâches suivantes :
  - Identification des activités critiques.
  - Identification des ressources.
  - Identification des solutions pour le maintien des activités critiques.

Très haute disponibilité : focus sur blue-green deployment

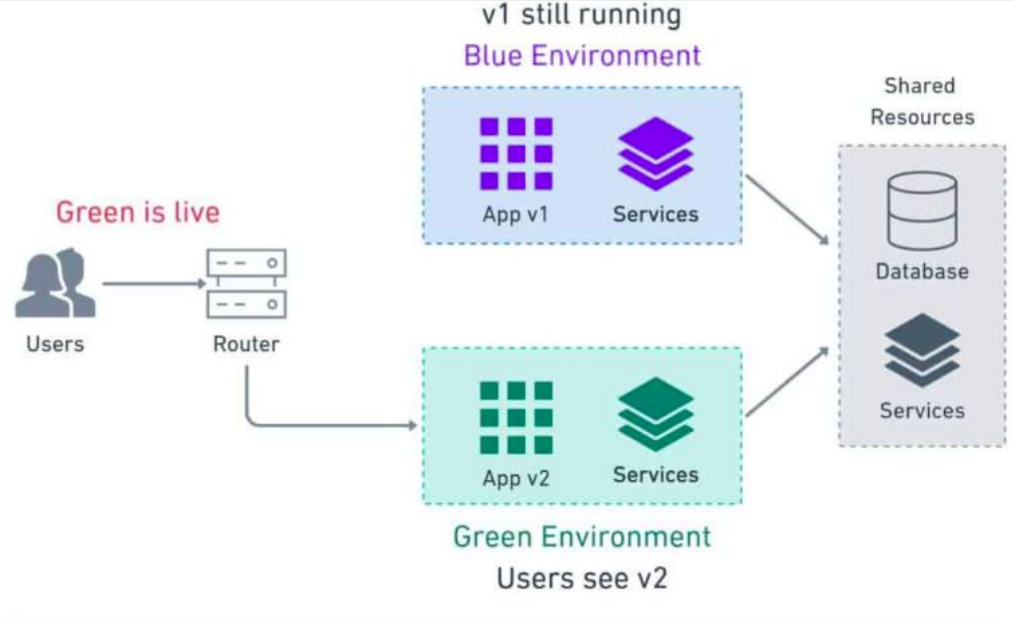
- exemple 1 : avec ressources partagées : étape 1



Users continue accessing v1 on blue while the new v2 release is deployed on green.

Très haute disponibilité : focus sur blue-green deployment

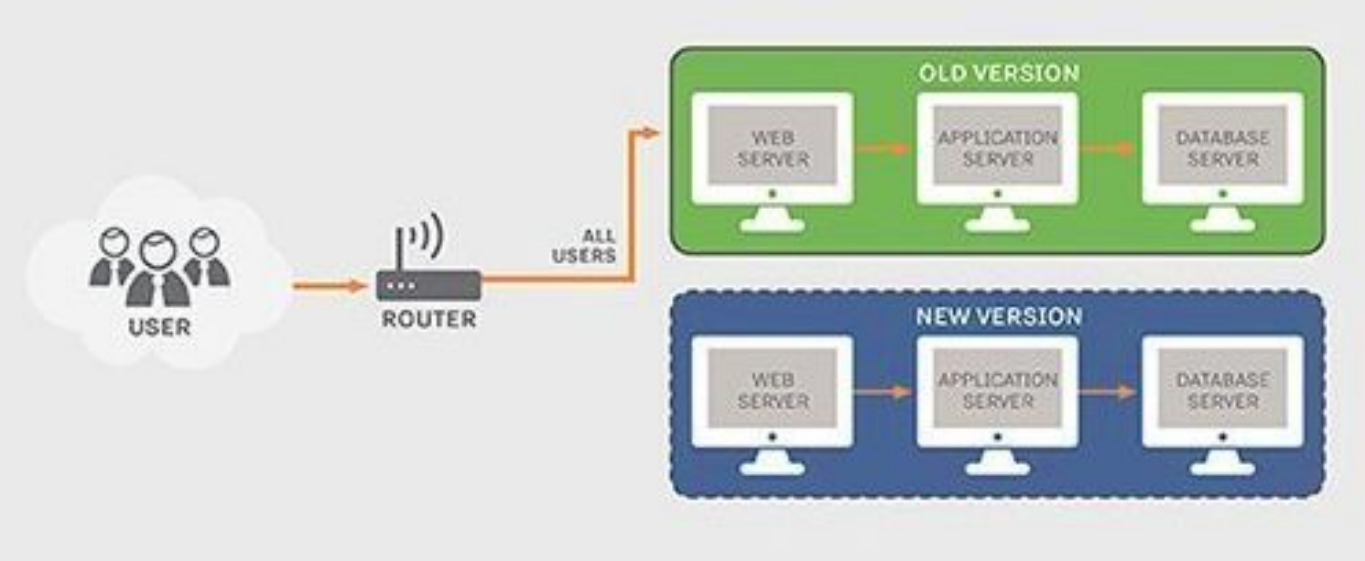
- exemple 1 : avec ressources partagées : étape 2



Deployment is complete once users are switched to the new version running on green

Très haute disponibilité : focus sur blue-green deployment

- exemple 2 : avec ressources dédiées



Très haute scalabilité : définitions

- Définition : capacité d'un produit à s'adapter à un changement d'ordre de grandeur de la demande, en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande.

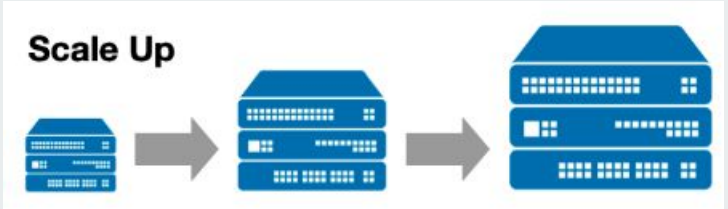
- Scalabilité verticale : possibilité d'augmenter les capacités d'un serveur (CPU, RAM, Stockage).

- Scalabilité horizontale : possibilité d'ajouter des serveurs d'un type donné. Par exemple : ajout possible de serveurs d'application web avec répartition de charge.

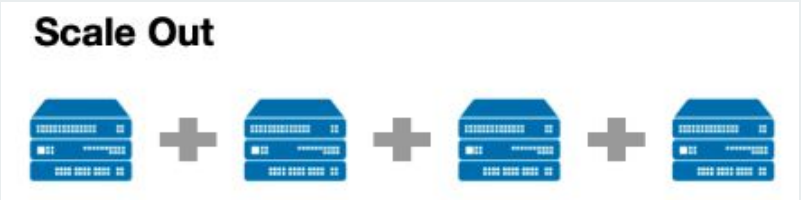
Très haute scalabilité : quelques bonnes pratiques

- Privilégier la scalabilité horizontale à la scalabilité verticale.

  La scalabilité verticale (scale up) peut être mise en œuvre très rapidement mais offre une solution à court terme.



  La scalabilité horizontale (scale out) est mise en oeuvre avec une solution d'équilibrage de charge (load balancing)



Très haute scalabilité : quelques bonnes pratiques

- Privilégier une architecture micro-services qui permet la mise en œuvre de la scalabilité sélective par service.
- Mettre en place des services sans état (stateless)
- Mettre en place des solutions de cache
- Privilégier les traitements asynchrones (message queuing) quand c'est possible
- Privilégier les appels non bloquants (Non Blocking IO)

Très haute scalabilité : focus sur la fonction load balancing

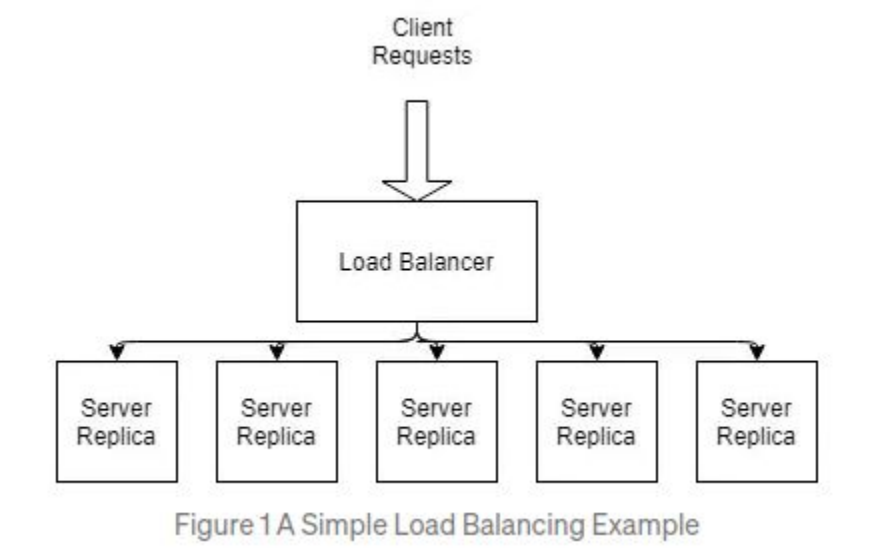
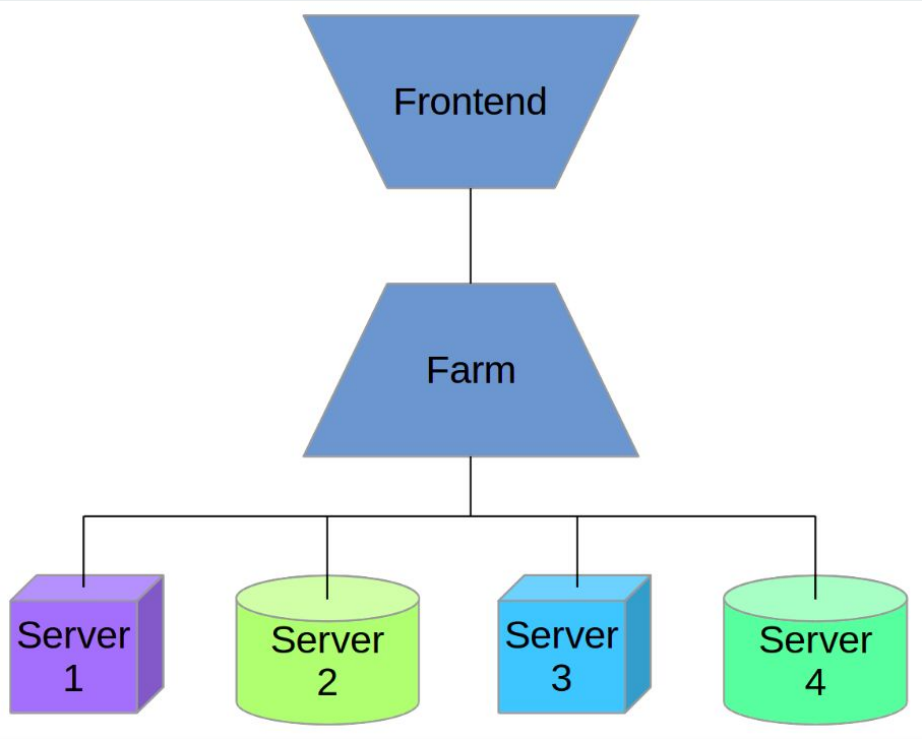


Figure 1 A Simple Load Balancing Example

load balancing avec dimensionnement “manuel”

Très haute scalabilité : focus sur la fonction load balancing



Exemple : IPLB OVH

- Frontend : définit le type de protocole (HTTP/TCP/UDP) du service OVHcloud Load Balancer. C'est également la partie qui expose le port d'écoute du service
- Ferme : reçoit le trafic provenant du frontend, c'est la partie qui s'occupe de faire la répartition de charge
- Serveur : ce sont les serveurs qui reçoivent le trafic final et qui répondent via l'application
- Route et règle : gestion des redirections du trafic entrant vers les fermes correspondant à la règle définie (ex : en fonction d'une url)

Très haute scalabilité : focus sur la fonction load balancing

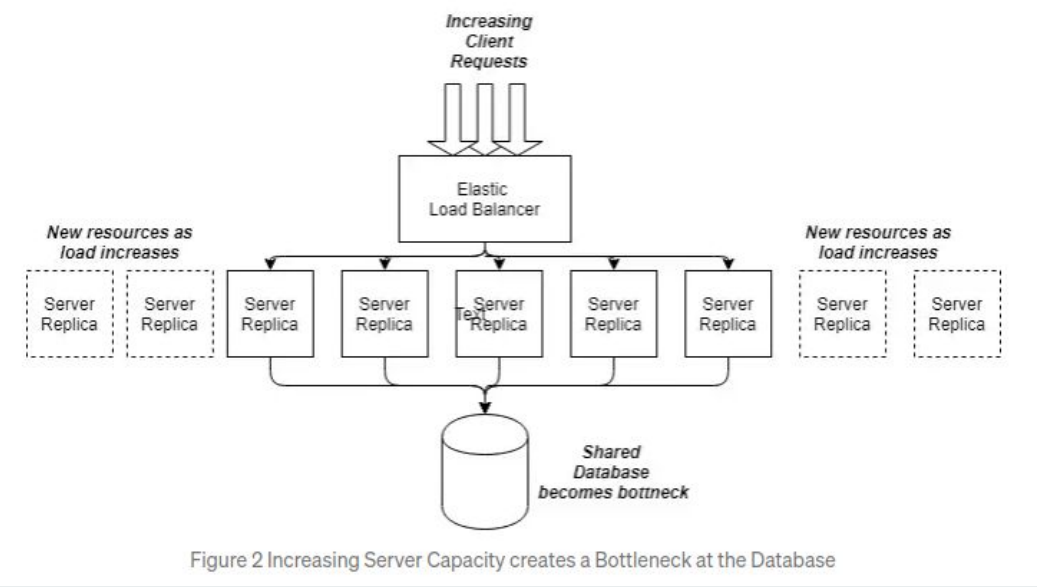


Figure 2 Increasing Server Capacity creates a Bottleneck at the Database

load balancing avec redimensionnement "automatique" :
Elastic Load Balancing

Très haute scalabilité : cas de la base de données

Scénario d'une montée en charge progressive d'une base de données relationnelle (SGBDR)

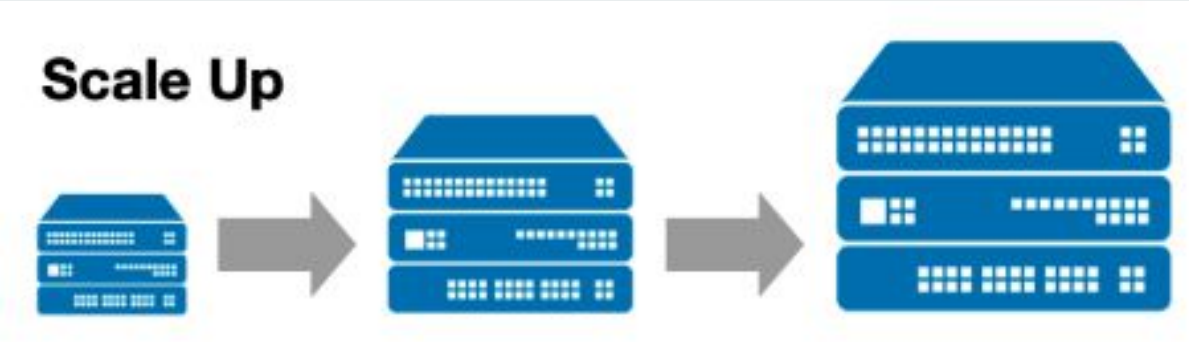
Etape 1 : quelques bonnes pratiques et optimisations

- mise en place de pools de connexions
- mise en place et optimisations des index
- éviter les "select * "
- éviter ou optimiser les sous requêtes (sub select)
- éviter les select avec like %

Très haute scalabilité : cas de la base de données

Etape 2 : scalabilité verticale

augmentation CPU, RAM, stockage



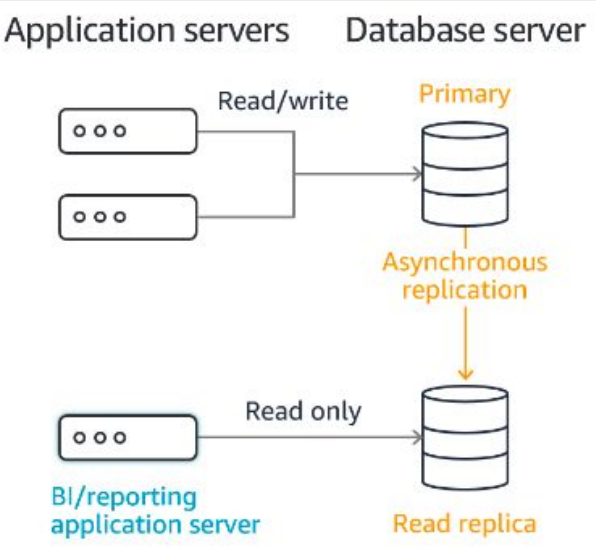
Facile à mettre en oeuvre mais vite limité

Très haute scalabilité : cas de la base de données

Etape 3 : mise en place d'une réplication pour les requête en lecture seule (read replica)

Cette étape permet de "soulager" le serveur primaire d'une partie des connexions (les requêtes en lecture seule).

