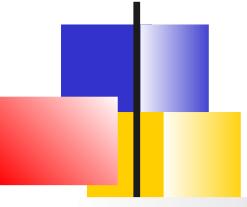


Istio : Maillage de Services sur Kubernetes

David THIBAU - 2024

david.thibau@gmail.com

Référence : **Istio in Action**, *Christian E. Posta, Rinor Maloku*
2022 Manning Publications



Agenda

Introduction

- Architectures micro-services
- Services transverses
- Convictions d'Istio

Démarrer avec Istio

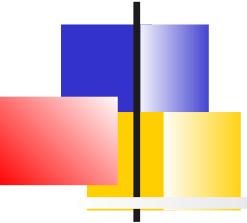
- Concepts
- Composants d'observabilité
- Installation
- Le mode ambient
- Le proxy Envoy

Ressources Istio

- Gateways
- Routing
- Résilience
- Observabilité
- Sécurité

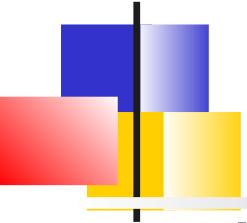
Compléments

- Installation personnalisée
- Résolution de problèmes
- Monitoring de istod



Introduction

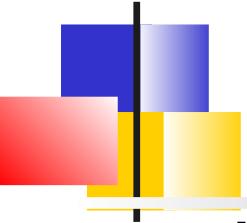
Architectures micro-services
Services transverses
Convictions d'Istio



Introduction

Le terme « **micro-services** » décrit un pattern architectural visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel

C'est le même objectif que les méthodes agiles ou les approches *DevOps* :
« *Déployer plus souvent* »

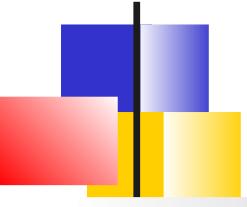


Architecture

L'architecture implique la décomposition des applications en très petit services

- faiblement couplés
- ayant une seule responsabilité métier
- Développés par des équipes full-stack indépendantes.

Le but étant de livrer et maintenir des systèmes complexes avec rapidité et qualité



Bénéfices attendus

Scaling indépendant : Seuls les services les plus sollicités sont scalés
=> Économie de ressources

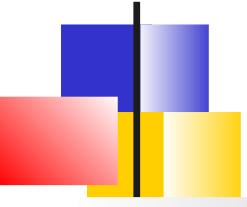
Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes
=> Agilité de déploiement

Maintenance facilitée : Le code d'un micro-service est limité à une seule fonctionnalité
=> Corrections, évolutions plus rapide

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Un dysfonctionnement peut être plus facilement localiser et isoler.

Communication inter-équipe renforcée : Full-stack team
=> Favorise le CD des applications complexes



Exigences

RéPLICATION : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte automatique : Les services sont typiquement distribués dans l'environnement d'exécution, le scaling peut être automatisé.

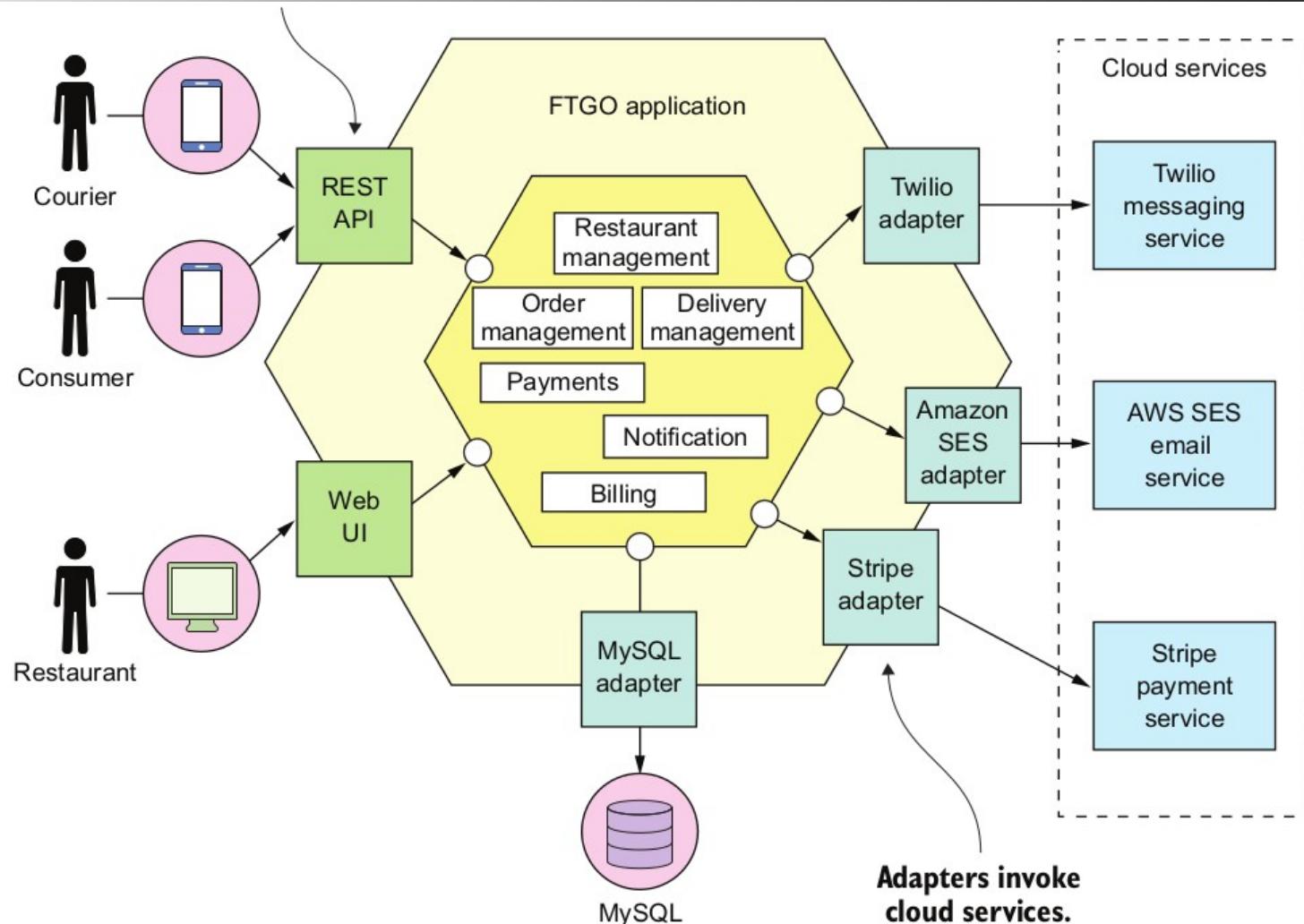
Les points d'accès doivent pouvoir être localisés automatiquement

Monitoring : Les points de surveillances sont distribués. Les traces et les métriques doivent être agrégés en un point central

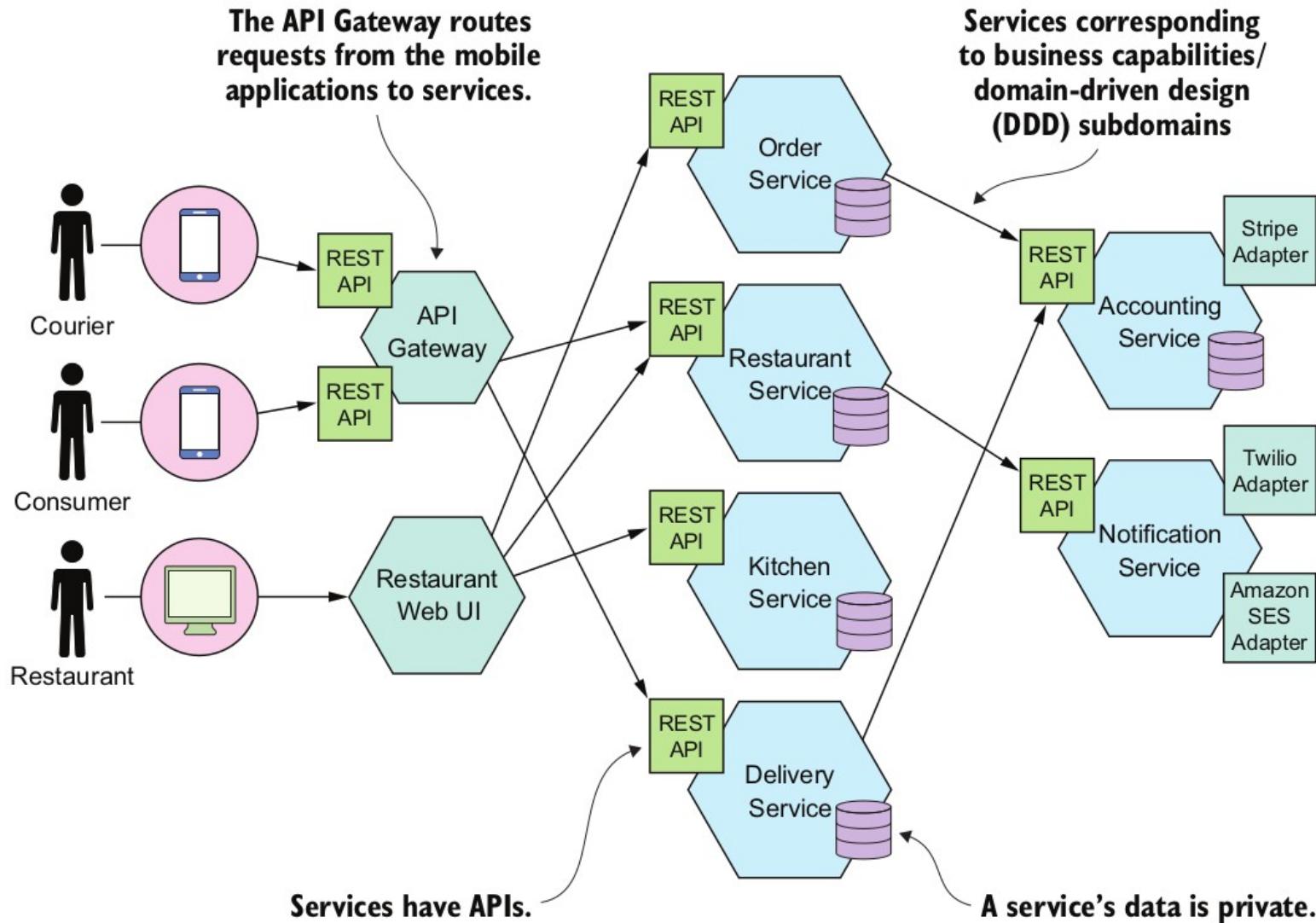
Résilience : Plus de services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.

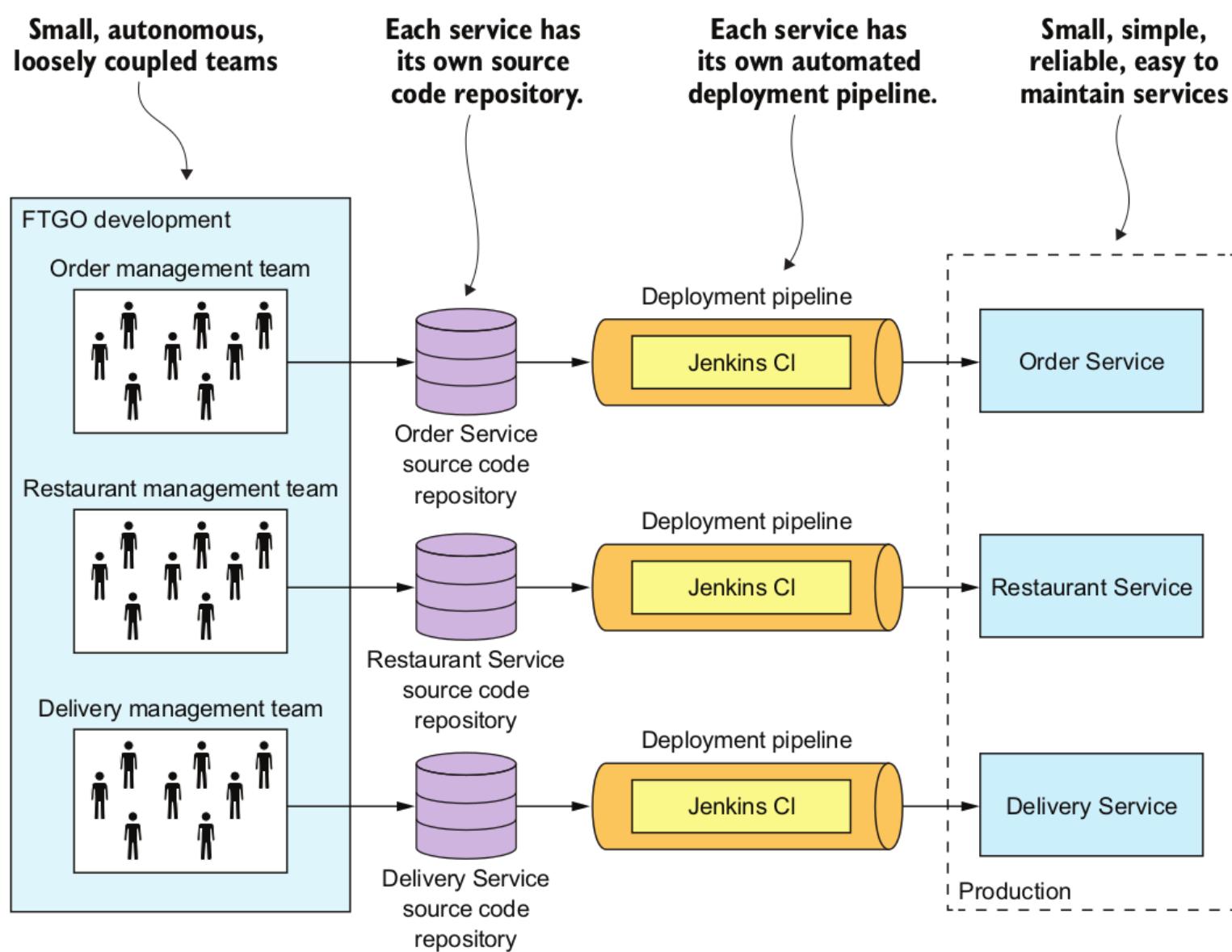
Architecture monolithique Hexagonale

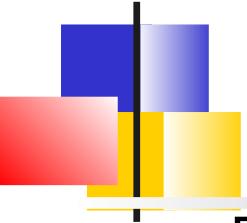


Une architecture micro-service



Organisation DevOps





Problèmes à résoudre et design patterns

Décomposition en services, Patterns :