Mais que se passe-t-il si le nombre de requêtes en écriture augmente ?

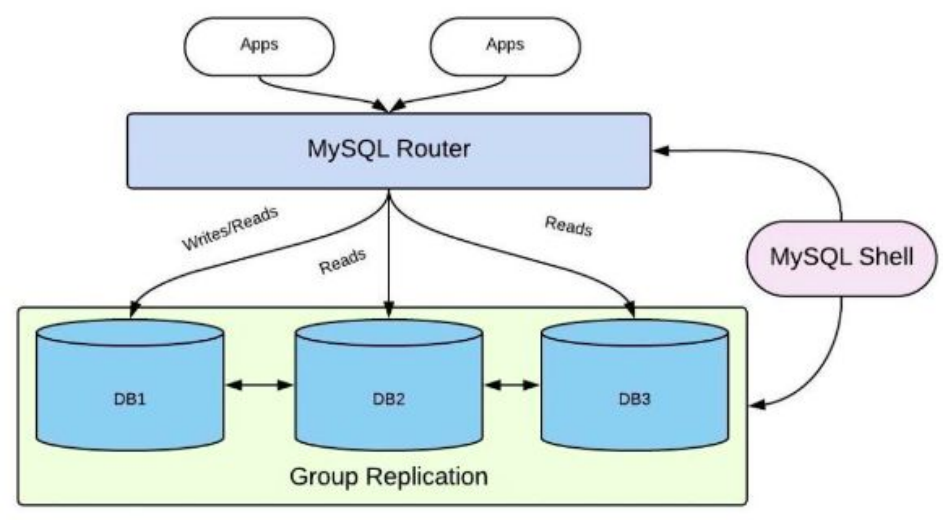


Très haute scalabilité : cas de la base de données

Etape 4 : mise en place d'une réplication "multi-primaire"

Cette étape permet d'absorber une forte quantité de requêtes en lecture seule ou lecture / écriture.

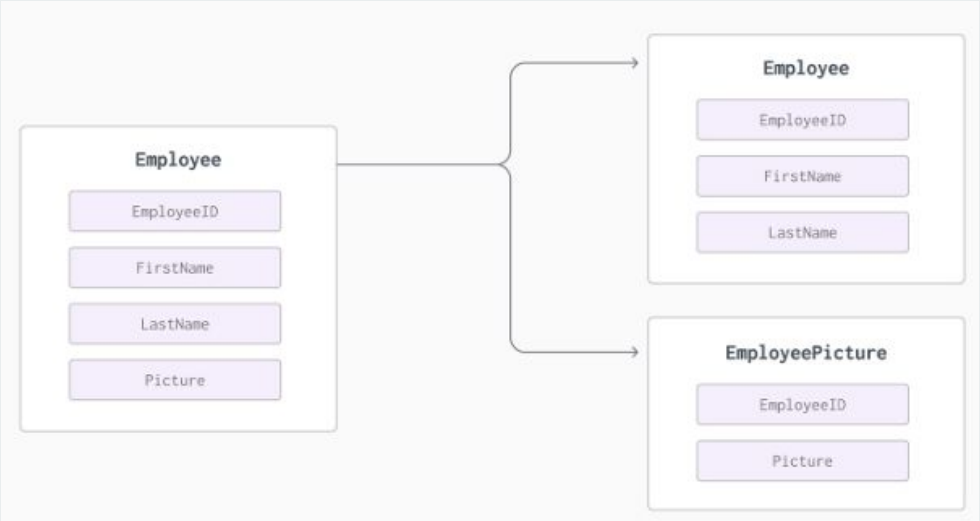
Mais que se passe-t-il en cas de forte volumétrie de données ?



Très haute scalabilité : cas de la base de données

Etape 5 : mise en place du partitionnement vertical

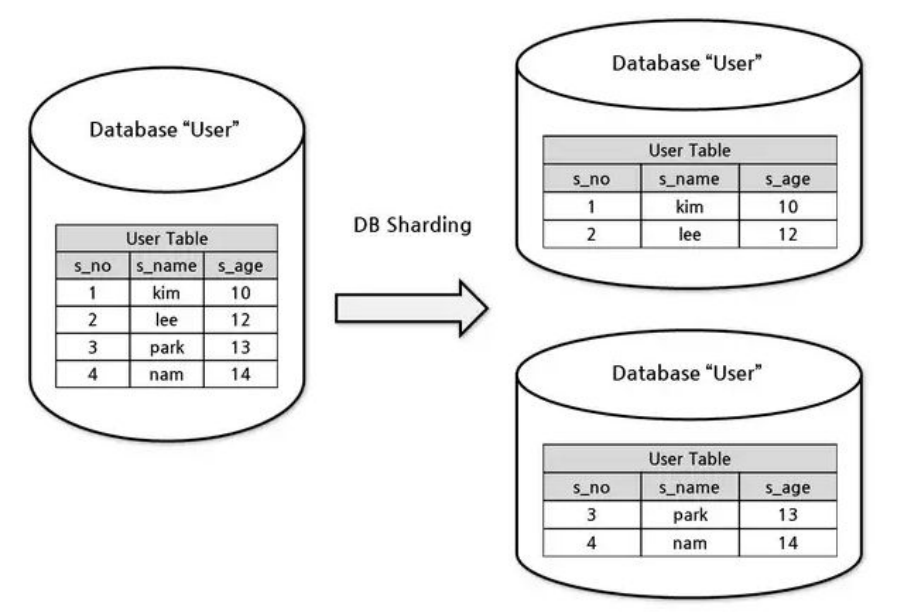
Répartir les colonnes d'une table sur plusieurs tables de base de données



Très haute scalabilité : cas de la base de données

Etape 6 : mise en place du partitionnement horizontal ou sharding

Répartir les lignes d'une table sur plusieurs instances de base de données



Très haute scalabilité : cas de la base de données

Etape 6 : mise en place du partitionnement horizontal ou sharding

Plusieurs stratégies de sharding :

- "Directory-Based Sharding" : un service (lookup service) référence l'emplacement (le shard) de chaque ligne
- "Range-based Sharding" : ex : toutes les lignes avec l'id compris entre 1-100 sont sur le serveur 1, toutes les lignes avec l'id compris entre 101-200 sont sur le serveur 2, …
- "Hash-based" Sharding : une fonction (hash function) détermine le serveur de stockage en fonction de l'id de la ligne
- Sharding géographique : répartition des lignes par pays ou région

Principe : Commodity hardware

- Agencement de nombreuses machines de grande série à la place d'un seul grand système.
- Permet de diminuer le coût moyen de transaction : coût relatif à la transaction 3 fois moins élevé pour un serveur d'entrée de gamme que pour un serveur haut de gamme.

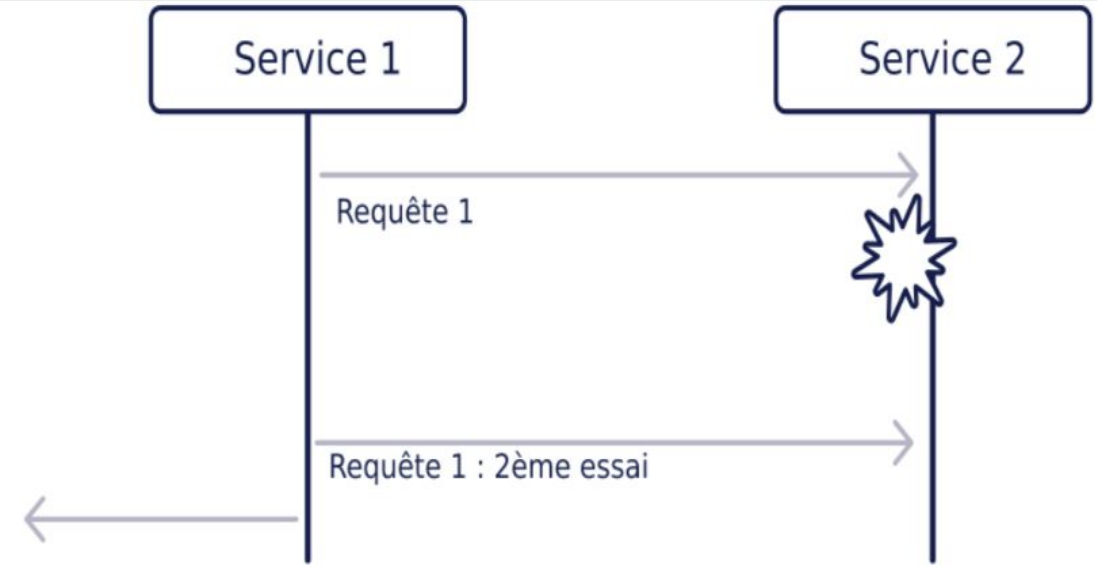
Principe : Design for failure

- "Everything fails all the time." – Werner Vogels (CTO & VP Amazon)
- il est impossible de prévoir toutes les défaillances qui peuvent se produire sur un système informatique, à toutes les couches : une règle de gestion incohérente, des ressources systèmes non relâchées après une transaction, une panne de disque dur…
- d'où le principe "design for failure", ("conçu pour supporter la défaillance") : une application informatique doit être capable de supporter la panne de n'importe quel composant logiciel ou matériel sous-jacent.

Principe : Design for failure

- Priorité haute sur les services et fonctionnalités jugés indispensables
- “Eventual consistency” : au lieu de rechercher systématiquement la consistance à chaque transaction, avec des mécanismes souvent coûteux type XA, la consistance est assurée à la fin (eventual), lorsque les services défaillants sont à nouveau disponibles
- “Graceful degradation” : en cas de pics de charge, les fonctionnalités coûteuses en performance sont désactivées à chaud
- “Obsession de la mesure” : tout est mesuré et supervisé

Pattern : Retry

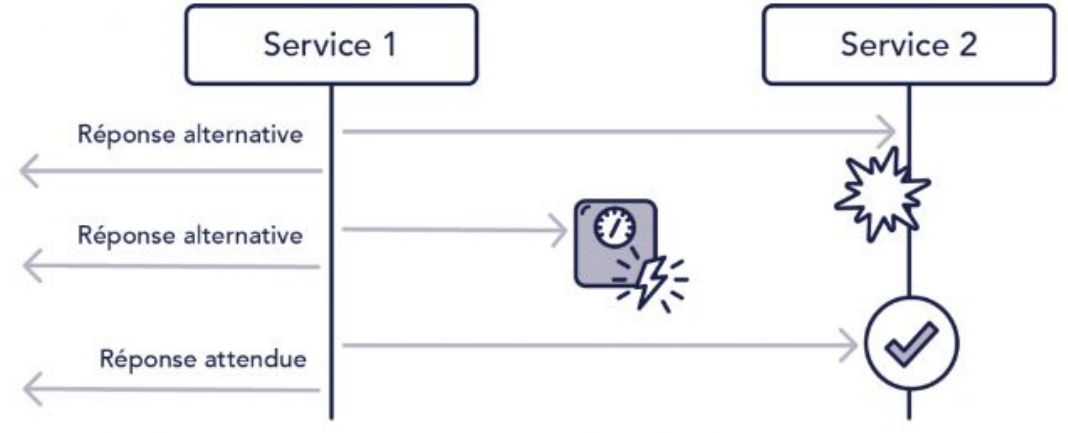


- Objectif : consiste à envoyer à nouveau la requête qui a échoué.
- Inconvénients : si le service appelé reste hors service, l'application risque une surcharge en multipliant les requêtes

Pattern : Timeout

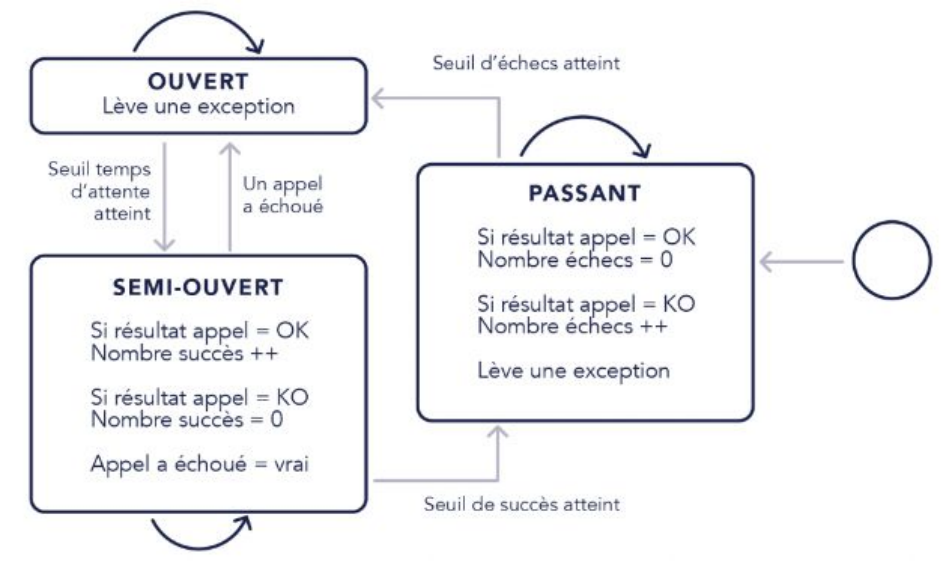
- objectif : ne pas attendre indéfiniment une réponse en positionnant un temps d'attente maximal.
- inconvénients :
  - Si le service appelé est hors service, le service appelant va quand même faire l'appel.
  - L'erreur n'arrivera qu'après le temps d'attente maximum
  - Consommation de ressource (connexion, mémoire…) inutile, car nous aurons une erreur à la fin

Pattern : Circuit Breaker



- Objectif : permet de contrôler la collaboration entre différents services afin d'offrir une grande tolérance à la latence et à l'échec.
- Ce pattern agit comme un proxy implémentant une machine à états (Ouvert, Passant (fermé), Semi-ouvert) pour l'apprentissage de l'état du service.
  On peut aussi le voir comme un feu tricolore de signalisation.

Pattern : Circuit Breaker



- En temps normal le circuit breaker est en mode passant.
- Lorsque le nombre d'échecs successifs (ou toute autre métrique) dépasse un seuil, le circuit s'ouvre pour ne plus laisser passer de requêtes. À ce moment-là, deux mécanismes se déclenchent :
- Mise en place de la réponse alternative de repli
- Activation du processus du passage à l'état semi-ouvert (ici nous déclenchons un minuteur)
  Une fois le seuil de passage à l'état semi-ouvert atteint (ici seuil du temps d'attente), le circuit breaker laisse à nouveau passer quelques requêtes et passe dans l'état passant si tout se déroule bien.

- Avec un peu d'imagination, nous pouvons envisager une infinité de possibilités :
  Stocker toutes les requêtes en erreur avec le maximum de détail pour les traiter plus tard
  Avoir plusieurs stratégies de réponse alternative en fonction du type d'erreur renvoyé par le service appelé (code "HTTP 503 Service Unavailable", mauvaise réponse…)
  Avoir des seuils intelligents qui s'adaptent après une période d'apprentissage.

Quiz 5

https://docs.google.com/forms/d/e/1FAIpQLSfyGjWg7k6IDb334KgJpM2Tr0oPYPx12SxQLNtrq23glumsYQ/viewform?usp=dialog

Etude de cas : Netflix

Quelques chiffres

- + de 180 M comptes utilisateurs
- dans + de 200 pays
- support de + 2200 appareils impliquant de nombreuses combinaisons résolution / format
- pour chaque vidéo, + 1100 réplicas
- poids moyen d'une vidéo : 500 Mo
- nombre de vidéos vues par jour : 500 M
- trafic sortant quotidien : 500 M x 500 Mo = 250 Po (250 000 To)
- nombre de vidéos entrantes (à traiter) par jour : 1000

Etude de cas : Netflix

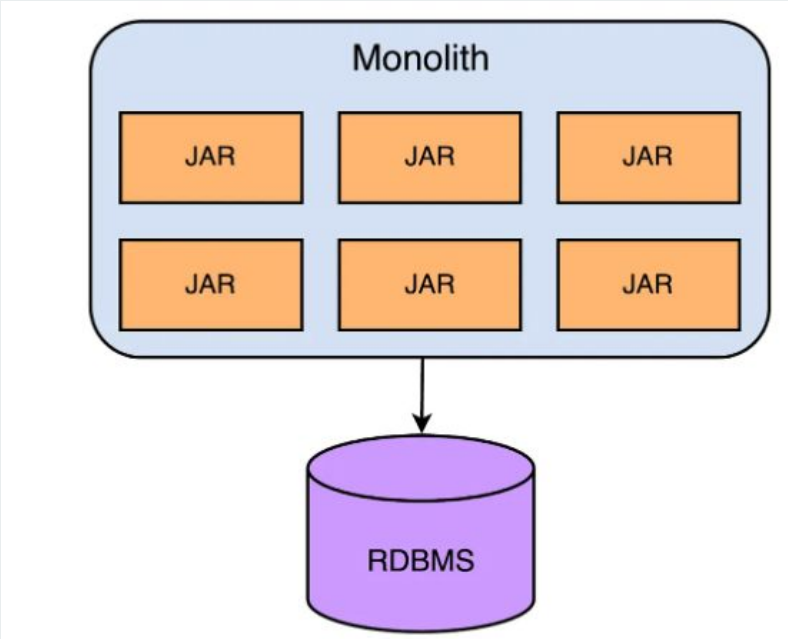
Historique

- fondé en 1997 en Californie
- fourniture d'un service en ligne de location et d'achat de DVD, livrés à domicile

- Le service de vidéo à la demande par abonnement mensuel est lancé en 2007.

Etude de cas : Netflix

Architecture initiale



Etude de cas : Netflix

- août 2008 :
  - corruption majeure de la base de données
  - 3 jours d'indisponibilité totale

- 2 décisions importantes en réponse à cet événement
  - migration des données vers AWS
  - évolution vers une architecture micro-services

Etude de cas : Netflix

Quelques principes majeurs :

- Achat vs Développement interne
  - Privilégier l'utilisation ou la contribution aux technologies open source.
  - Développer à partir de zéro uniquement ce qui est absolument nécessaire.
- Services sans état
  - Les services doivent être conçus sans état, à l'exception des couches de persistance et de cache.
  - Pas de sessions persistantes.
  - Réalisation de tests de chaos pour vérifier qu'une panne d'instance n'affecte pas le système global.
- Scaling horizontal vs scaling vertical
  - Le scaling horizontal offre une plus grande marge de manœuvre en termes d'évolutivité.
  - Scaling horizontal privilégié vs scaling vertical

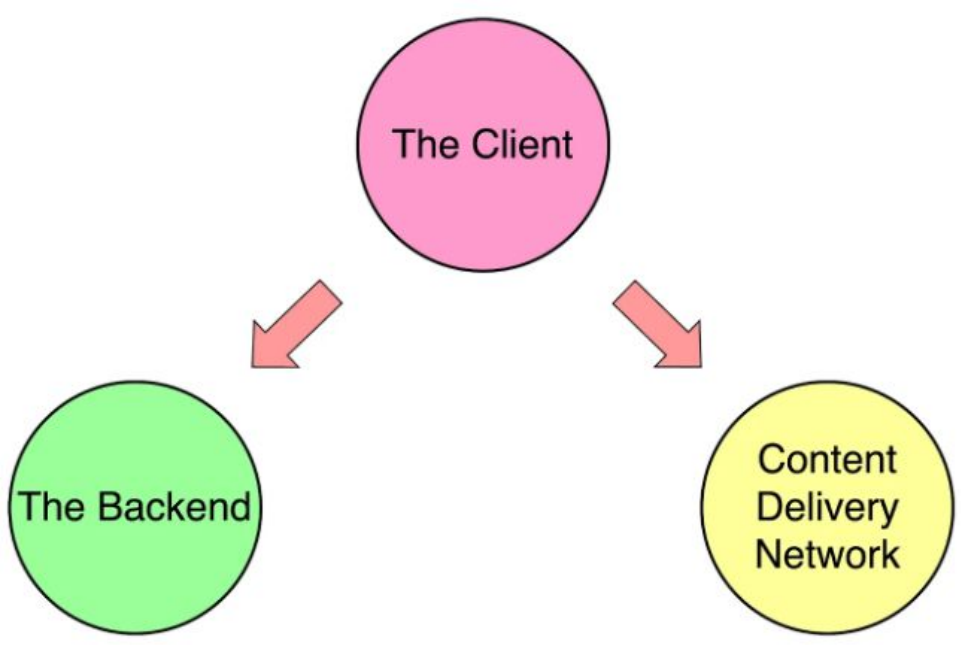
Etude de cas : Netflix

Quelques principes majeurs :

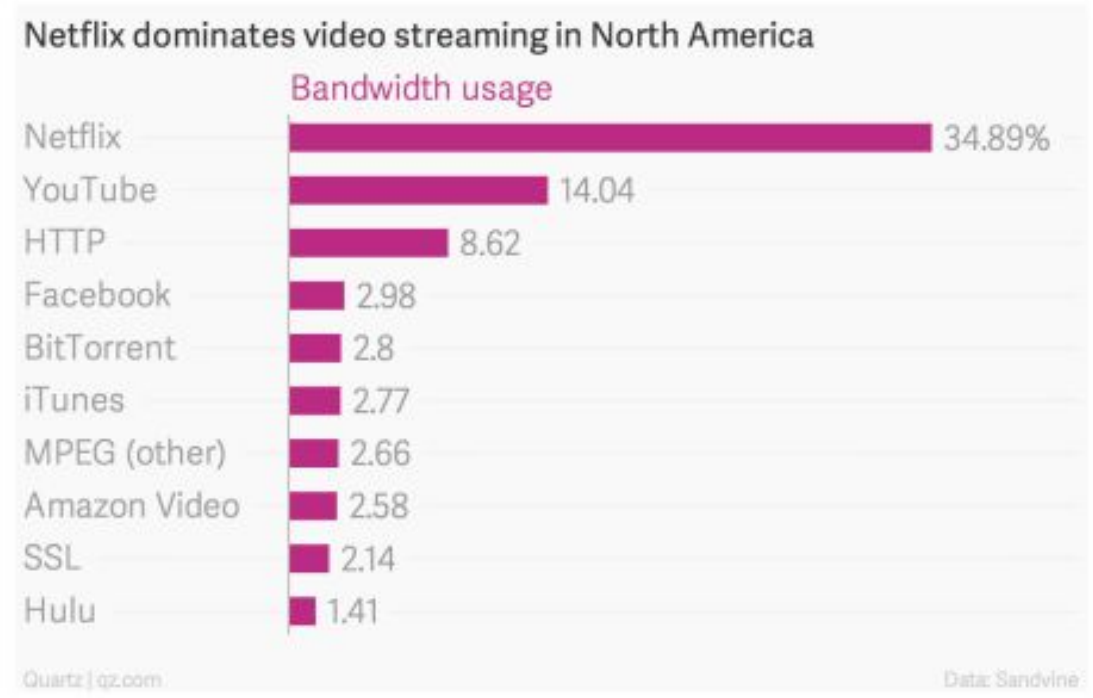
- Redondance et isolation
  - Créez plusieurs copies de chaque élément, par exemple des bases de données répliquées et plusieurs instances de service.
  - Réduisez l'impact d'un problème en isolant les charges de travail.
- Automatisez les tests destructifs
  - Les tests destructifs des systèmes doivent être une activité continue.
  - Utilisez des outils comme Chaos Monkey pour réaliser ces tests à grande échelle.