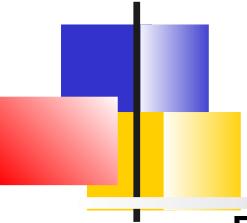
- DDD ou sous-domaines
- Fonctionnalités métier

Communication entre service, Aspects et patterns:

- Style (RPC, Asynchrone, etc.)
- Découverte des services, (Self-registry pattern, ...)
- Résilience: Retry, timeout, Circuit Breaker Pattern
- Messagerie transactionnelle : Rest + message en 1 transaction
- APIs évolutives

Distribution des données, Aspects et Patterns

- Gestion des transactions ? Saga Pattern
- Requêtes avec jointures ? CQRS



Patterns et problèmes à résoudre

Déploiement des services, Patterns :

- Un container par service, Déploiements immuables, Orchestration de Containers
- Serverless

Observabilité afin de fournir des *insights* applicatifs :

- Health check API, Agrégation des traces, Tracing distribué, Détection d'exceptions, Métriques applicatifs, Audit

Tests automatisés :

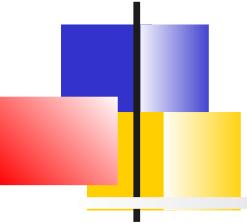
- Service en isolation, Tests des contrats (APIs)

Patterns transverses :

- Externalisation des configurations, Pipelines CD, ...

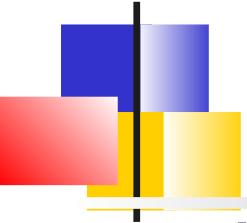
Sécurité :

- TLS, mTLS, Jetons d'accès JWT, ACLs ...



Introduction

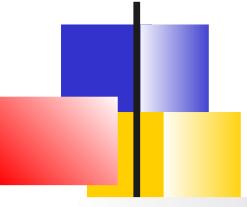
Architectures micro-services
Services transverses
L'approche Istio



Services Transverses

Les architectures distribuées doivent s'appuyer sur des services transverses.

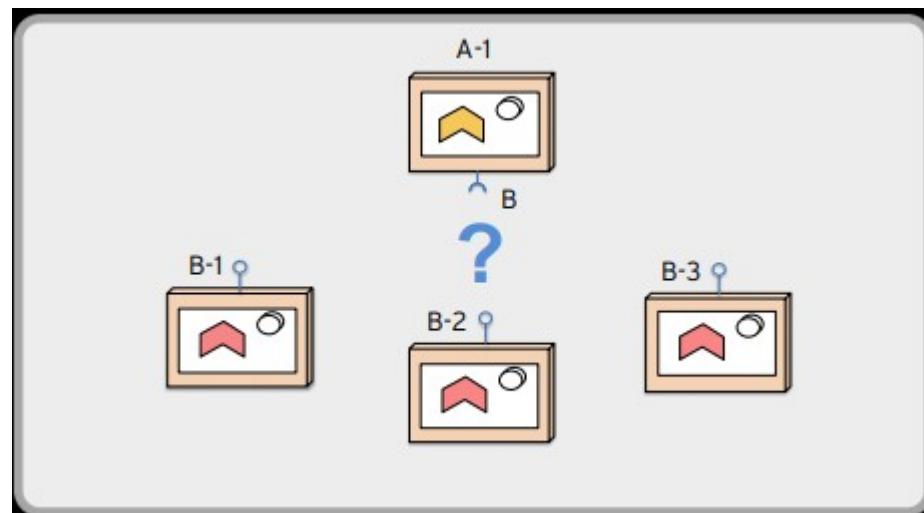
L'implémentation pouvant être un framework ou l'infrastructure

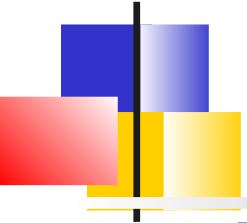


Discovery

La réPLICATION nécessite un service de découverte :

- Où sont les services
- Quelles répliques appeler (Load Balancing)

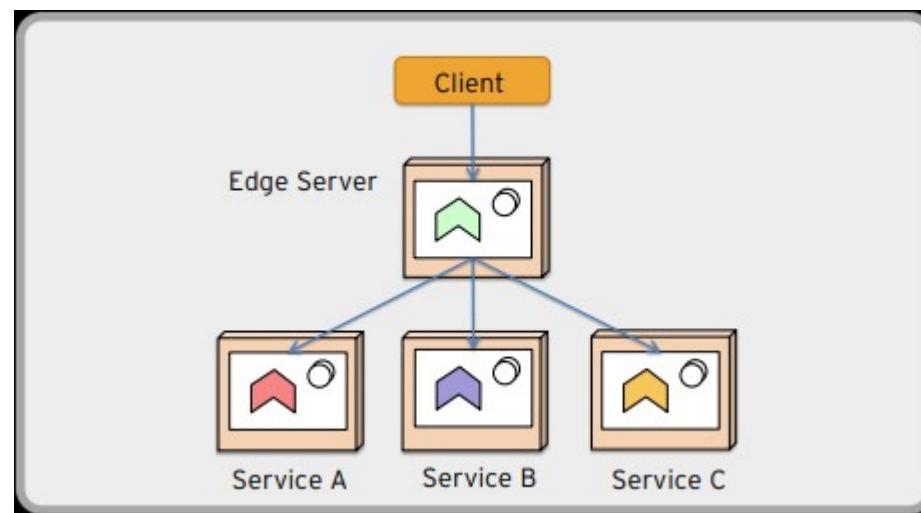


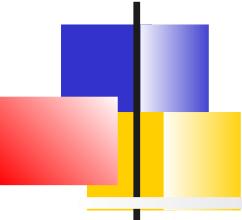


Edge server

Définir les frontières du cluster

- Comment cacher les services privés ?
- Comment exposer et protéger les services publics?

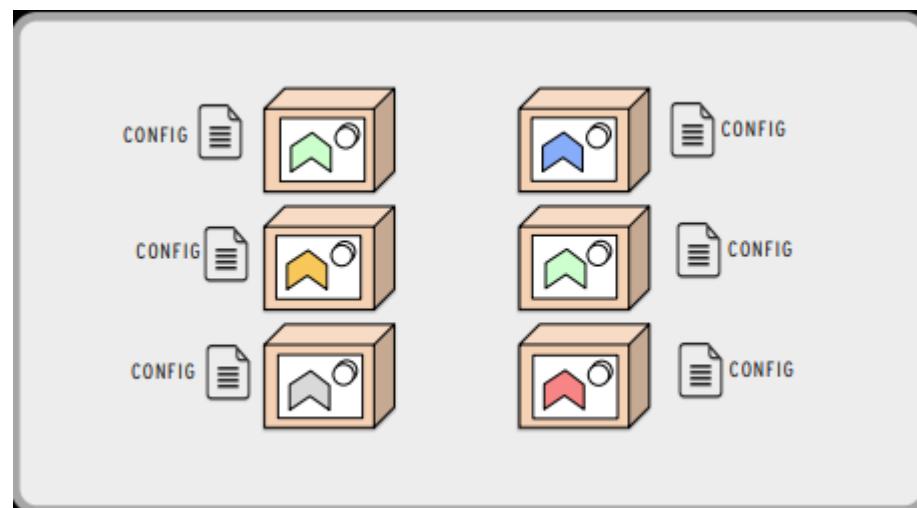




Configuration centralisée

Configuration centralisée

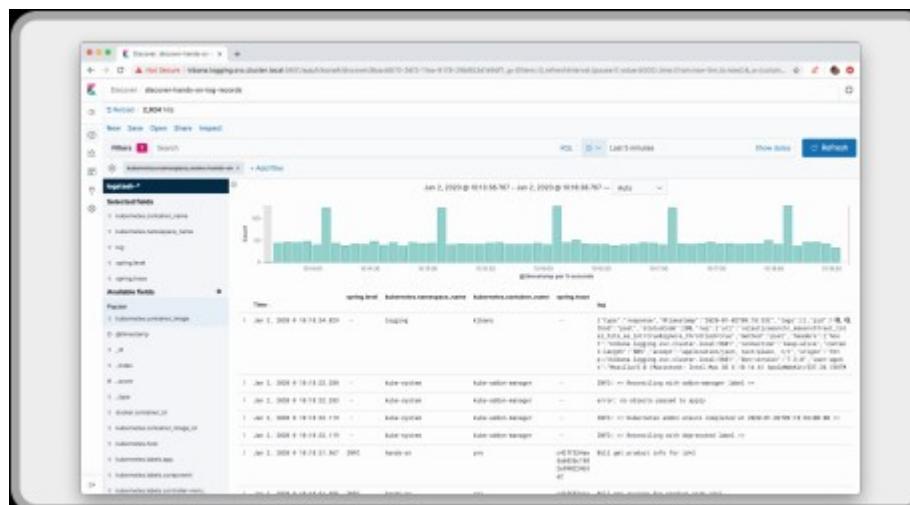
- Où se trouve la configuration
- La configuration des services est-elle à jour ?
- Comment la mettre à jour

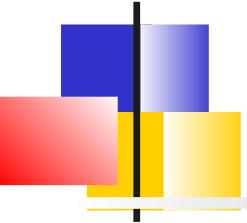


Analyse des traces

Les services génèrent des traces

- Ou réside t'elles ?
- Comment corrélérer les traces des différents services ?

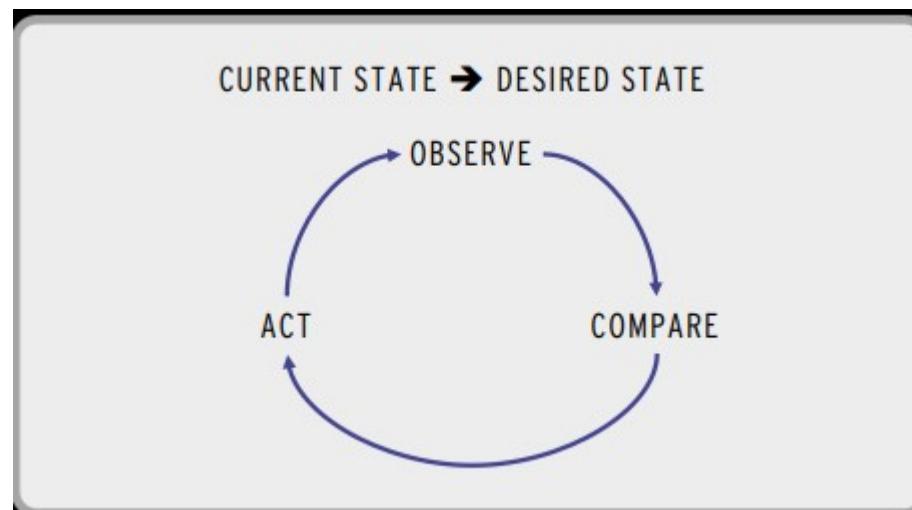


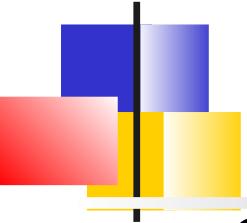


Gestion des services

Comment :

- Déployer les service
- Les scaler
- Les mettre à jour
- Redémarrer les services défaillants