Etude de cas : Netflix

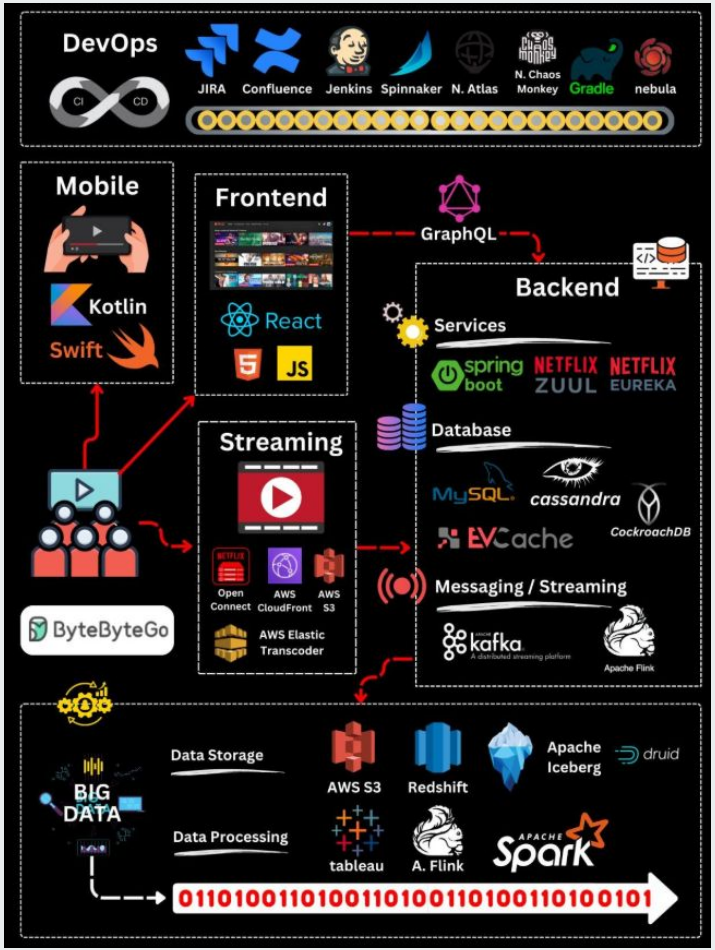
Architecture : 3 parties principales



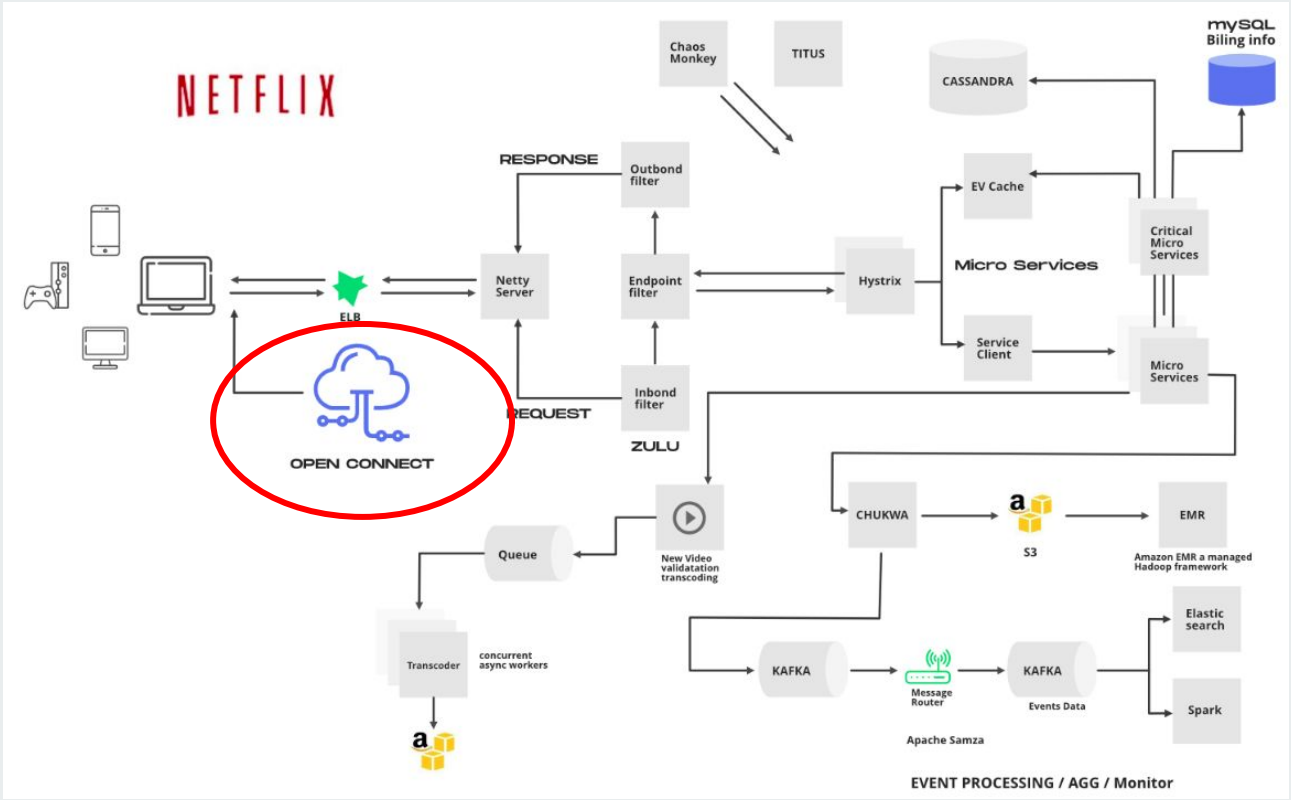
Etude de cas : Netflix



Etude de cas : Netflix : stack technique

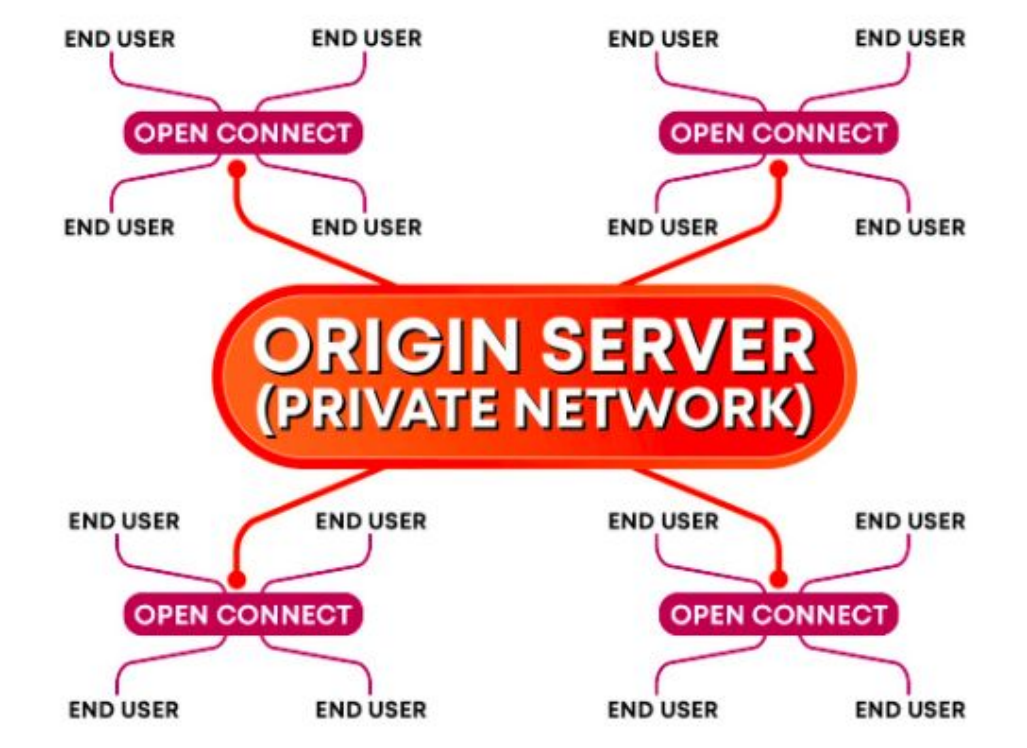


Etude de cas : Netflix : open connect

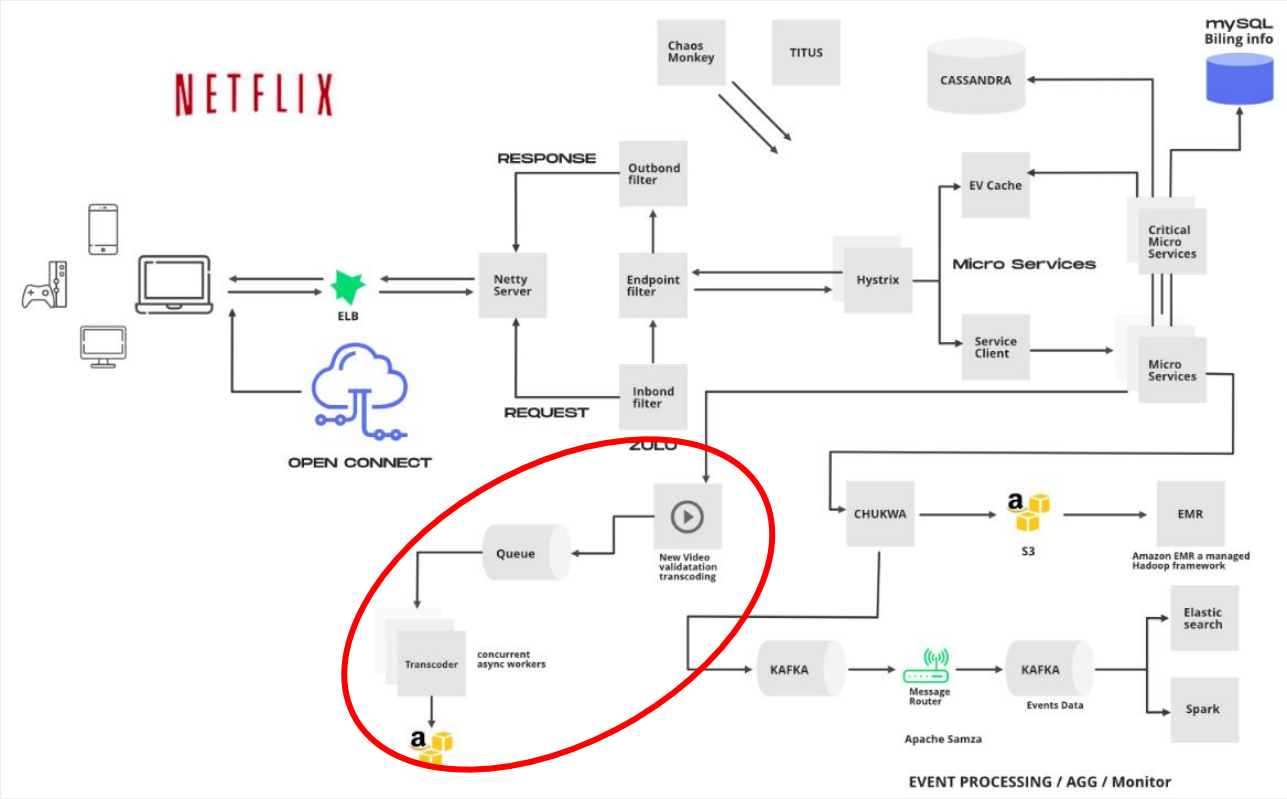


- CDN Netflix “maison”
- CDN : Content Delivery Network
- Définition : Un CDN est un réseau de serveurs qui distribue du contenu à partir d'un serveur « d'origine » dans le monde entier, en mettant en cache le contenu à proximité de l'endroit où chaque utilisateur final accède à Internet.

Etude de cas : Netflix : open connect



Etude de cas : Netflix : gestion des vidéos entrantes



- traitements asynchrones
- stockage des vidéos traitées dans AWS S3
- stockage des métadonnées (description, langue, acteurs, ...) en SGBDR
- distribution des vidéos à diffuser vers les noeuds du CDN Open Connect

Etude de cas : Netflix : gestion des vidéos entrantes

- Netflix reçoit des vidéos des maisons de production en très haute définition : 1000 par jour
- Avant de servir les vidéos aux utilisateurs, il effectue un certain nombre de prétraitements.

- Netflix prend en charge plus de 2200 appareils et chacun d'entre eux requiert des résolutions et des formats différents.

- Pour que les vidéos puissent être visualisées sur différents appareils, Netflix effectue un transcodage ou un encodage, qui consiste à convertir la vidéo originale en différents formats et résolutions.
  1 vidéo entrante HD → environ 1100 “variantes” de cette vidéo

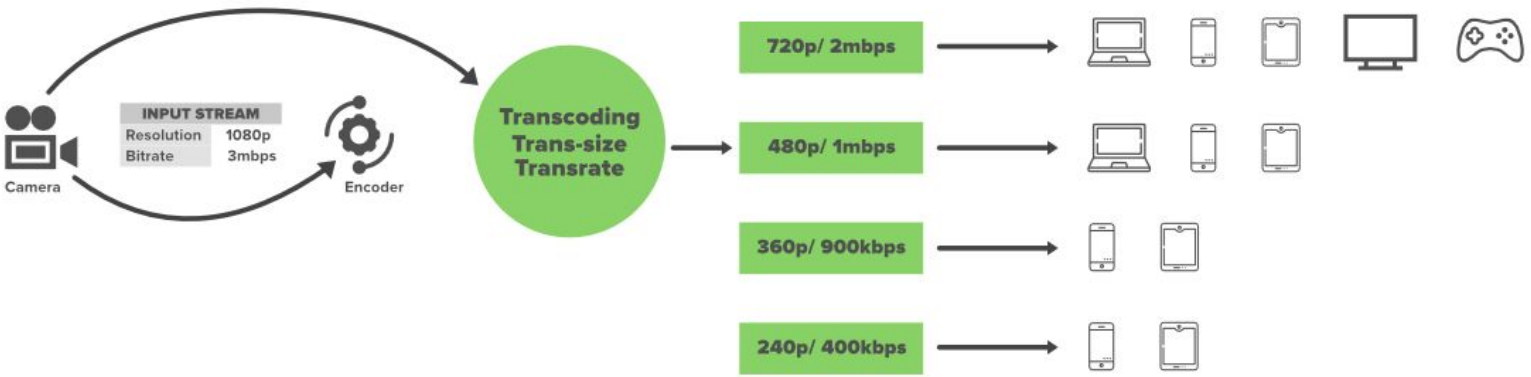
Etude de cas : Netflix : gestion des vidéos entrantes

- Lorsque l'utilisateur appuie sur le bouton de lecture d'une vidéo, Netflix analyse la vitesse du réseau ou la stabilité de la connexion, puis détermine le meilleur serveur Open Connect à proximité de l'utilisateur.
  En fonction de l'appareil et de la taille de l'écran, le bon format vidéo est diffusé sur l'appareil de l'utilisateur.

  Lorsque vous regardez une vidéo, vous avez peut-être remarqué que la vidéo est pixellisée et qu'elle repasse en HD après un certain temps. Cela s'explique par le fait que Netflix vérifie recherche le meilleur serveur de connexion ouvert et passe d'un format à l'autre (pour une expérience visuelle optimale) lorsque cela est nécessaire.

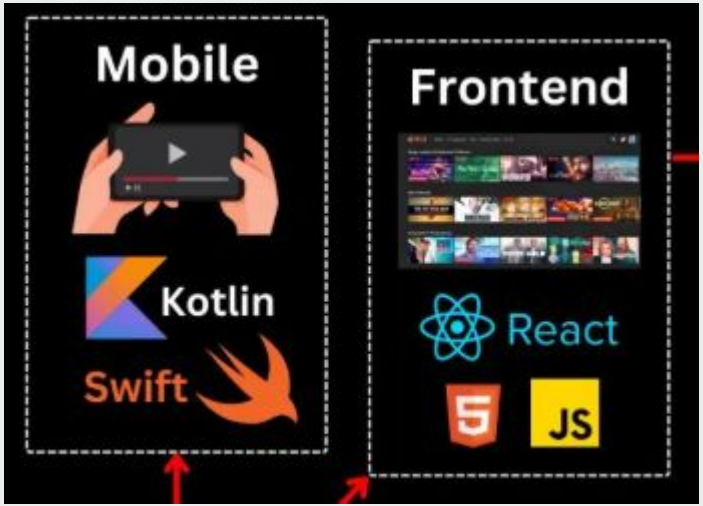
Etude de cas : Netflix : gestion des vidéos entrantes

INPUT PROTOCOLS	INPUT CODECS		OUTPUT CODECS		OUTPUT PROTOCOLS
Adobe RTMP, RTSP/RTP, MPEG-TS, ICY (SHOUTcast/Icecast)	Video:	H.265/HEVC, H.264/AVC, VP9, VP8 MPEG4 Part 2, MPEG2	Video:	H.265/HEVC H.264/AVC, H.263 (v2), VP9	Apple HLS, Adobe HDS, MPEG-DASH, Microsoft Smooth Streaming, Adobe RTMP, RTSP/RTP, MPEG-TS
	Audio:	MP3, AAC, AAC-LC, HE-AAC+ v1 & v2, MPEG1 Part 1/2, Speex, G.711, Opus, Vorbis	Audio:	AAC, AAC-LC, HE-AAC+ v1 & v2, Opus, G.711	



Etude de cas : Netflix

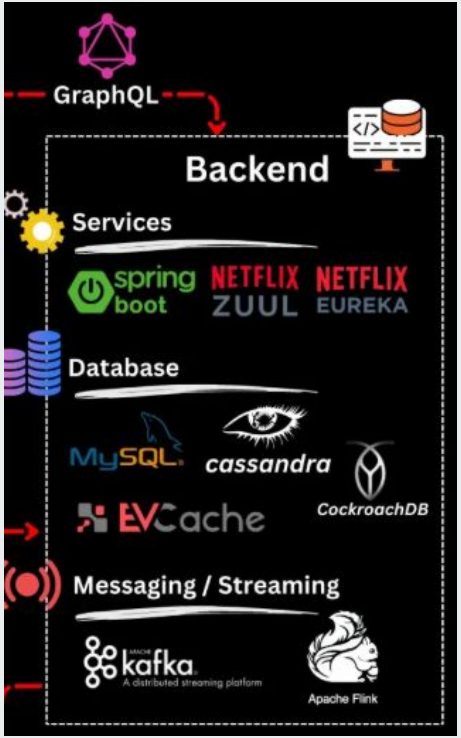
- App mobiles natives : Swift and Kotlin
- Web application : React



Etude de cas : Netflix

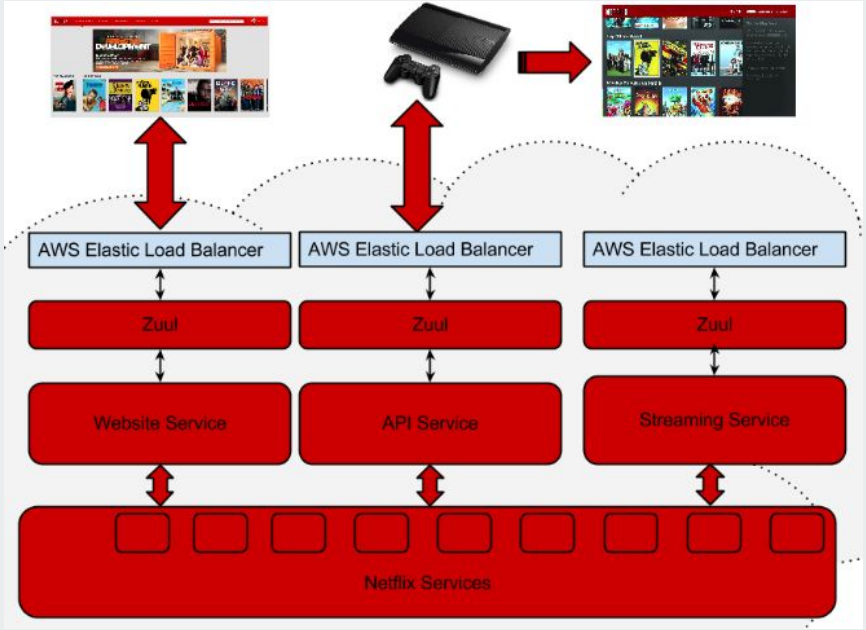
Services backend

- Spring boot
- ZUUL
- Eureka



Etude de cas : Netflix : Zuul

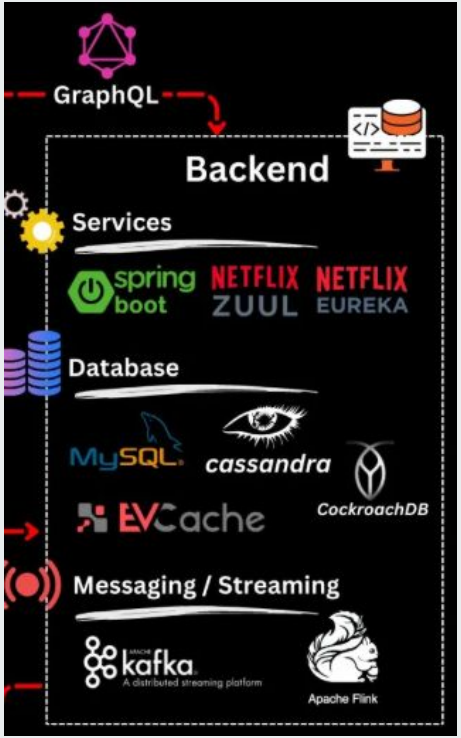
- Zuul : API Gateway



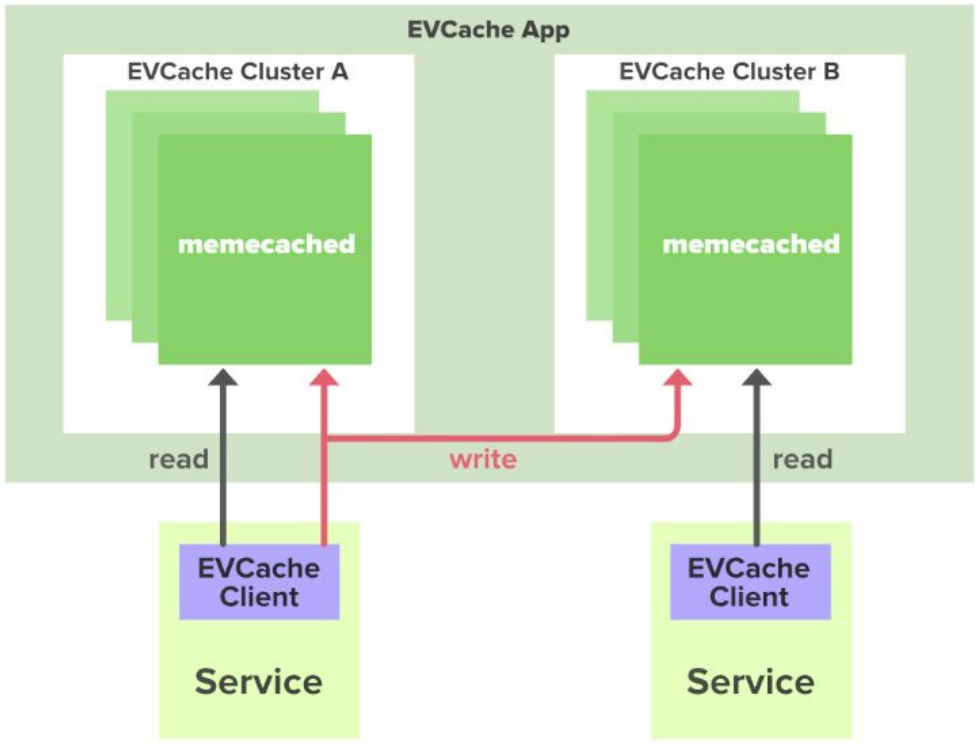
Etude de cas : Netflix

Bases de données

- MySql
- EV cache
- Cassandra
- CockroachDB

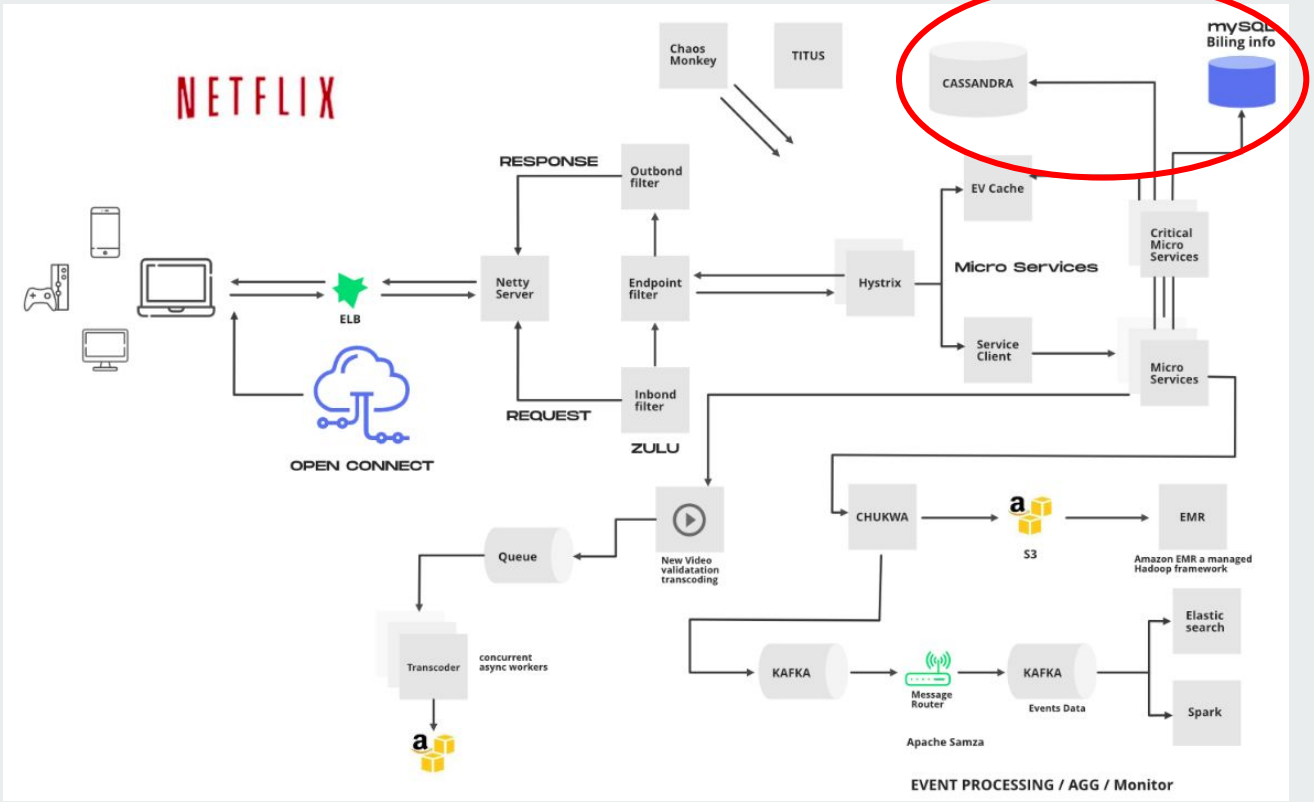


Etude de cas : Netflix : EV Cache



- Les données sont partagées entre les clusters de la même zone et plusieurs copies du cache sont stockées dans les nœuds partagés.
- A chaque écriture,tous les nœuds de tous les clusters sont mis à jour.
- Pour chaque lecture, la demande est faite au cluster le plus proche (et non à tous les clusters et nœuds) et à ses nœuds.
- Dans le cas où un nœud n'est pas disponible, la lecture se fait à partir d'un autre nœud disponible.

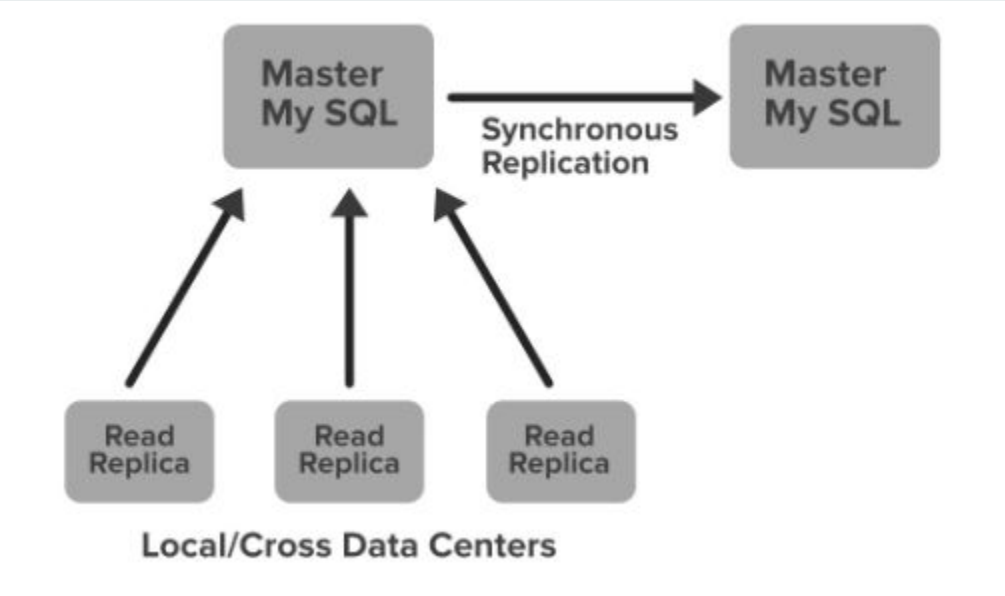
Etude de cas : Netflix : bases de données



2 types de bases de données :

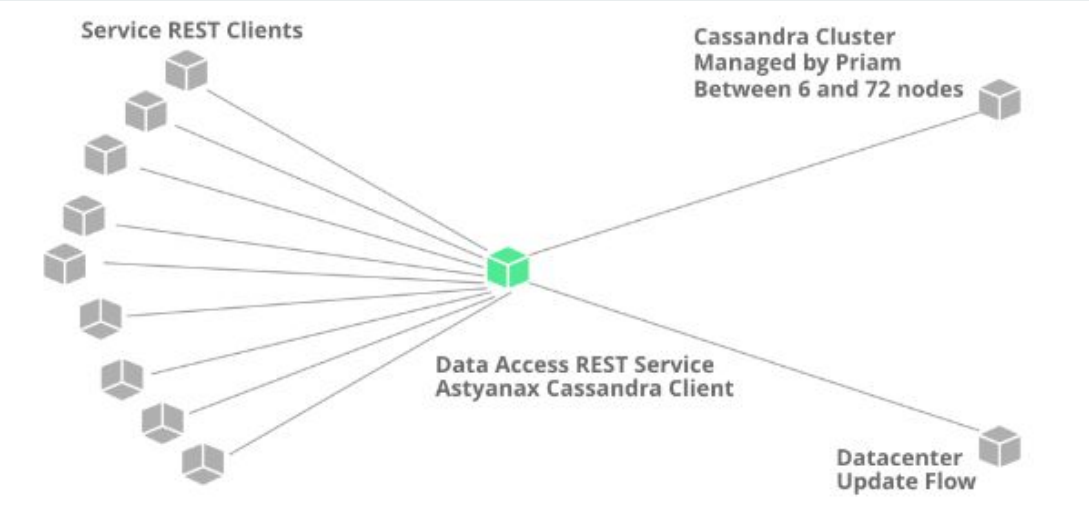
- SGBDR : MySql
- NoSql : Cassandra

Etude de cas : Netflix : bases de données



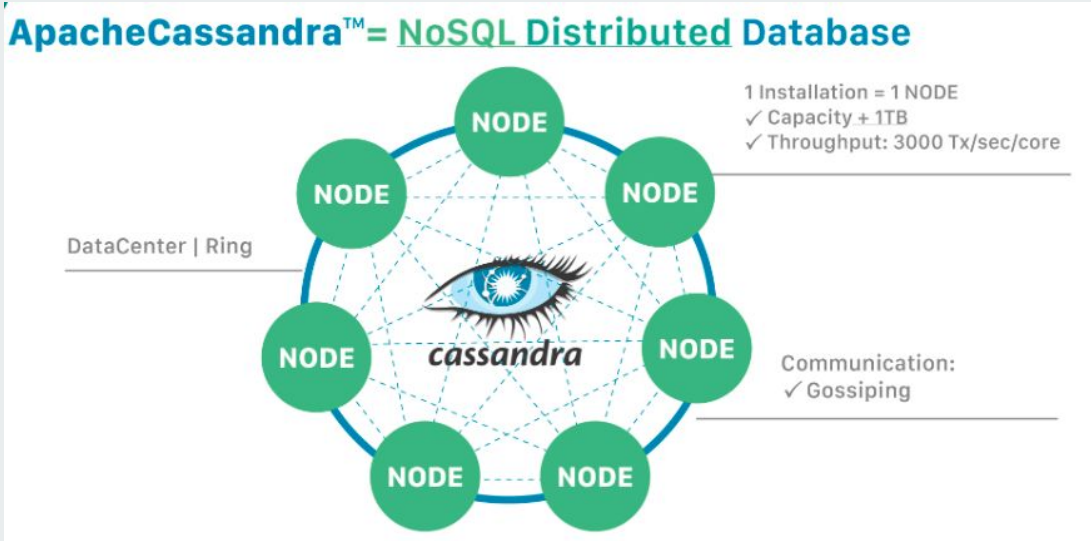
- MySql
- Informations concernant les comptes utilisateurs, la facturation , les abonnements
- Réplication master / master
- Requêtes en lecture dirigées vers les Read Replica

Etude de cas : Netflix : bases de données

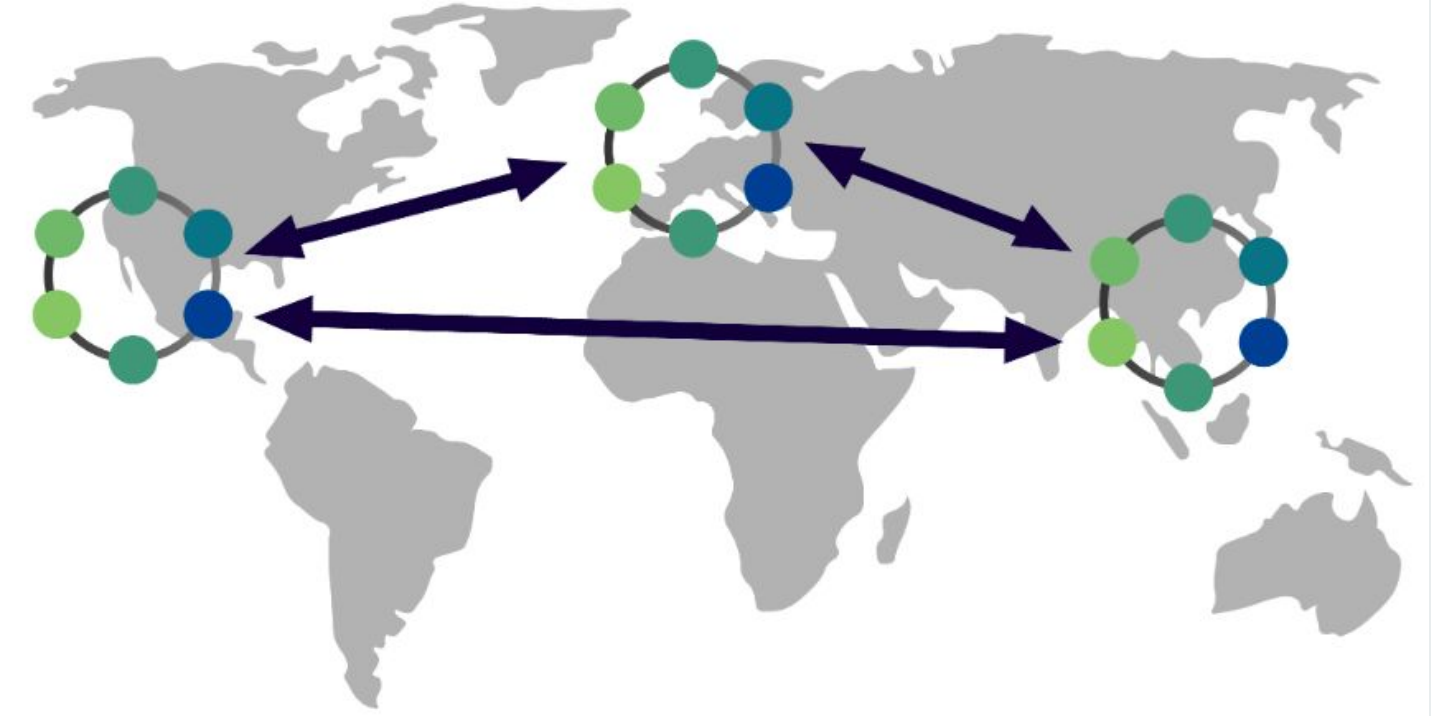


- Cassandra : “wide columns”
- Informations concernant l’historique de consommation des utilisateurs
- Plus de 50 Clusters
- Plus de 500 Noeuds
- Plus de 30 To de sauvegardes quotidiennes