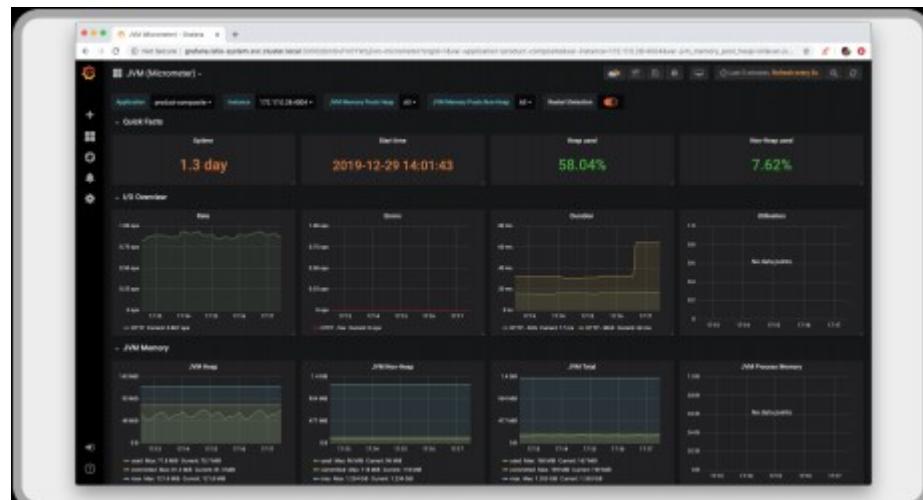


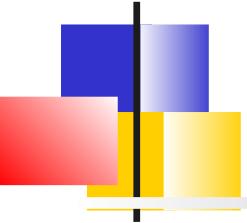


Observabilité

Observer pour comprendre, résoudre les problèmes, optimiser

- Déetecter les défaillances
- Quelles ressources sont utilisées ?
- Quel est l'usage des service ?
- Les services sont-ils performant

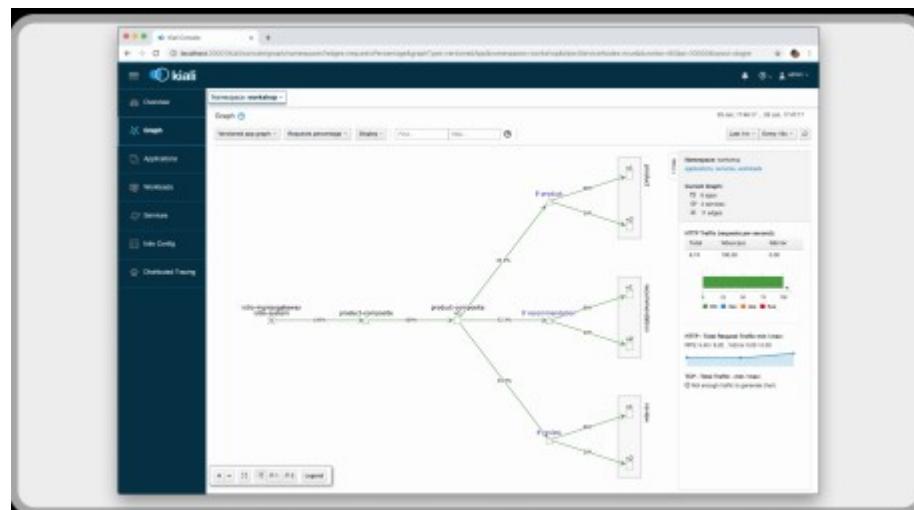




Gestion de trafic

Comment contrôler le routing ?

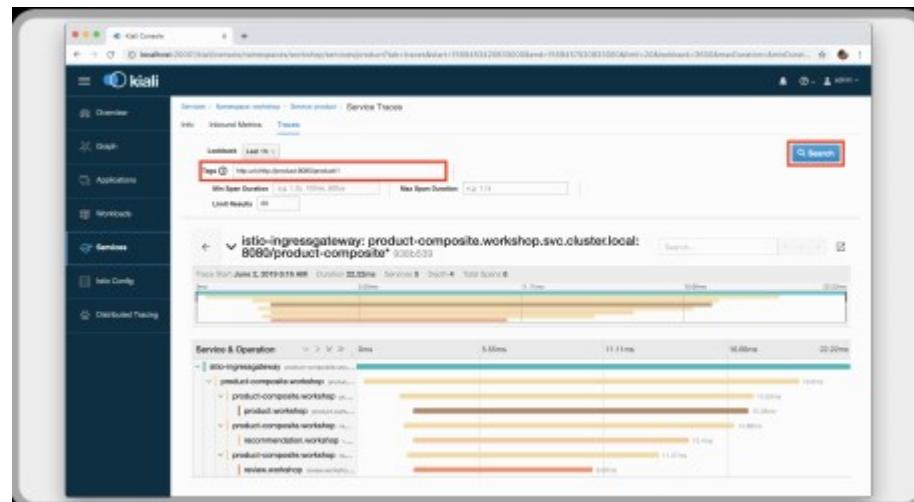
- Répartir la charge
- Limiter pour protéger
- Déploiement Blue/Green ou Canary

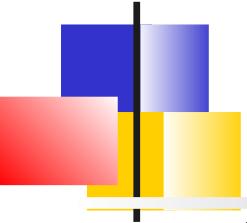


Tracing distribué

Qui appelle qui ?

– Suivre une requête dans l'architecture

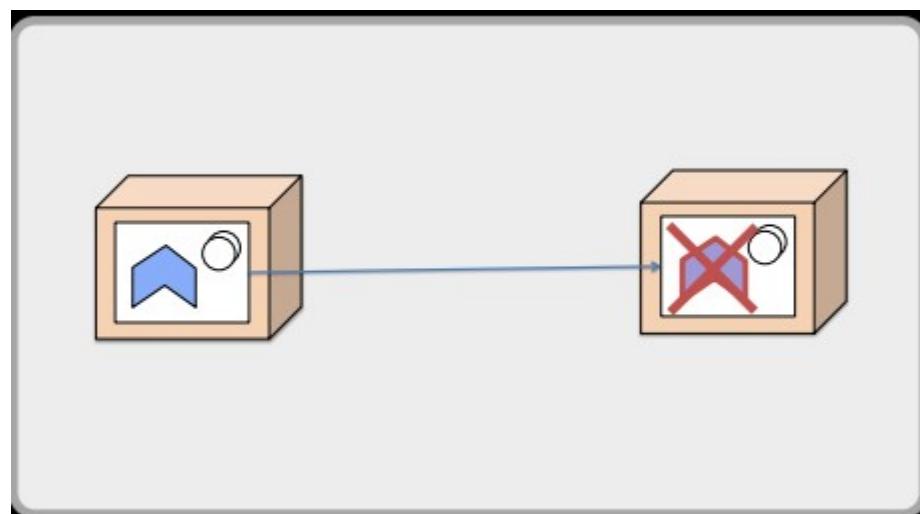


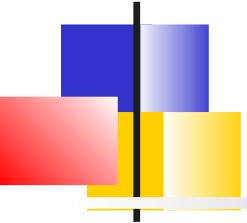


Résilience

Comment gérer les défaillances inévitables ?

- Pas de réponse ou réponse lente
- Défaillance temporaire
- Surcharge



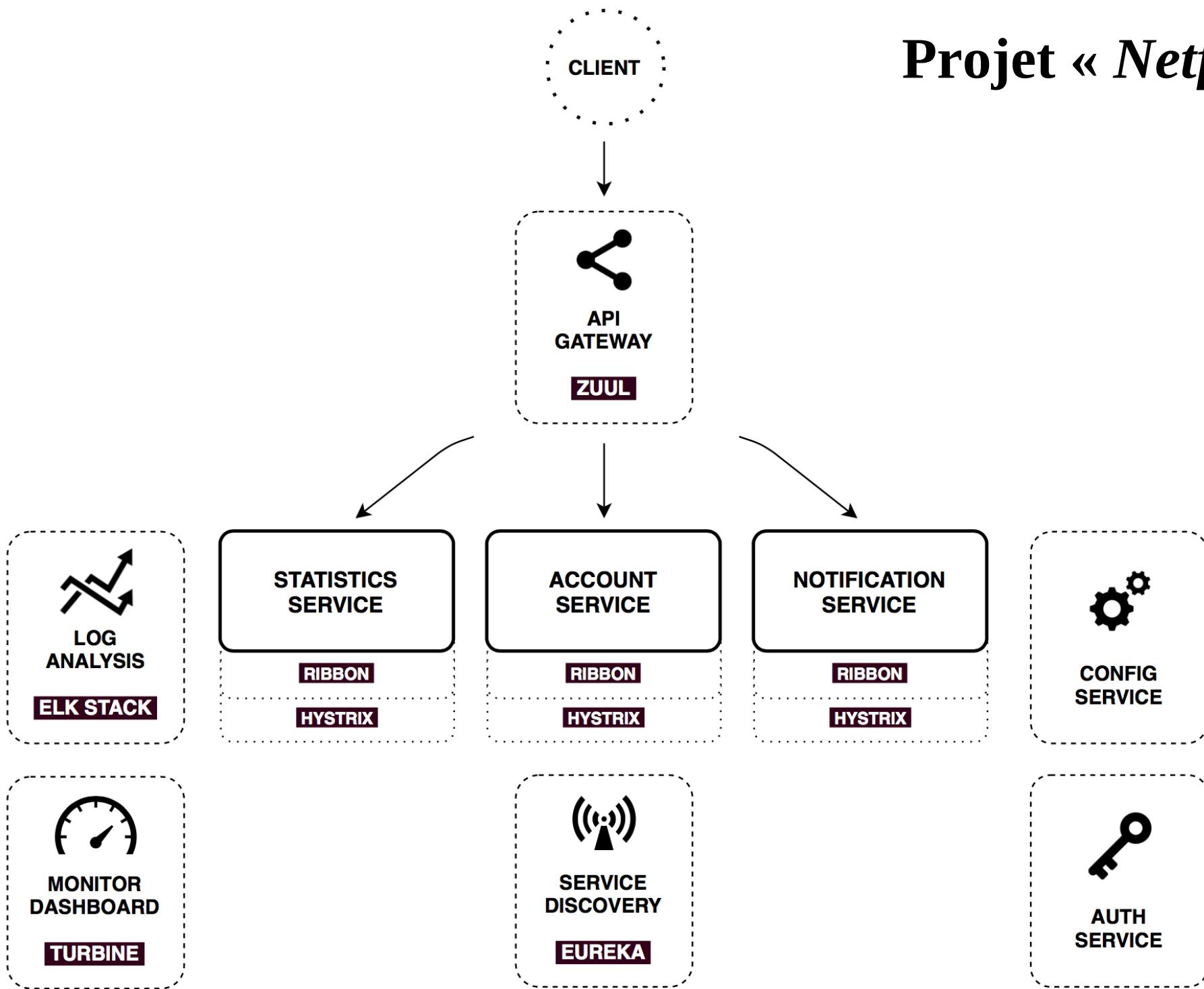
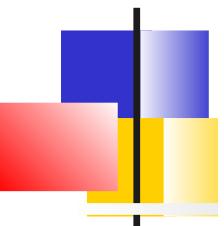


Services techniques vs Infra

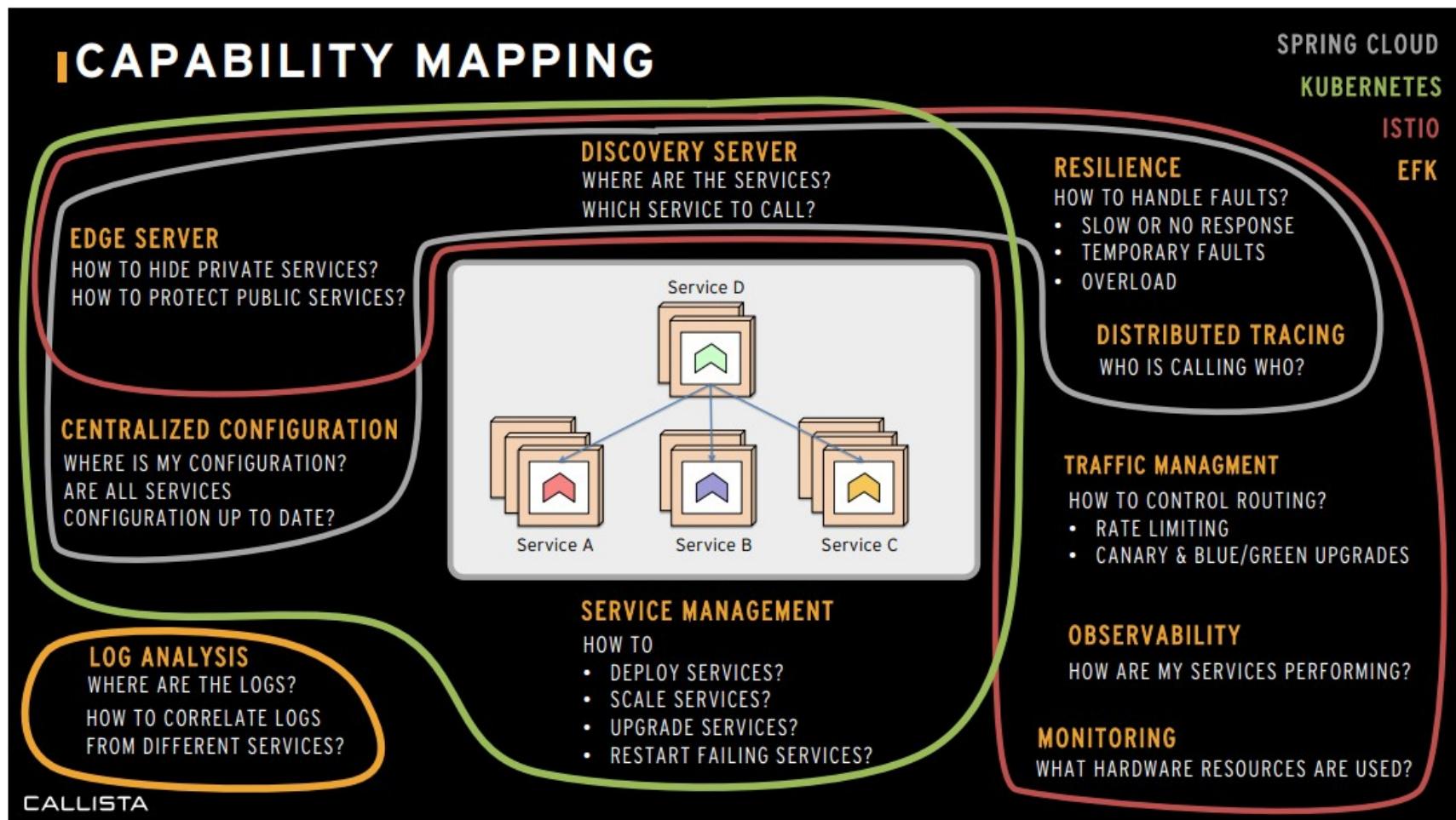
Qui fournit les services techniques ?