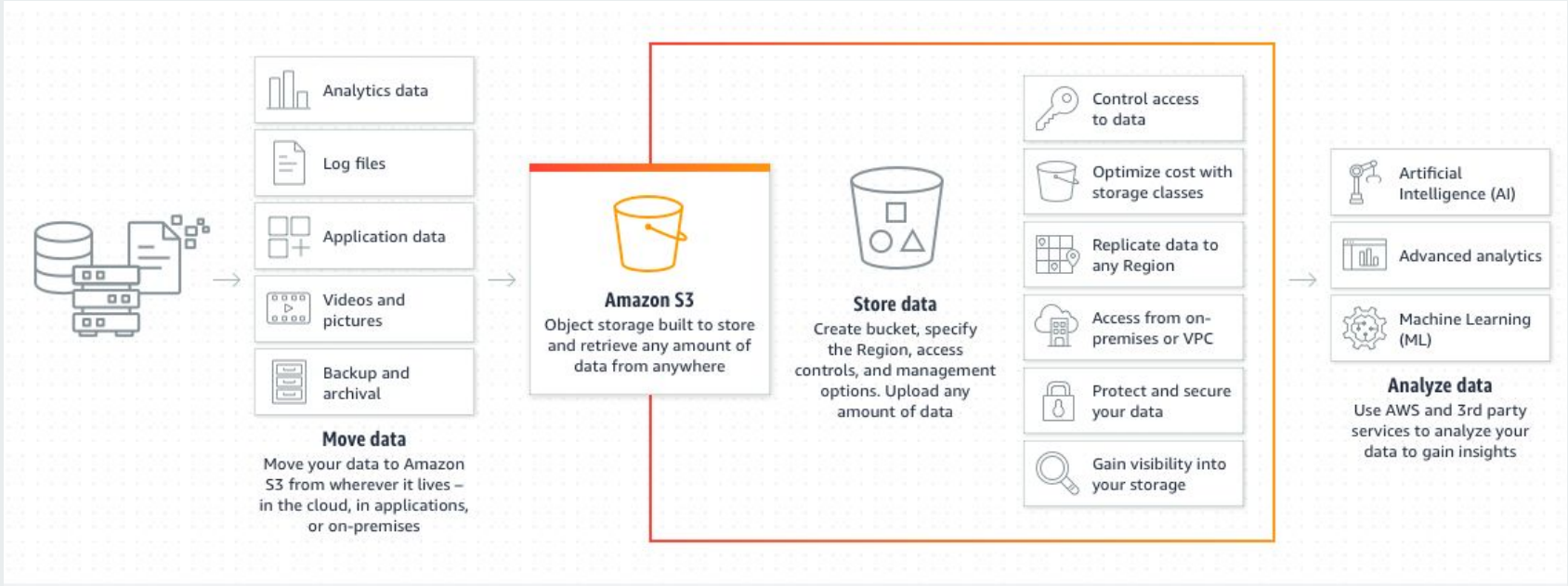
Etude de cas : Netflix : focus sur Cassandra



Etude de cas : Netflix : focus sur Cassandra

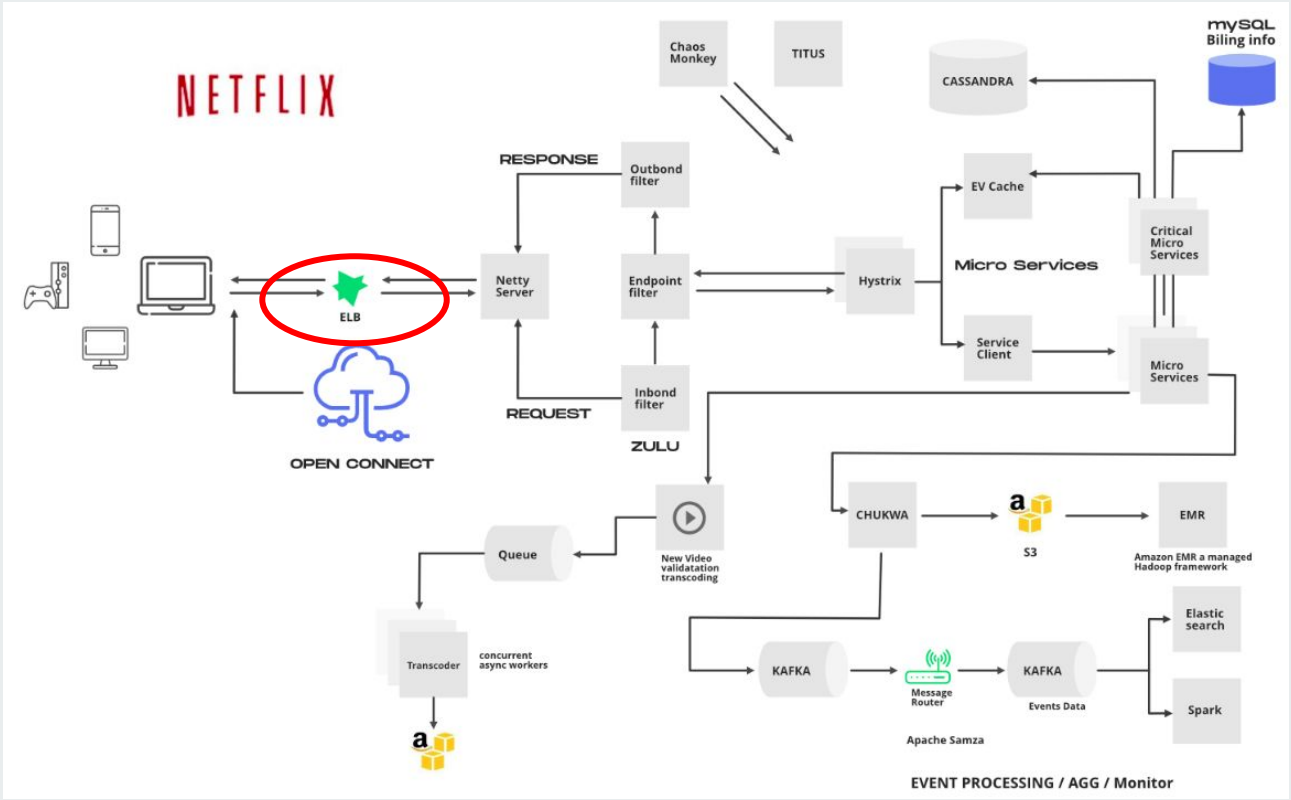


Etude de cas : Netflix : focus sur AWS S3



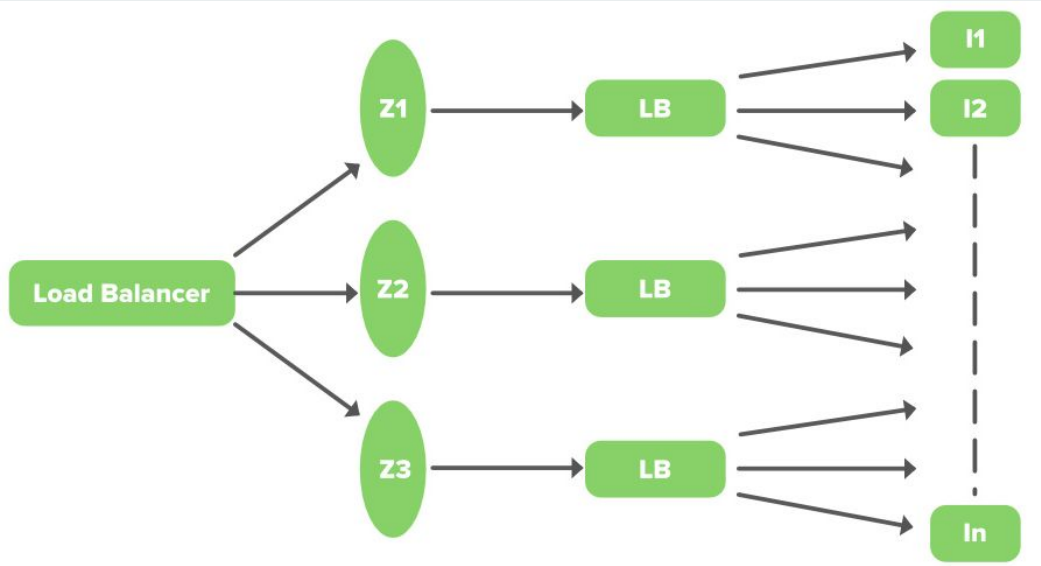
AWS S3 : [AWS Simple Storage Service]

Etude de cas : Netflix : elastic load balancer



ELB : Elastic Load Balancer

Etude de cas : Netflix : elastic load balancer



2 niveaux de load balancing :

- 1 : zones
- 2 : instances

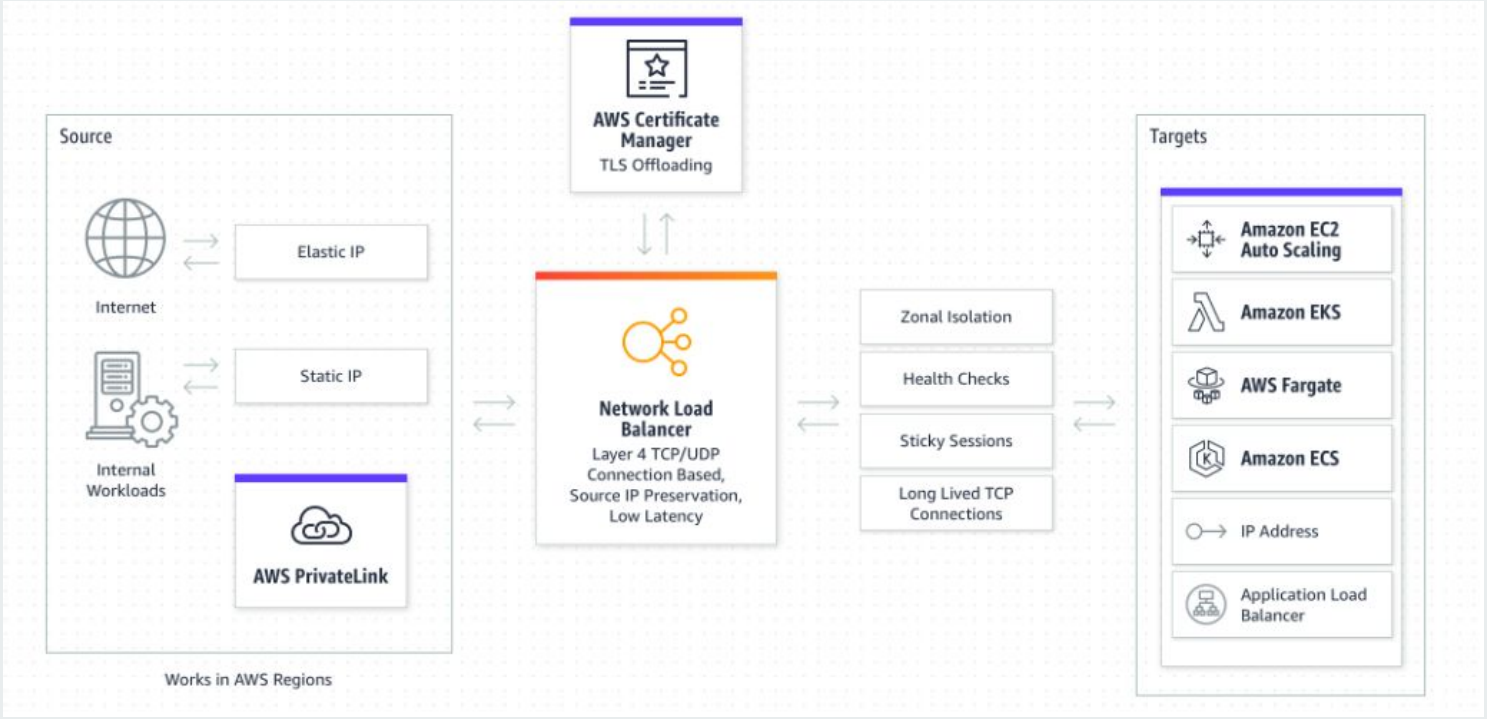
Etude de cas : Netflix : focus sur AWS ELB

Chez AWS, 3 types de load balancing :

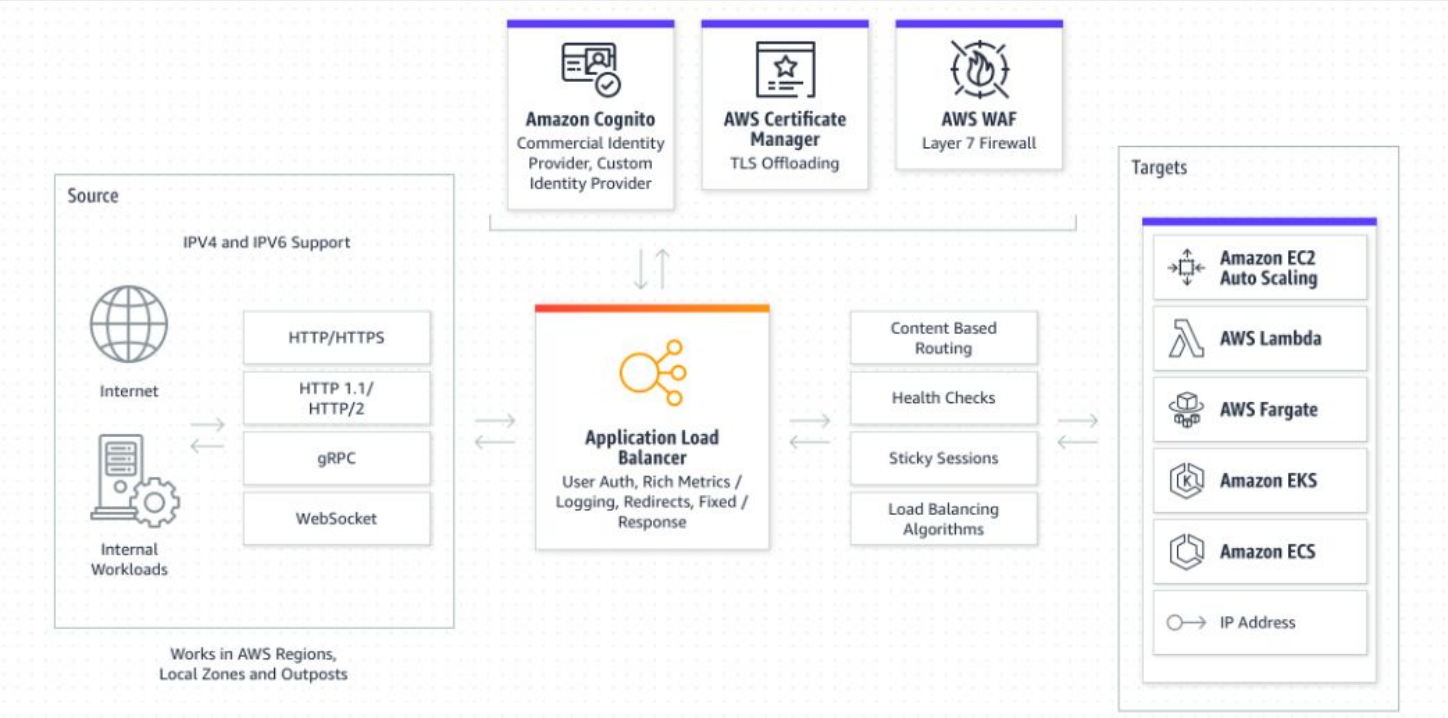
- Application Load Balancing
- Gateway Load Balancing
- Network Load Balancing

https://aws.amazon.com/fr/elasticloadbalancing/features/?nc=sn&loc=2&dn=1

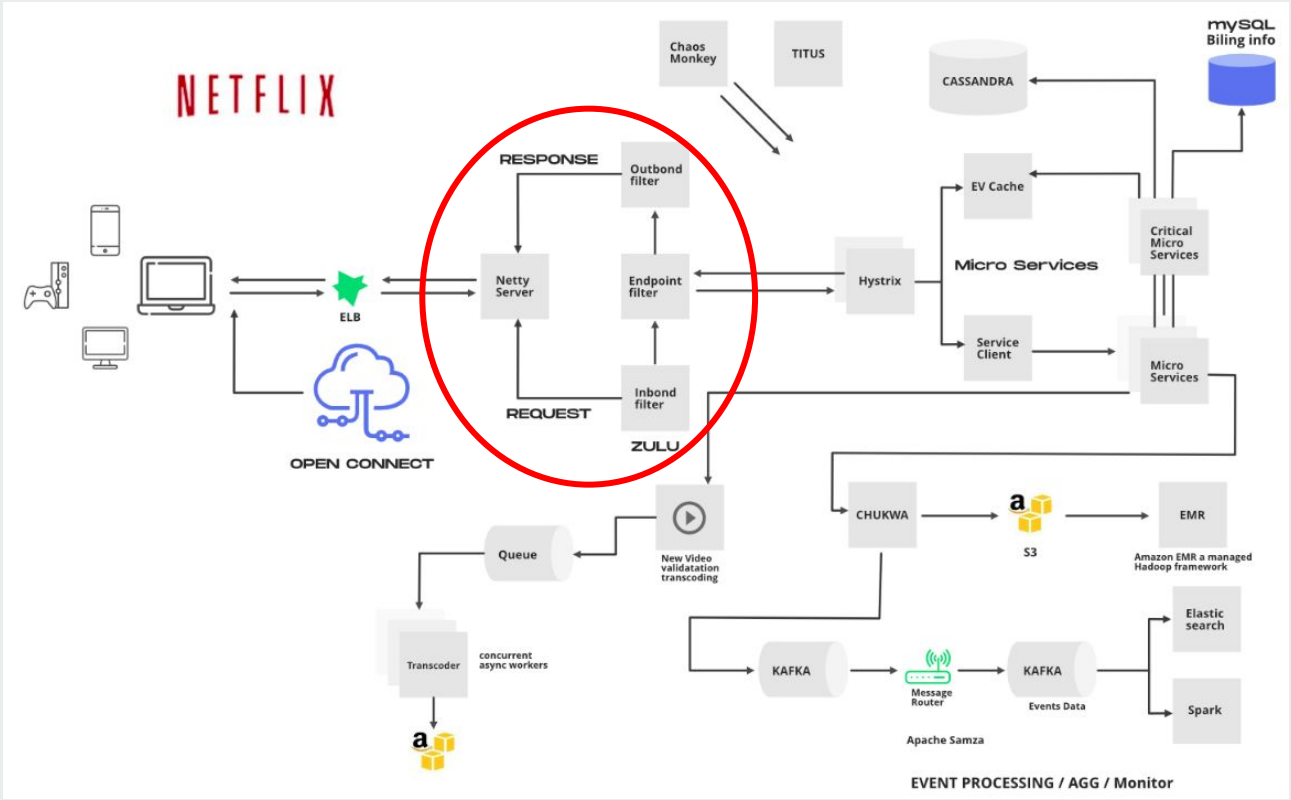
Etude de cas : Netflix : focus sur AWS ELB



Etude de cas : Netflix : focus sur AWS ELB



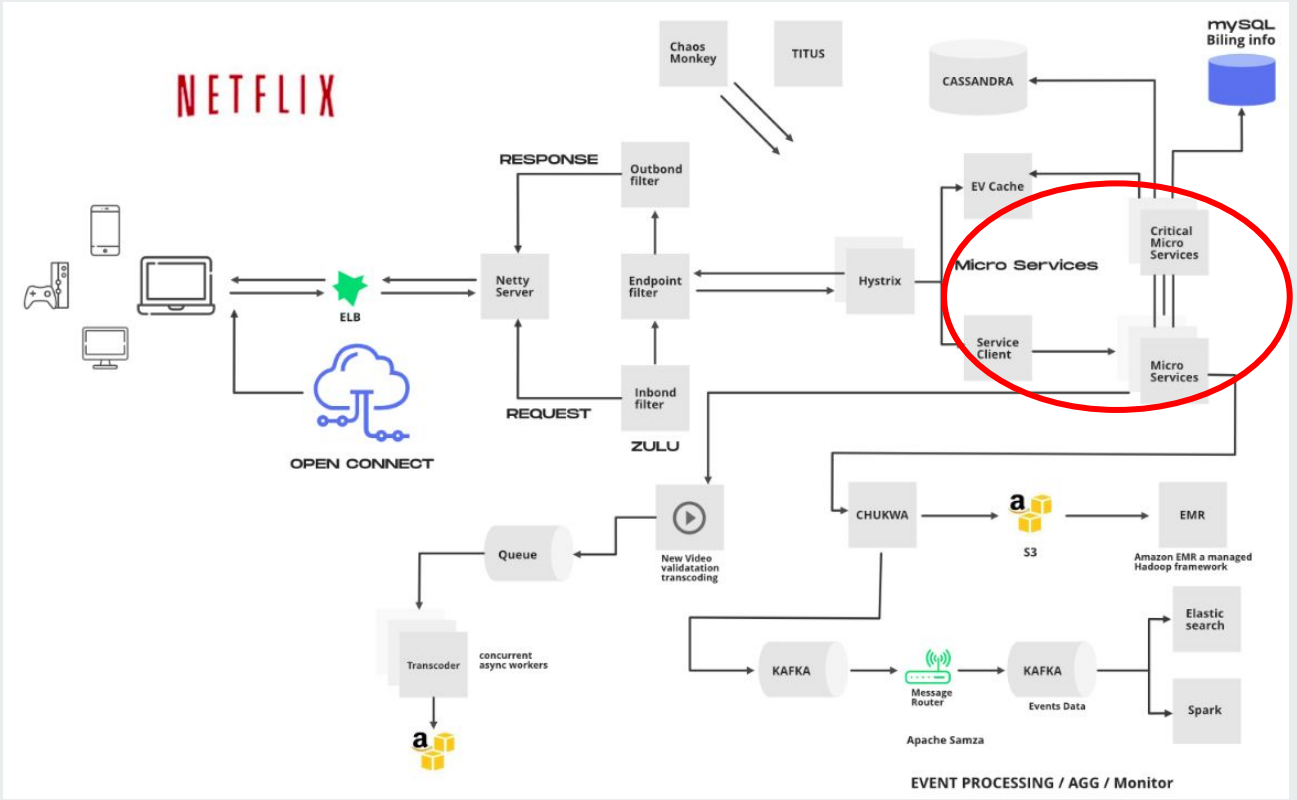
Etude de cas : Netflix : API Gateway



Plusieurs composants :

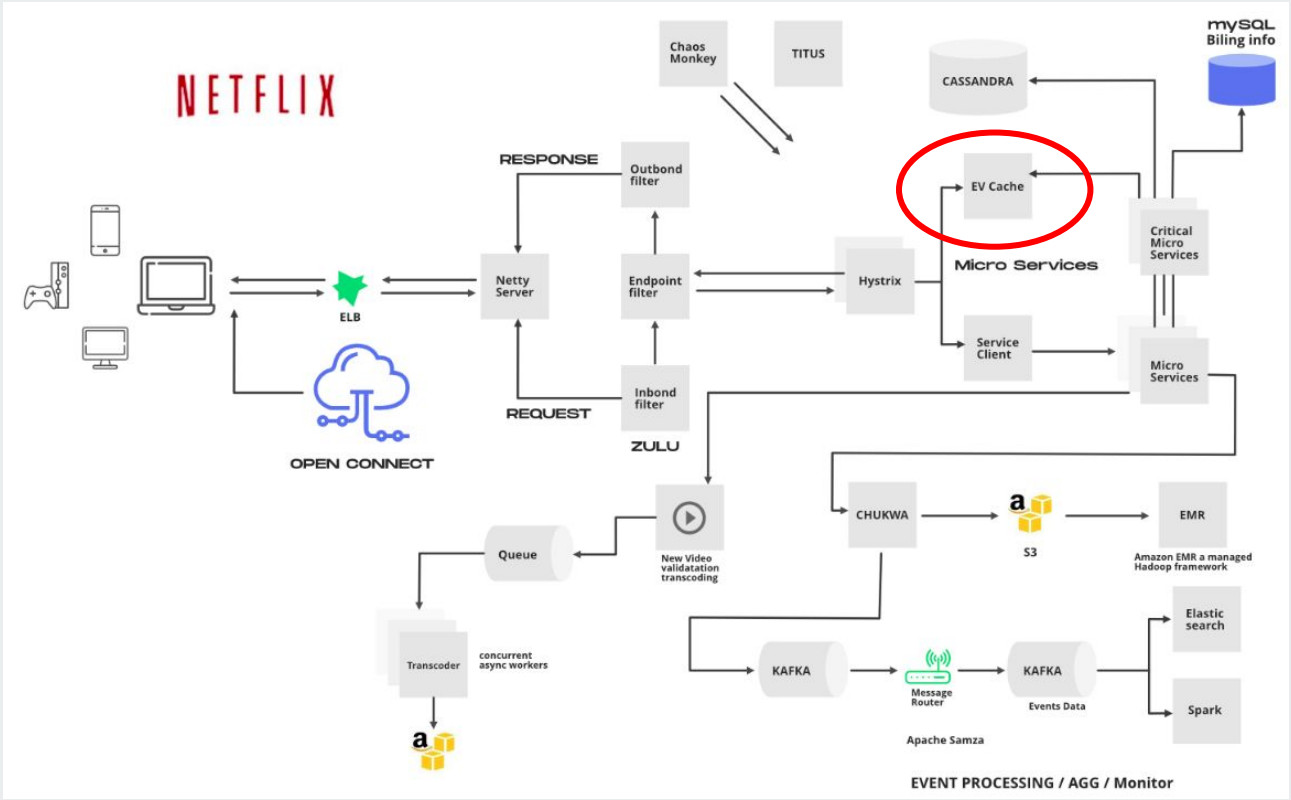
- ZUUL
- Netty
- Inbound filter
- Endpoint filter
- Outbound filter

Etude de cas : Netflix : Microservices



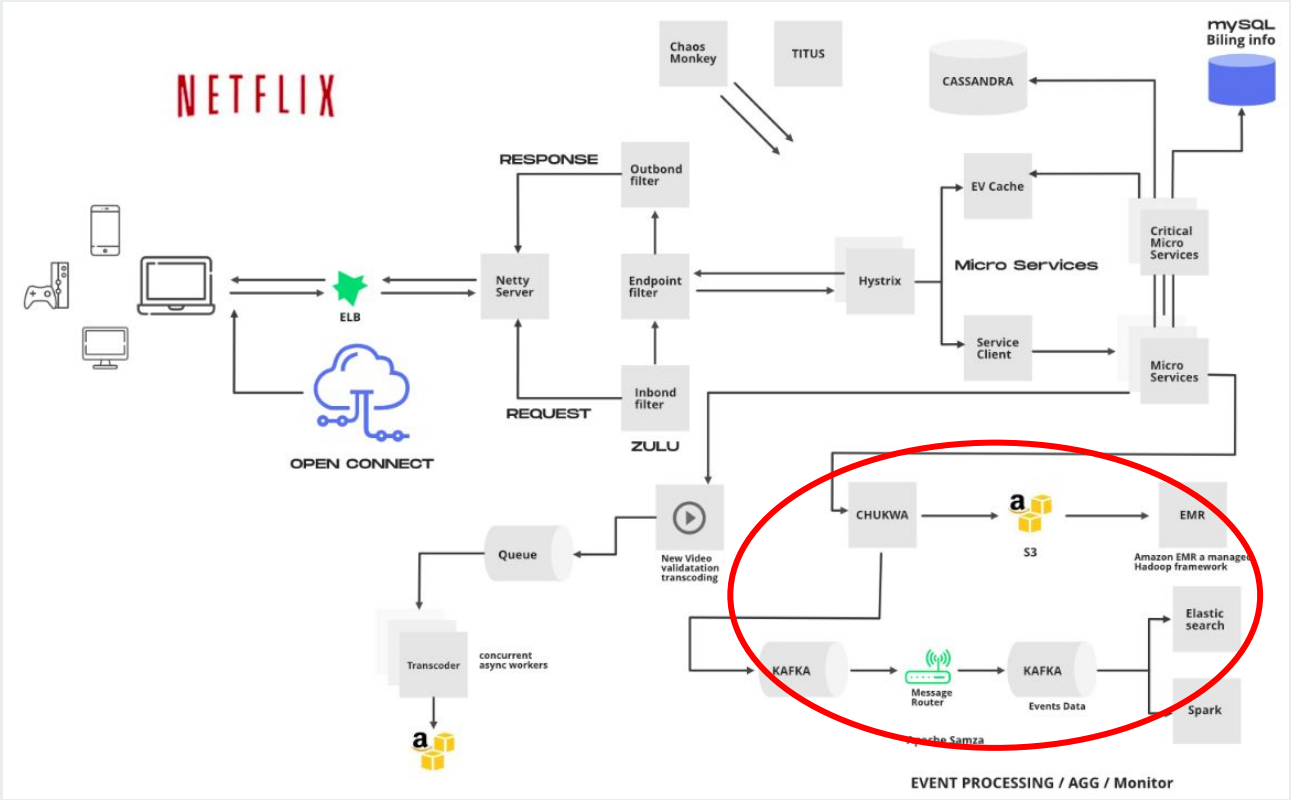
- Séparation des microservices les plus critiques (recherche et affichage vidéo) pour leur affecter une très haute disponibilité.
- Tous les services sont stateless (sans état)

Etude de cas : Netflix : EV Cache



- EV Cache : surcouche "Netflix" à Memcached
- utilisation d'instances AWS EC2

Etude de cas : Netflix : gestion des événements



Plusieurs types d'événements :

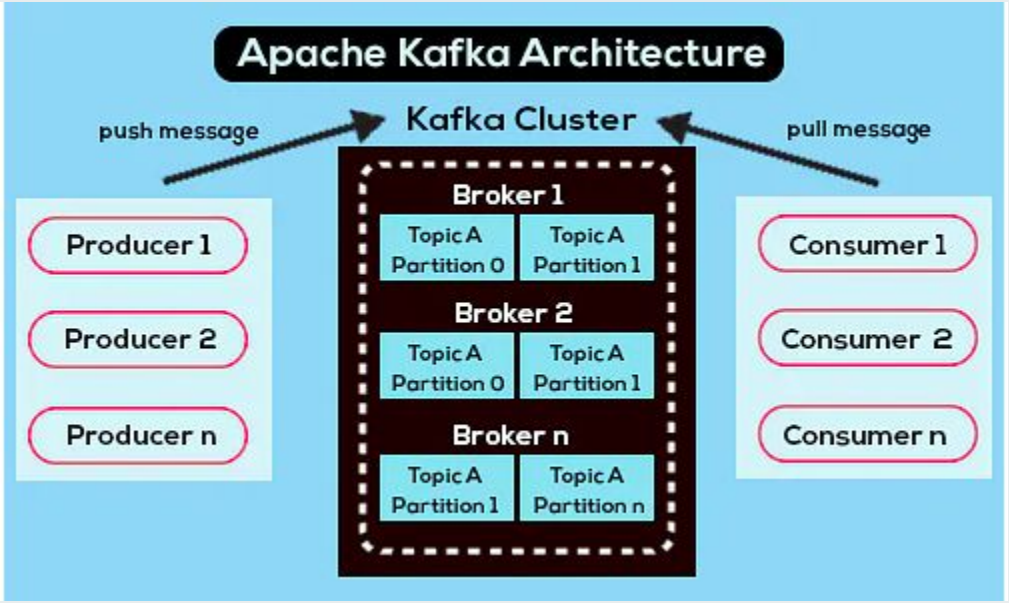
- erreurs applicatives et systèmes
- activité utilisateur
- données de performance, temps de réponse
- information de visionnage des vidéos