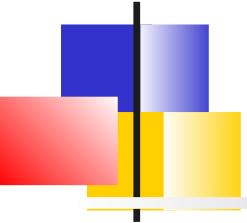
- Dans les premières architectures, c'est le software => Exemple framework Netflix proposé par Spring Cloud
- Actuellement, de nombreux services techniques migrent vers l'infrastructure :
 - Discovery, Config, Répartition de charge offert nativement par Kubernetes
 - Résilience, Sécurité, Monitoring : Service mesh de type Istio

Projet « Netflix »



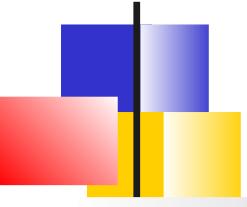
Capability Mapping





Introduction

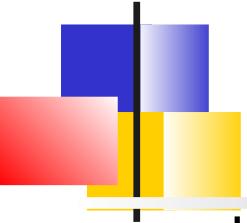
Architectures micro-services
Services transverses
L'approche Istio



Les convictions

Le réseau n'est pas fiable et pour construire des systèmes fortement distribués, le réseau doit devenir une considération de conception centrale.

La résilience, la sécurité et la collecte de métriques sont des préoccupations transverses et non spécifiques à une application



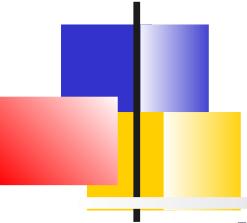
Agnostique de la pile technologique

Les solutions s'appuyant sur des librairies ou frameworks ont de nombreux inconvénients

- L'introduction d'un nouveau service dans l'architecture est contraints par les décisions d'implémentation effectués par des équipes externes au projet
- La dépendance envers une pile technologique pousse les équipes à trouver des alternatives (quelquefois partielles) qui demandent de l'investigation et de la montée en compétence et introduit de l'hétérogénéité

Les problèmes de mise en réseau d'applications ne sont pas spécifiques à une application, un langage ou un framework particulier

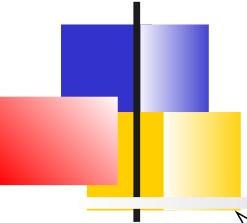
=> un moyen indépendant de la technologie pour mettre en œuvre ces préoccupations et éviter aux applications d'avoir à le faire elles-mêmes.



Proxy

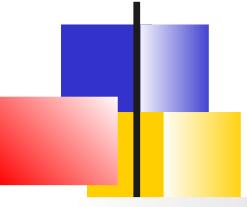
L'utilisation d'un *proxy* est un moyen de déplacer ces préoccupations horizontales dans l'infrastructure.

Un proxy est un composant d'infrastructure intermédiaire qui peut gérer les connexions et les rediriger vers les backends appropriés.



Service mesh

- Le maillage de services décrit une infrastructure qui permet aux applications d'être sécurisées, résilientes, observables et contrôlables.
- Le plan de données (Data plane), i.e règles de routage, télémétrie, sécurisation d'Istio, est présent dans des proxys de service, basés sur le **proxy Envoy**, qui cohabitent avec les applications.
La configuration des proxys est envoyée par le plan de contrôle, **le démon istiod**
- Istio est destiné aux microservices ou aux architectures de type architecture orientée services (SOA), mais il ne se limite pas à ceux-ci.

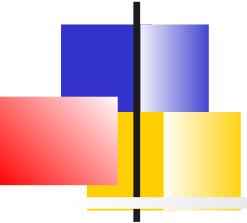


Layer 7

Le besoin : Un proxy sensible aux applications et capable d'effectuer les connexions réseau pour les services applicatifs en comprenant les protocoles utilisés (HTTP).

- Il doit comprendre les messages, les requêtes, les réponses à la différence des proxies d'infrastructure qui travaillent au niveau connexions et paquets

En clair, un proxy adapté à la couche 7 du réseau



Approche en couche L4 et L7

L7 Processing
Layer

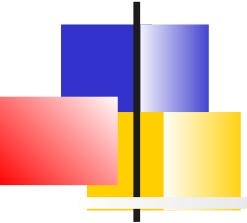
All features of the Secure Overlay plus...

- **Traffic Mgmt:** HTTP routing & load balancing, Circuit breaking, Rate limiting, Fault injection, Retry, Timeouts, ...
- **Security:** Rich authorization policies
- **Observability:** HTTP metrics, Access Logging, Tracing

Secure
Overlay
Layer

Streamlined, low resource, high performance with zero trust

- **Traffic Mgmt:** TCP Routing
- **Security:** mTLS tunneling, Simple authorization policies
- **Observability:** TCP metrics & logging



Démarrer avec Istio

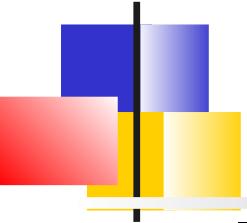
Concepts

Composants d'observabilité

Installation

Mode ambient

Proxy Envoy

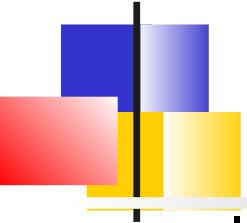


Introduction

Istio est une implémentation Open source d'un service mesh.

Créé initialement à Lyft, Google et IBM

Désormais communauté élargie à Lyft,
Red Hat, VMWare, Solo.io, Aspen Mesh,
Salesforce et d'autres



Les 2 modes d'Istio

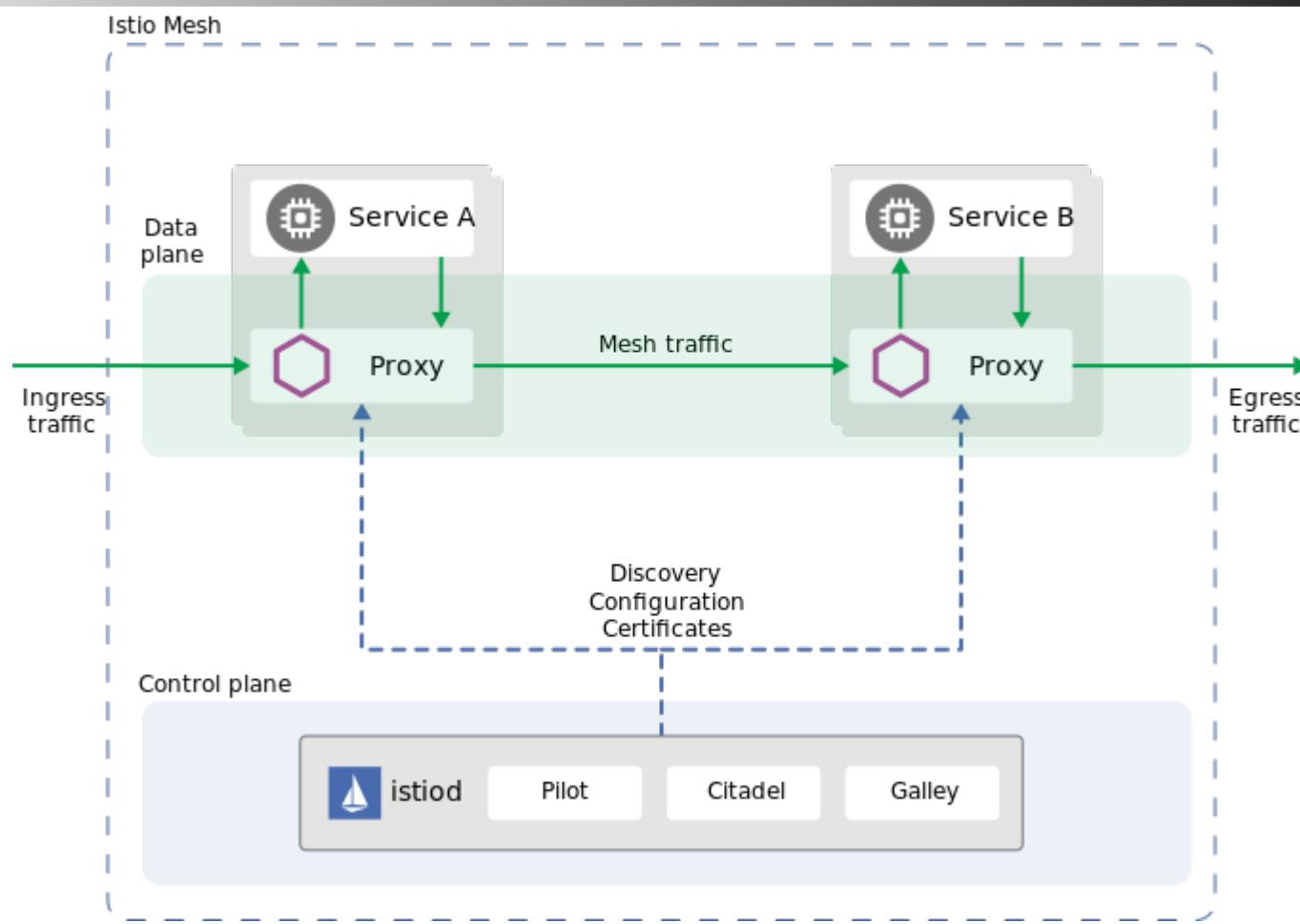
Istio est un **service mesh** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

Depuis la version 1.22, Istio support 2 modes pour gérer les communications :

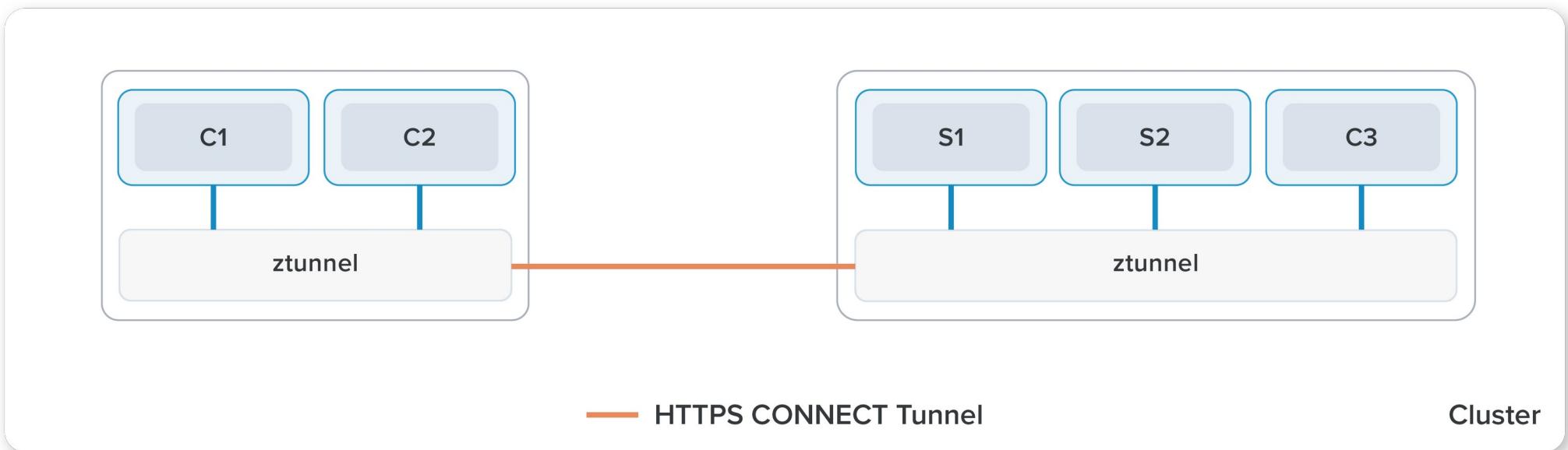
- **Sidecar mode** : Un *sidecar proxy* Layer-7 est déployé à côté de chaque micro-service et intercepte toutes les communications.
- **Ambient mode** :
 - Tout le trafic est acheminé via un proxy de nœud de couche 4
 - Si nécessaire les applications utilisent un proxy de couche 7 indépendant du service applicatif

Un tableau de contrôle gère les proxys de façon centralisée.

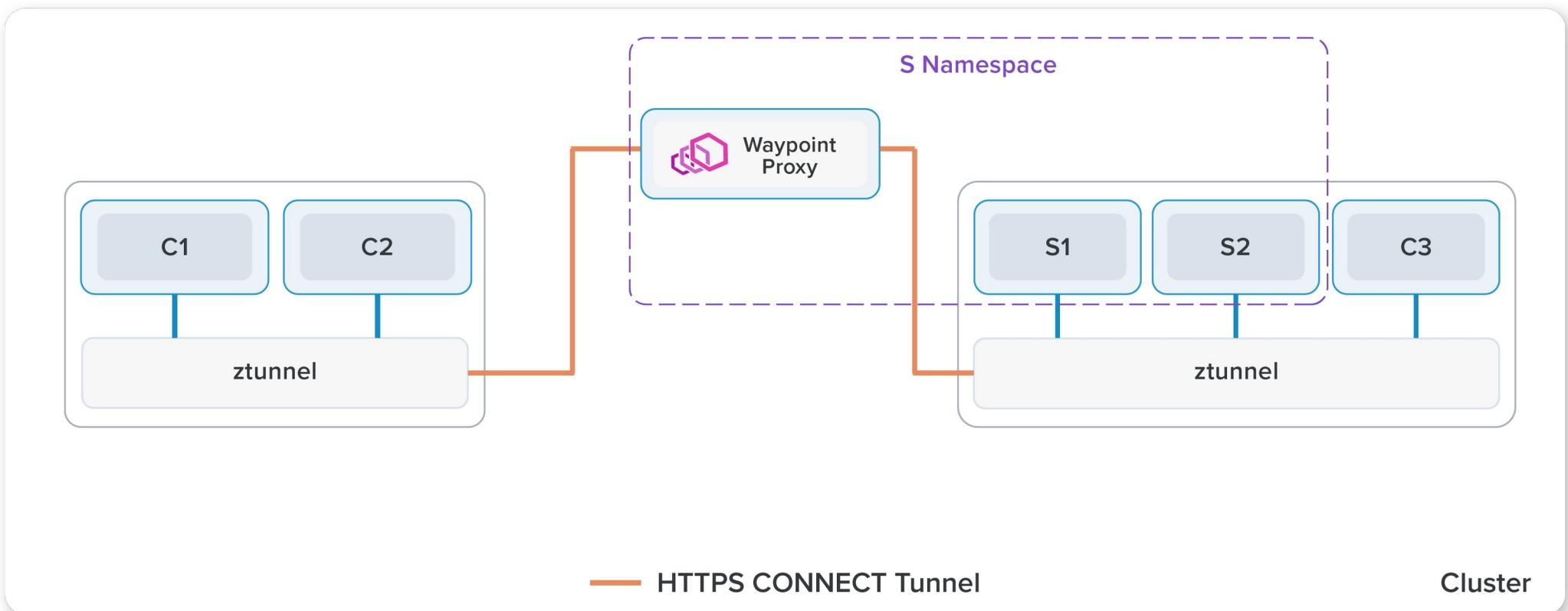
Architecture Side Car

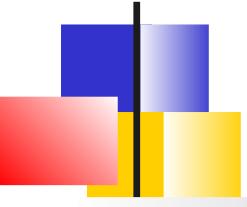


Architecture Ambient L4



Architecture Ambient L7

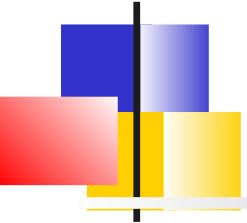




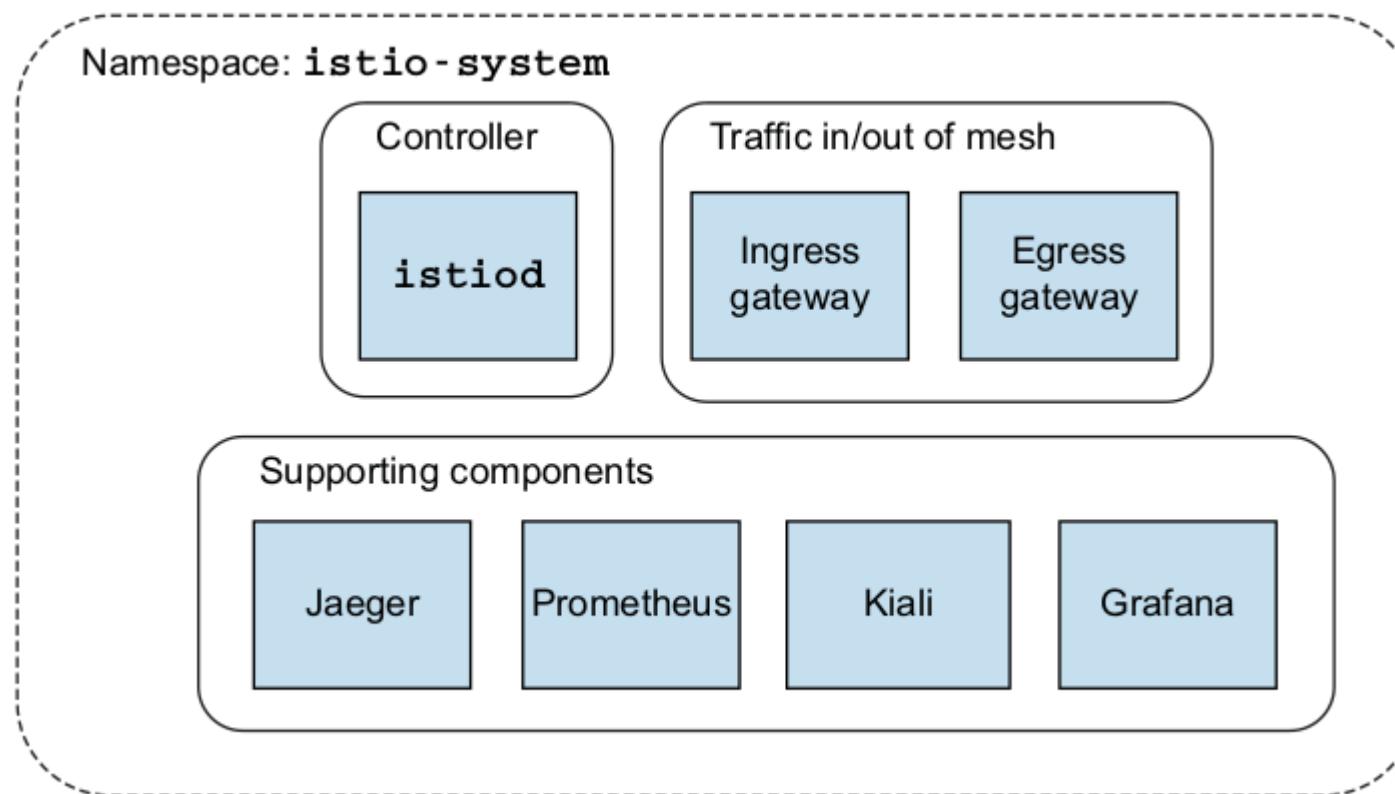
Control Plane

Le **control plane** matérialisé par *istiod* et ses composants fournit les fonctions suivantes :

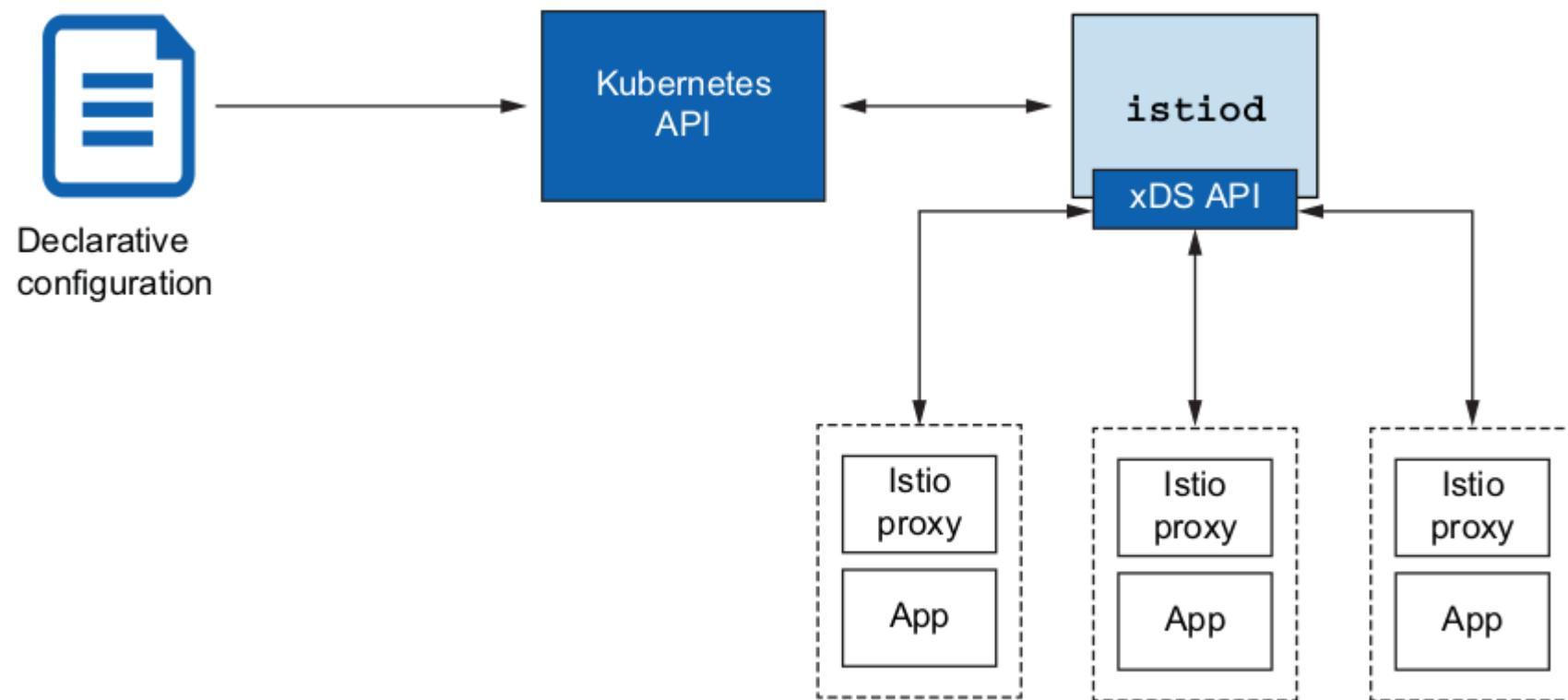
- API permettant de spécifier le comportement de routage/résilience souhaité
- API pour que les proxy Envoy consomment leur configuration
- Un service de découverte
- Identification des workloads via des certificats TLS
- Délivrance et rotation des certificats
- API pour spécifier les autorisations d'accès aux ressources
- Collecte de métriques unifiée
- Injection manuelle ou automatique des proxy
- Ouverture des frontières du réseau, ingress et egress

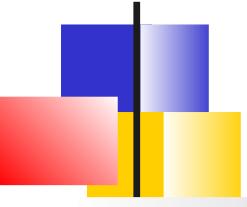


Istiod et composants supportés



Ressources Kubernetes





Ressources Istio

Gateways : Trafic extérieur au service mesh.
(Istio Gateway ou Kubernetes Gateway)

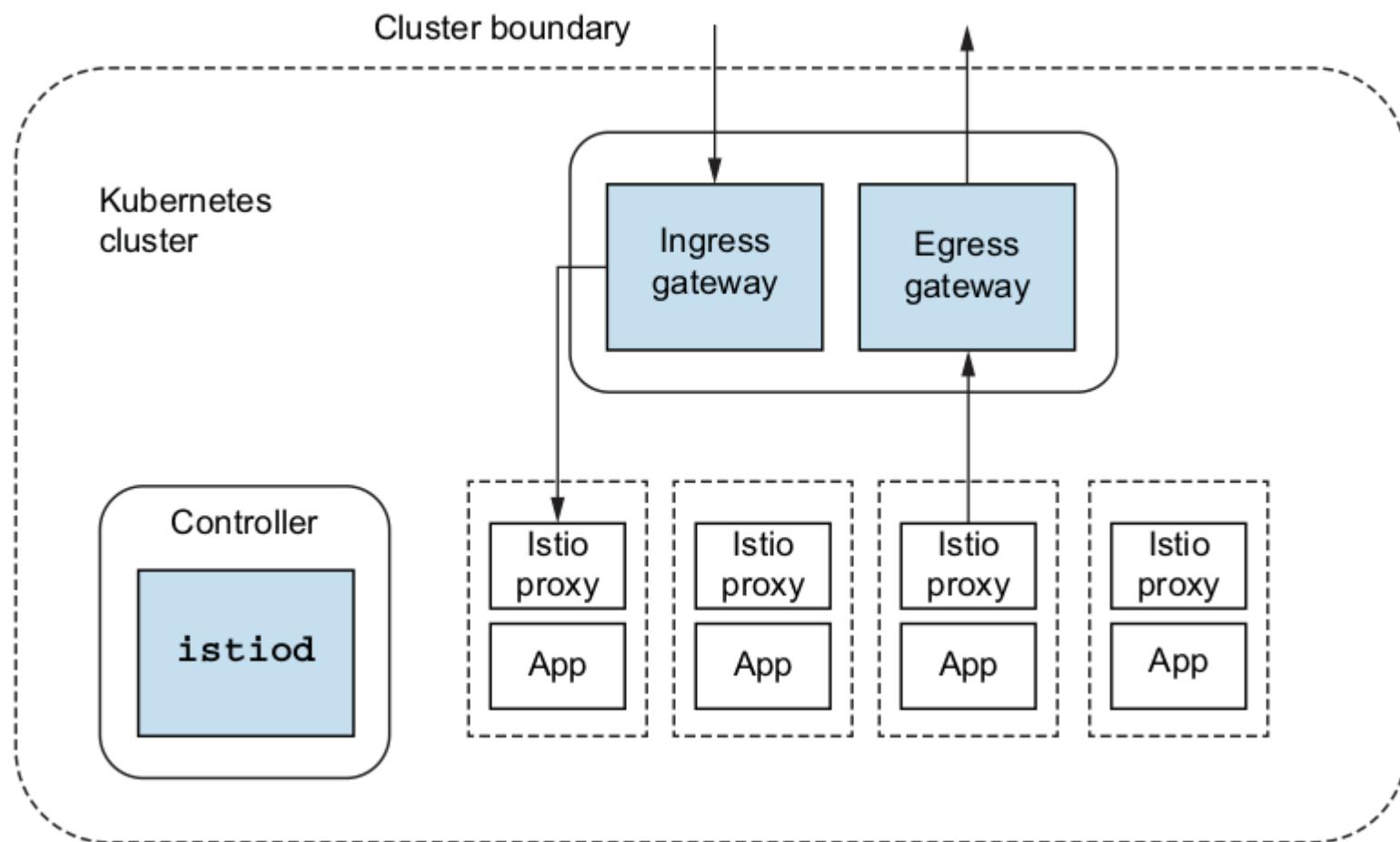
Virtual services : Permet de configurer comment les requêtes sont routées.