- 600 milliards d'événements / jour
- 1,5 Po (1500 To) en volume
- des pics de 8 millions d' événements / seconde

Etude de cas : Netflix : gestion des événements

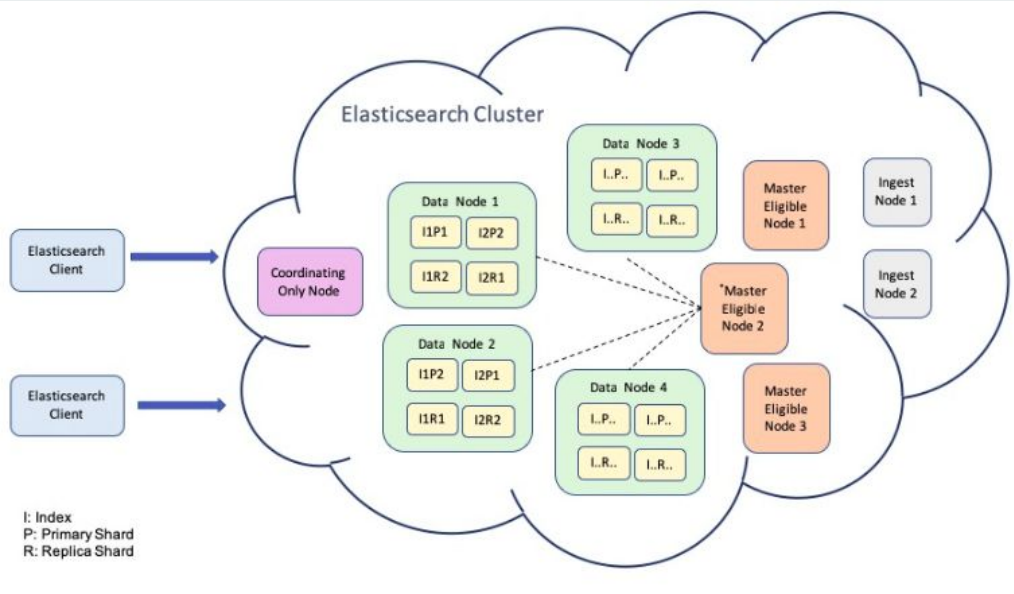
- Apache Kafka : plateforme distribuée de diffusion de données en continu, capable de publier, stocker, traiter et souscrire à des flux d'enregistrement en temps réel.
- Elasticsearch : stockage des données liés aux incidents
  150 clusters, 3500 instances
- AWS EMR : Hadoop : données d'utilisation

Etude de cas : Netflix : focus sur Apache Kafka



- Apache Kafka : plateforme distribuée de diffusion de données en continu, capable de publier, stocker, traiter et souscrire à des flux d'enregistrement en temps réel.
- Scalabilité horizontale
- Faible latence (< 10 ms)
- Très haut débit
- Tolérance aux pannes

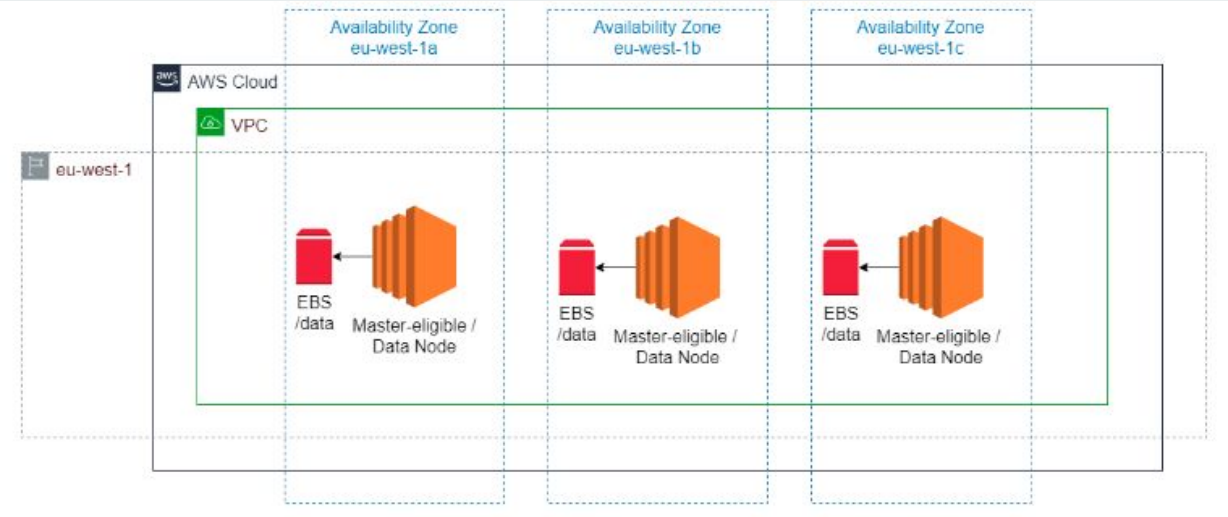
Etude de cas : Netflix : focus sur Elasticsearch



Plusieurs types de noeuds (rôles):

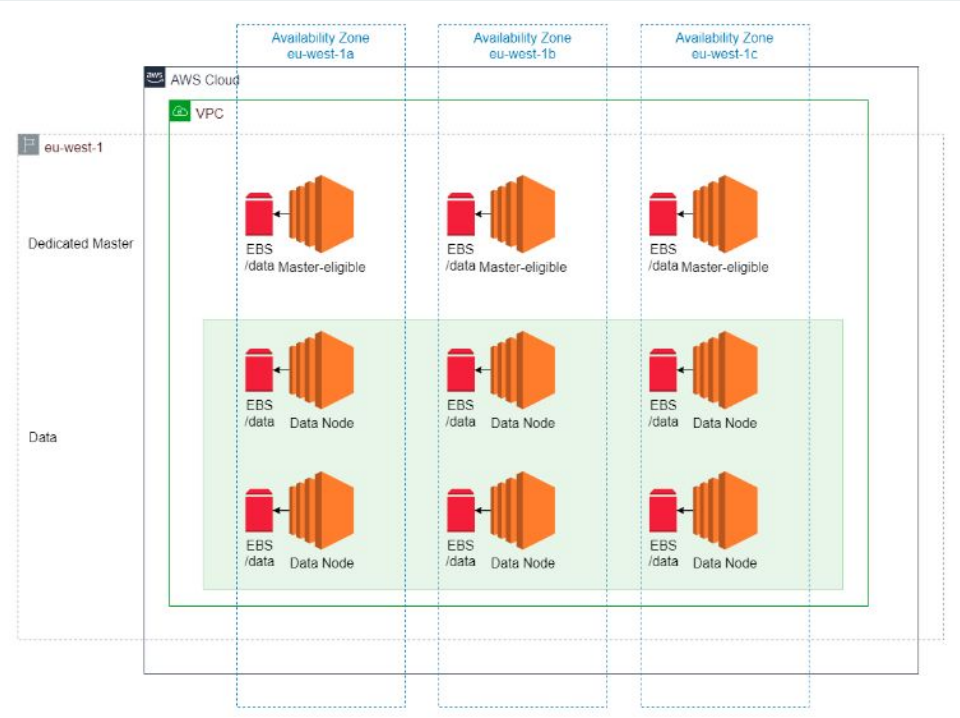
- Master Eligible : gestion du cluster, la création et la suppression d'index, l'allocation des “shards” entre les nœuds ainsi que l'ajout et la suppression des nœuds du cluster.
- Data : contient les “shards”
- Coordinating Only : routage des requêtes, distribution de l’indexation de masse
- Ingest : traitement pré-indexation

Etude de cas : Netflix : focus sur Elasticsearch



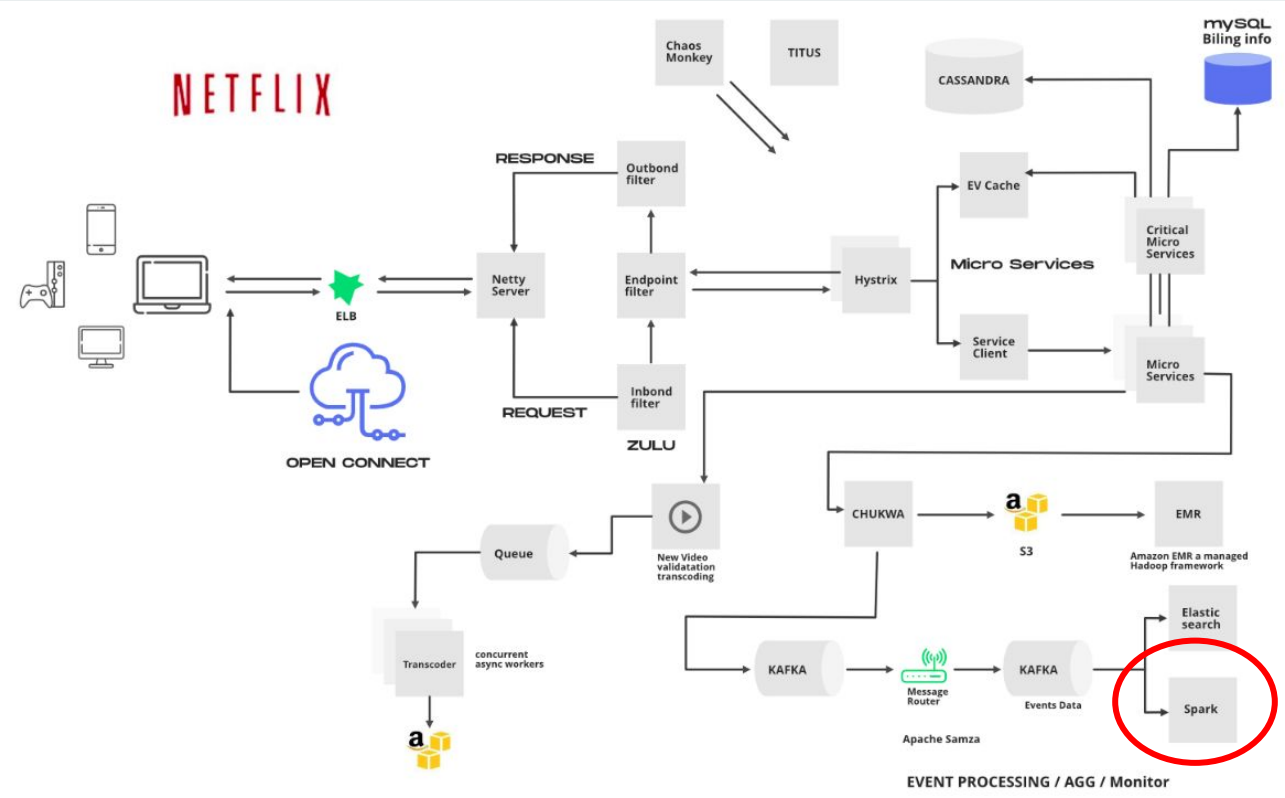
- 1 noeud peut avoir plusieurs rôles
  Ex : data et master-eligible dans le cas où la volumétrie des données n'imposent pas de noeud master-eligible dédié

Etude de cas : Netflix : focus sur Elasticsearch



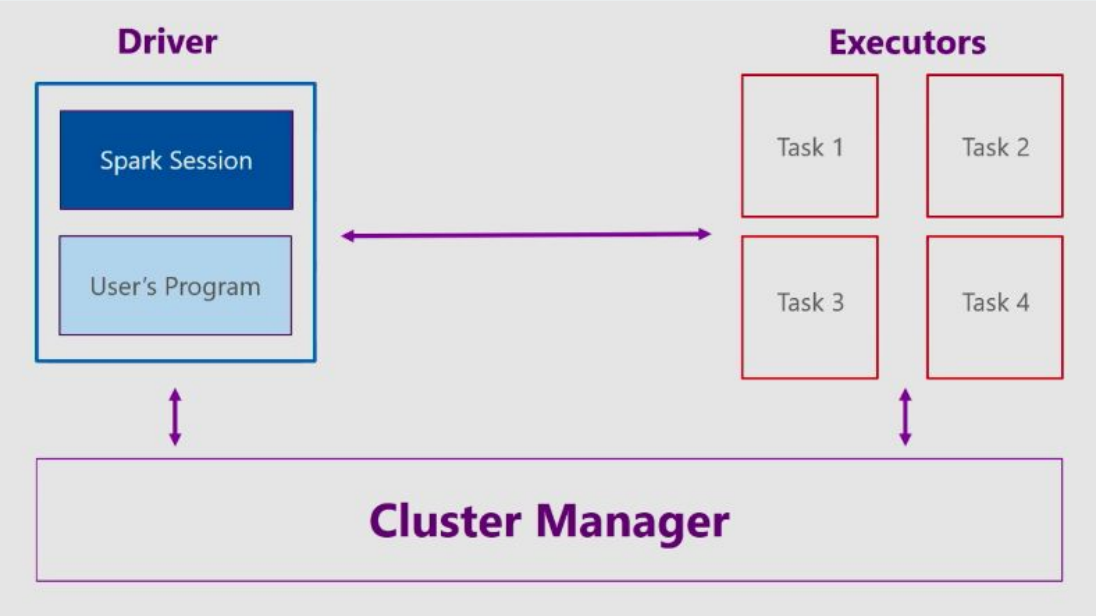
- noeuds master-eligible dédié dans le cas où la volumétrie des données augmente.

Etude de cas : Netflix : personnalisation / recommandation



- Machine Learning
- Apache Spark : Spark est un framework open source de calcul distribué

Etude de cas : Netflix : focus sur Apache Spark



Apache Spark est un moteur d'analyse unifié conçu pour le traitement des données à grande échelle avec des modules intégrés pour SQL, le traitement par flux, le machine learning et le traitement graphique.
Spark peut s'appliquer à des sources de données diverses.

Son point fort : le calcul en mémoire

Langages : Scala, Python, Java, SQL, R ,C#, F#

Etude de cas : Netflix : personnalisation / recommandation

Personnalisation

- Personnalisation des vignettes de présentation des vidéos en fonction du nombre de clics reçues par chaque vignette.
  Pour un même film, plusieurs vignettes sont “mises en compétition”.
  Celles qui offrent les meilleures performances (c’est-à-dire celles qui provoquent le plus grand nombre de visionnages de vidéos) sont conservées

Etude de cas : Netflix : personnalisation / recommandation

Recommandation

- plusieurs critères pris en compte :
  - historique de l'utilisateur
  - activités des autres utilisateurs ayant les mêmes préférences
  - métadonnées des vidéos déjà vues
  - appareil utilisé

- deux algorithmes :
  - "Collaborative filtering" : L'idée de ce filtrage est que si deux utilisateurs ont un historique d'évaluation similaire, ils se comporteront de la même manière à l'avenir.
  - "Content-based filtering" : L'idée est de filtrer les vidéos qui sont similaires à celles qu'un utilisateur a déjà aimées. Le filtrage basé sur le contenu dépend fortement des informations provenant des produits tels que le titre du film, l'année de sortie, les acteurs, le genre.

Quiz 6

https://docs.google.com/forms/d/e/1FAIpQLScAKHSMLR-S6bIHp6SRopMMM80gqnzmNKZU3vHgyzFD-2SfPg/viewform?usp=dialog