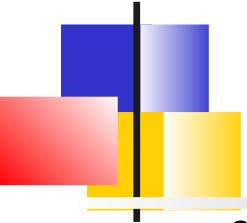
Destination rules : Stratégie de répartition de charge, Sécurisation, Circuit breaker, Définition de subset de service

Service entries : Une entrée dans le registre de service de istio

Sidecars : Configuration du proxy envoy sidecar

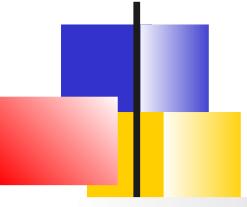
Gateways





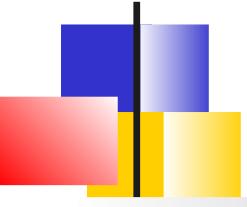
Ressource Gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: outfitters-gateway
  namespace: formation
spec:
  selector:
    istio: ingressgateway # Utilisation du contrôleur par défaut
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
    hosts:
      - "*"          # Gateway pour les hôtes virtuels
```



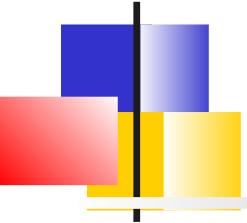
Exemple *DestinationRule*

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews-destination-rule
spec:
  host: reviews.formation.svc.cluster.local          # Service Kubernetes associé
  trafficPolicy:
    loadBalancer:
      simple: RANDOM                                # Algorithme de répartition entre sous-ensemble
  subsets:
    - name: v1                                      # Définition d'un subset
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN                      # Algorithme de répartition à l'intérieur sous-ensemble
    - name: v3
      labels:
        version: v3
```



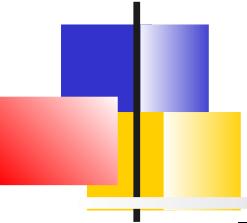
Exemple Virtual Service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:                      # Services, IP de destination
  - reviews.formation.io
  http:                         # Règles de routage
  - match:
    - headers:
        end-user:
          exact: jason
  route:
  - destination:
      host: reviews
      subset: v2                  # Un sous-ensemble du service
  - route:
    - destination:
        host: reviews
        subset: v3                  # Route par défaut
```



Démarrer avec Istio

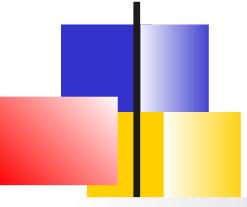
Concepts
Composants d'observabilité
Installation
Mode ambient
Proxy Envoy



Observabilité

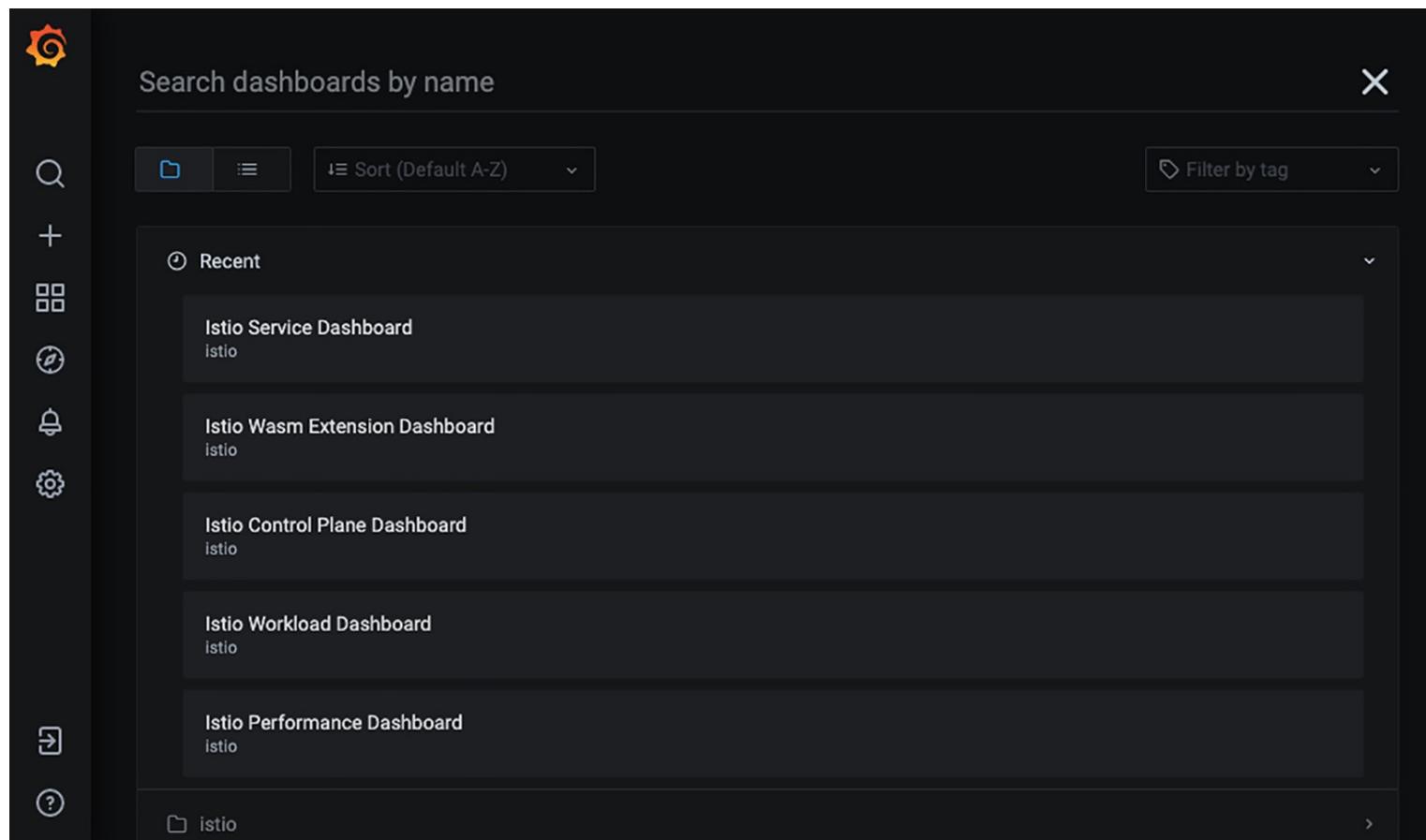
Istio crée 2 types de métriques d'observabilité :

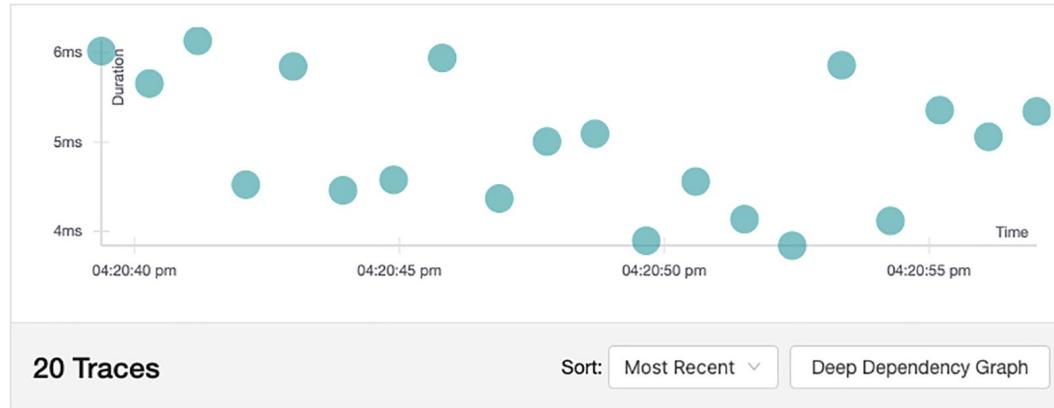
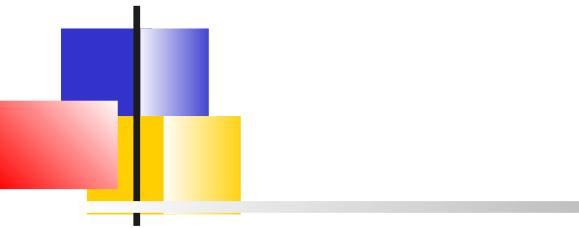
- Les métriques principales tels que le nombre de requêtes par seconde, d'échecs et les percentiles de latency.
- Le traçage distribué comme OpenTracing.io.
Istio peut envoyer des spans à des backends de tracing



Grafana

Istio a des tableaux de bord Grafana prédéfini





Compare traces by selecting result items			
<input type="checkbox"/>	istio-ingressgateway: webapp.istioinaction.svc.cluster.local:80/* de6e3fa	5.34ms	
	4 Spans	catalog.istioinaction (1) istio-ingressgateway (1) webapp.istioinaction (2)	Today 4:20:57 pm a few seconds ago
<input type="checkbox"/>	istio-ingressgateway: webapp.istioinaction.svc.cluster.local:80/* d56aead	5.06ms	
	4 Spans	catalog.istioinaction (1) istio-ingressgateway (1) webapp.istioinaction (2)	Today 4:20:56 pm a few seconds ago
<input type="checkbox"/>	istio-ingressgateway: webapp.istioinaction.svc.cluster.local:80/* feed590	5.36ms	
	4 Spans	catalog.istioinaction (1) istio-ingressgateway (1) webapp.istioinaction (2)	Today 4:20:55 pm a few seconds ago

Vue détaillée

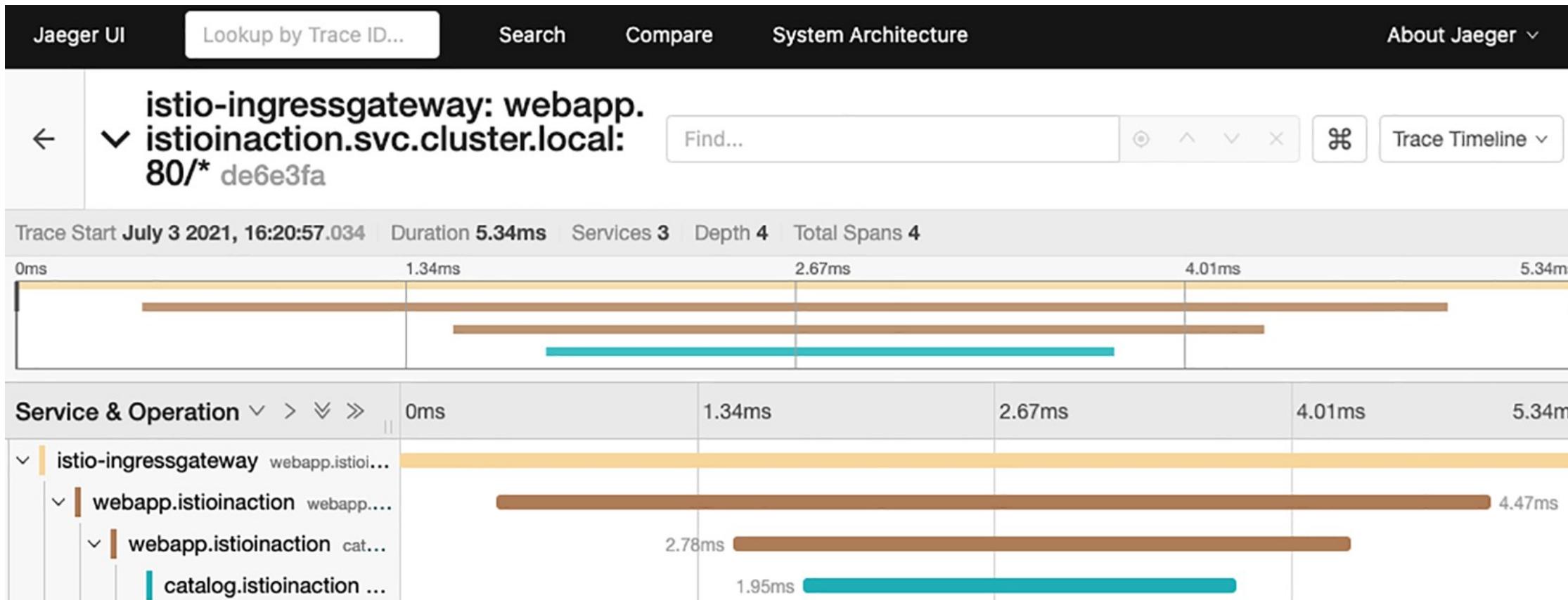


Tableau de bord Kiali

Namespace: default | Versioned app graph

Display Find... Hide... ⓘ

Last 1m | Every 15s ⏪

May 13, 5:17:28 PM ... 5:18:28 PM

istio-ingressgateway latest (istio-system)

productpage

reviews

details

ratings

v1, v2, v3

Legend: OK (green), 3xx (light blue), 4xx (red), 5xx (dark red)

NS default ✓

Current Graph:

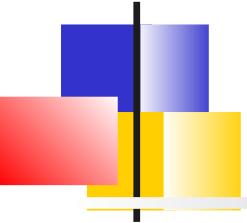
- 5 apps (7 versions)
- 4 services
- 12 edges

Incoming Outgoing Total

HTTP (requests per second):

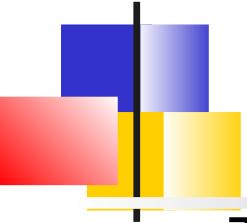
Total	%Success	%Error
0.47	100.00	0.00

OK 3xx 4xx 5xx



Démarrer avec Istio

Concepts
Composants d'observabilité
Installation
Mode ambient
Proxy Envoy



Installation

Télécharger une distribution qui contient :

- Des applications exemple **/samples**
- Un client binaire **istiocli**

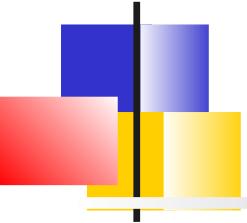
Ajouter le client au \$PATH

Exécuter une commande de déploiement en utilisant un profil d'installation (demo par exemple)

Éventuellement ajouter un label à un *namespace* pour profiter de l'injection automatique des proxy *Envoy*

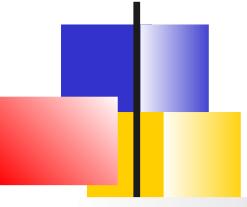
Vérifier l'installation

Éventuellement, installez les composants d'observabilité (Prometheus, Grafana, Jaeger, Kiali, etc).



Démarrer avec Istio

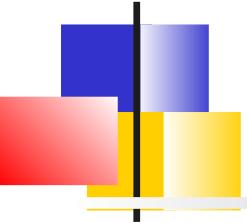
Concepts
Composants d'observabilité
Installation
Mode ambient
Proxy Envoy



Mode ambient

Dans le mode ambient, Istio implémente ses fonctionnalités à l'aide de :

- ztunnel : un proxy de couche 4 (L4) déployé sur chaque nœud du cluster
- Éventuellement Waypoint proxy : un proxy de couche 7 (L7) typiquement responsable d'un espace de noms.



ztunnel

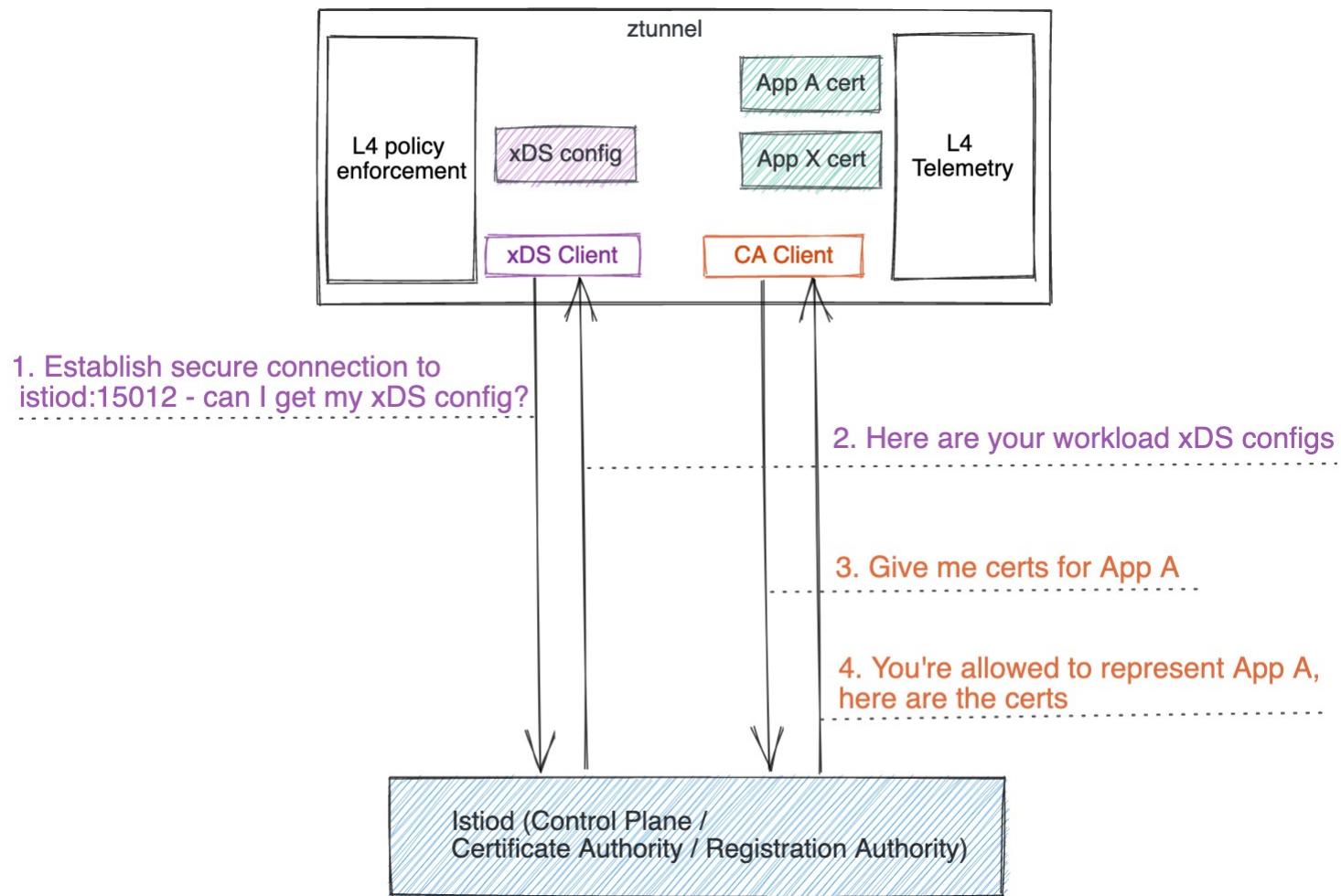
ztunnel (Zero Trust tunnel) est responsable de la connexion et de l'authentification sécurisées des workload au sein du mesh.

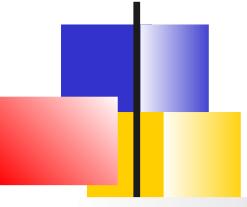
Il gère les fonctions L3 et L4 telles que mTLS, l'authentification, l'autorisation L4 et la télémétrie

Les politiques d'autorisation L4 déployées via un manifeste Istio identique au mode proxy ne peuvent se spécifier qu'en fonction de l'identité du client, du namespace ou des ips

La configuration dynamique de ztunnel utilise également xDS
Ztunnel est compatible avec Kubernetes NetworkPolicy dans le sens il ne contourne pas ses règles de sécurité

Architecture





Waypoint Proxy

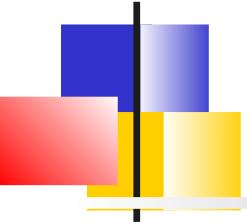
Waypoint Proxy est un déploiement du proxy Envoy

Ils s'exécutent en dehors des pods applicatifs et sont donc installés, mis à niveau et mis à l'échelle indépendamment des applications.

Ils sont utiles lorsque on veut faire de l'autorisation L7 (basé sur HTTP), de la télémétrie HTTP et l'utilisation des Virtual Host

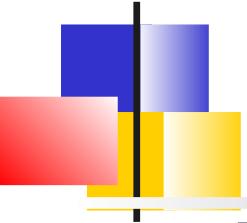
Contrairement au mode side-car, les politiques sont appliquées par le proxy de destination, agissant comme une gateway vers une ressource (un espace de noms, un service ou un pod) et appliquant ensuite toutes les politiques pour cette ressource.

Les Waypoint proxies sont déployées en utilisant des ressources Gateway de Kubernetes.



Démarrer avec Istio

Concepts
Composants d'observabilité
Installation
Mode ambient
Proxy Envoy



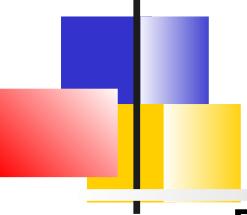
Envoy Proxy

Développé à Lyft

Puis Open Source à partir de Septembre 2016,

En Septembre 2017, rejoint le Cloud Native Computing Foundation (CNCF).

Écrit en C++



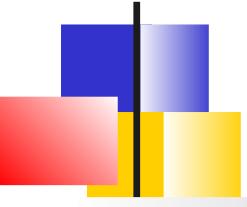
Proxy opérant au niveau protocole de la couche 7

Envoy comprend les protocoles de la couche 7 comme HTTP 1.1, HTTP 2, gRPC, et d'autres

Il peut ainsi collecter de nombreux métriques des requêtes qui le traverse comme les temps de réponse, le débit, les taux d'erreurs.

Envoy est très polyvalent et peut être utilisé dans différents rôles :

- Proxy à l'entrée du cluster (ingress endpoint)
- Proxy partagé pour un seul hôte ou groupe de services : Istio ambient
- Ou par service comme avec Istio Side Car.



Concepts Envoy

Listeners : Exposer un port au monde extérieur auquel les applications peuvent se connecter.

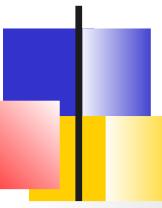
Un listener accepte du trafic et applique la configuration de ce trafic.

Routes : Règles de routage pour gérer le trafic entrant sur les listeners.

Par exemple, si une requête match `/catalog` , la router vers le cluster `catalog`.

Clusters : services en amont spécifiques vers lesquels Envoy peut acheminer du trafic.

Par exemple, `catalog-v1` et `catalog-v2` peuvent être des clusters séparés, et les routes peuvent spécifier les règles pour diriger le trafic vers v1 ou v2.



Fonctionnalités cœur d'Envoy

Découverte de Service : L'API de découverte est une API REST simple qui peut être utilisée pour encapsuler d'autres API de découverte de services courantes (comme HashiCorp Consul, Apache ZooKeeper, Netflix Eureka, etc.).

Le plan de contrôle d'Istio implémente cette API.

Répartition de charge : Envoy implémente quelques algorithmes d'équilibrage de charge : Random, Round robin, Pondéré, Moins chargé, Consistent hashing (sticky)

Routing : Comme Envoy comprend HTTP, il peut utiliser des règles compliquées basées sur les entêtes, l'URI, le contexte, les cookies, ...

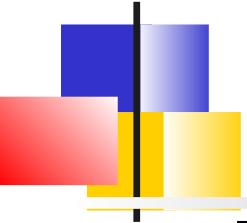
Déplacement ou masquage de trafic : Un certain pourcentage ou même une copie du trafic peut être dirigé sur un autre cluster.

Résilience au réseau : Timeouts et retry au niveau requête, limite de cadences

Dernier protocoles : HTTP/2 (multiplexage sur une seule connexion, push serveur, streaming, backpressure), gRPC (protoBuf)

Observabilité,

TLS : Envoy peut terminer une communication TLS (fournir les certificats) ou être à l'origine d'une interaction TLS

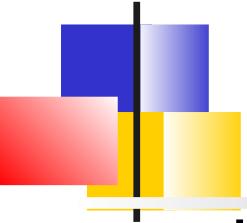


Configuration Envoy

Fichiers JSON ou YAML qui spécifient listeners, routes, et clusters ainsi que d'autres points

Envoy expose un ensemble d'APIs¹ construite sur gRPC pour effectuer des mises à jour dynamique de la configuration sans aucun temps d'arrêt ni redémarrage : listeners, routes, cluster, certificats.

1. Référencées sous le nom de *xDS services*, i.e. *Discovery Services*



Cycle de configuration xDS

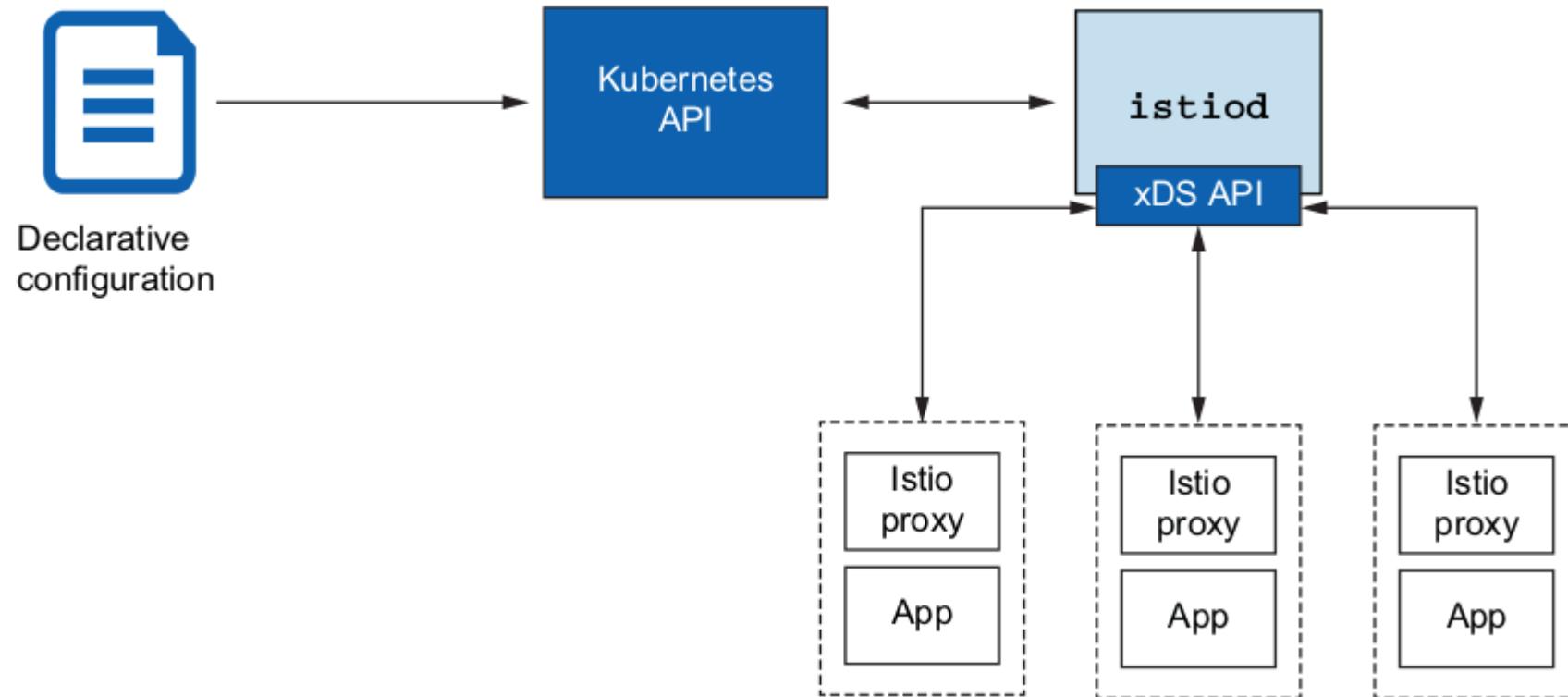
Lorsqu'un service (ou un pod) démarre,
Kubernetes notifie *Istiod*.

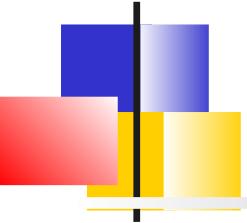
Istiod détermine la configuration réseau
nécessaire pour le proxy Envoy associé.

Istiod envoie cette configuration à Envoy via xDS.

À mesure que la topologie du réseau change
(nouveaux services, mises à jour de
configuration, etc.), Istiod envoie des mises à
jour dynamiques via xDS pour que Envoy reste à
jour en temps réel, sans interruption de service.

Envoy et Istio

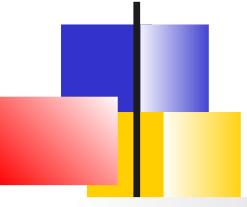




Ressources Istio

Gateway

Routing
Resilience
Observabilité
Sécurisation

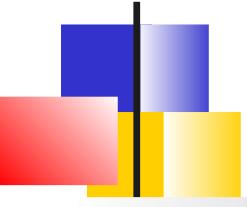


Ingress

Ingress fait référence au trafic qui provient de l'extérieur du réseau et qui est destiné à un endpoint au sein du réseau.

Le point d'entrée (ingress point) applique des règles et des stratégies concernant le trafic autorisé sur le réseau local.

- Si le trafic est autorisé, le point d'entrée forwarde sur le bon endpoint du réseau local.
- Sinon, il rejette le trafic.



Virtual IP ou Hosting

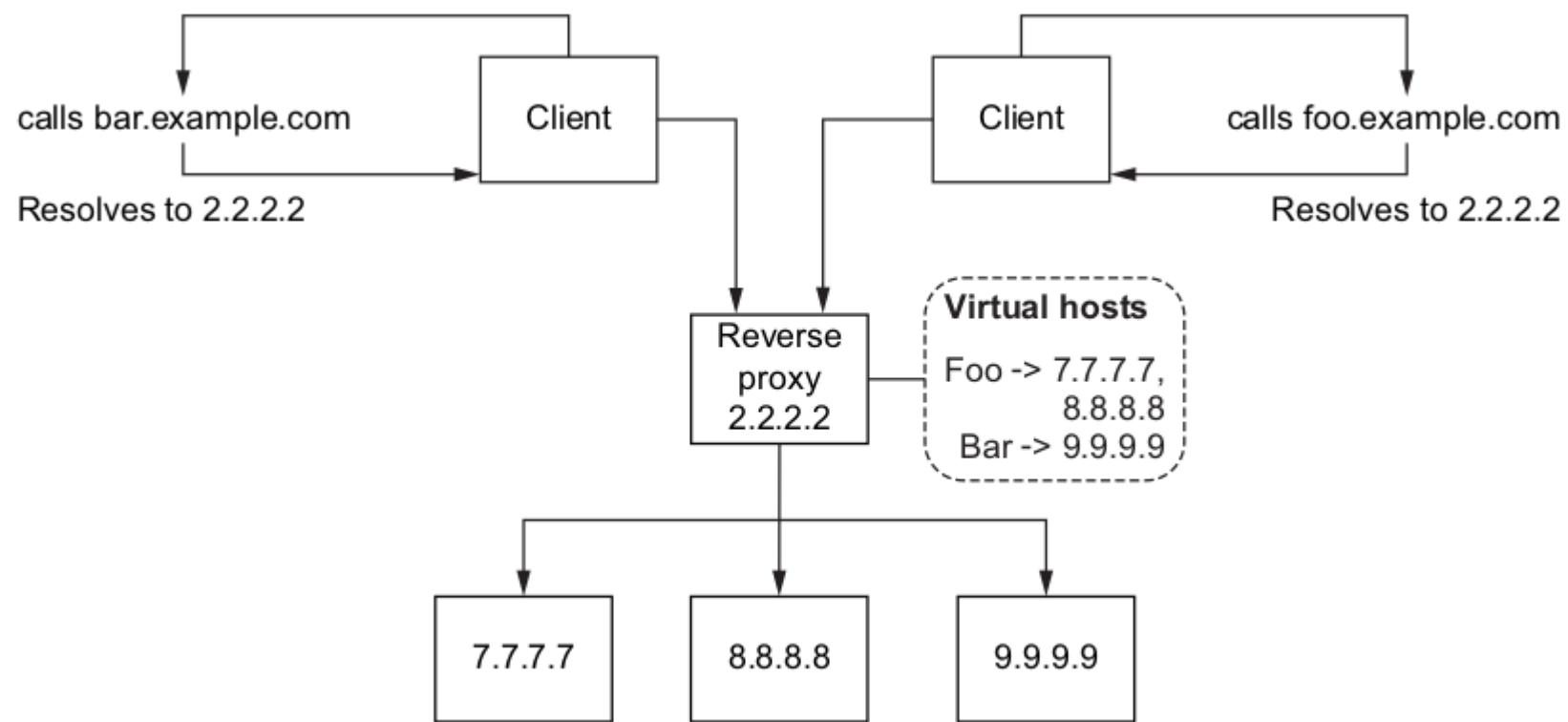
Les techniques de virtual IP ou virtual hosting permettent un routage plus robuste vers le service cible .

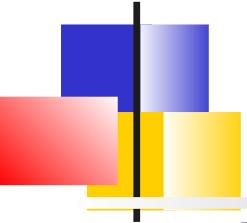
Ces techniques s'appuient sur un reverse proxy :

- Dont l'IP est déclarée au niveau DNS pour 1 ou plusieurs noms de domaine (virtual hosts)
- Qui soit capable de router vers le service interne adéquat en fonction des méta-données du protocole
(Par exemple Entêtes *Host* ou *:authority* pour Http, *SNI* pour TCP)

Ce sont exactement les caractéristiques de la gateway ingress de Istio qui est capable de faire du Load balancing (virtual IP) ou du virtual-host routing en s'appuyant sur Envoy

Virtual Hosting





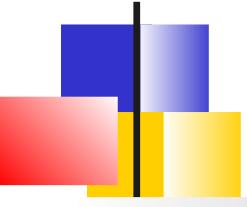
Gateways Istio

Dans ces versions récentes, Istio permet de configurer des gateways :

- Via une ressource Gateway Istio
- Ou une ressource Gateway Kubernetes¹

Les 2 techniques offrent plus de flexibilité qu'un gateway ingress

1. Nécessite que l'API gateway soit installé dans Kubernetes



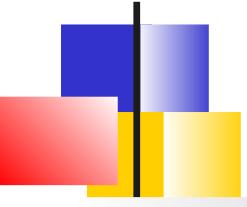
Istio-*gateway

istio-ingressgateway est un service s'exécutant dans le namespace *istio-system* qui permet de gérer les flux rentrant.

Il utilise le proxy Envoy qui peut être configuré avec 2 types de ressources Kubernetes :

- ***Gateway***
- ***VirtualService***

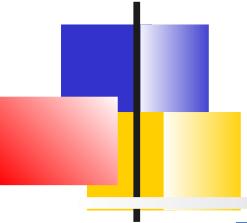
Ces mêmes ressources configurent les flux sortants : ***istio-egressgateway***



Ressource Gateway

Une ressource **Gateway** spécifie les ports ouverts et les hôtes virtuels associés à ces ports.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
  name: coolstore-gateway # le nom de la gateway
spec:
  selector:
    istio: ingressgateway # l'implémentation de la gateway
  servers:
    - port:
        Number: 80          # Le port exposé
        name: http
        protocol: HTTP
  hosts:
    - "webapp.formation.io" # Virtual host
```

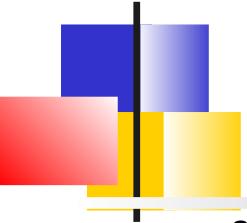


VirtualService

VirtualService est une ressource qui définit la manière dont un client communique avec un service spécifique via son nom de domaine complet.

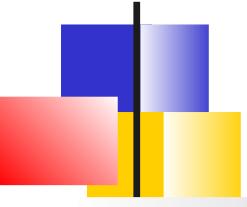
Cela comprend

- les versions d'un service disponibles
- Des propriétés de routage (comme les nombre de retry et les timeout)



Exemple

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: webapp-vs-from-gw  # Nom du service virtuel
spec:
  hosts:
    - "webapp.formation.io"  # Hôte virtuel associé
  gateways:
    - coolstore-gateway      # La gateway associé
  http:
    - route:
        - destination:      # Le service de destination
          host: webapp
        port:
          number: 8080
```

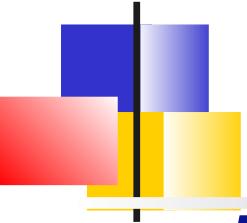


Scope d'une config *VirtualService*

La spécification d'un *VirtualService* inclut un champ **gateways** qui permet de définir quand la règle s'applique :

- Un ensemble de gateways avec désigné par :
[<gateway namespace>/]<gateway name>
- Le mot clé **mesh** si elle s'applique à tout le mesh (valeur par défaut)

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  gateways:
  - mesh      # Le virtual service s'applique à tous le mesh
```



Virtual Service Reference

hosts : Les hôtes de destination vers lesquels le trafic est envoyé

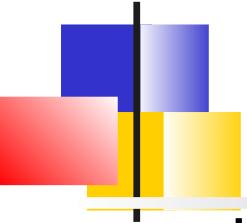
gateways : Le nom des gateways qui applique ces routes ou mesh pour tous les sidecar

http : Une liste ordonnée de règle de routage pour le trafic HTTP

tls : Une liste ordonnée de règle de routage pour le trafic TLS & HTTPS propagé

tcp : Une liste ordonnée de règle de routage pour le trafic opaque TCP.

exportTo : Une liste de namespaces dans lequel le virtual service est exporté



API Gateway Kubernetes

L'équivalent avec l'API Kubernetes est :

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: webapp-gateway
spec:
  gatewayClassName: istio
  listeners:
  - name: http
    hostname: "webapp.formation.io"
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
        from: Same # gateway doit être dans le même namespace que la destination
```

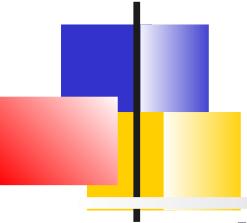
Cela déploie également un service de proxy associé.
On peut surveiller son statut par :

```
kubectl wait --for=condition=programmed gtw webapp-gateway
```

HttpRoute

La ressource **HttpRoute** de l'API Gateway de Kubernetes fait office de *VirtualService* :

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: httpbin
spec:
  parentRefs:
  - name: httpbin-gateway
  hostnames: ["httpbin.example.com"]
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /**
  backendRefs:
  - name: httpbin
    port: 8000
```

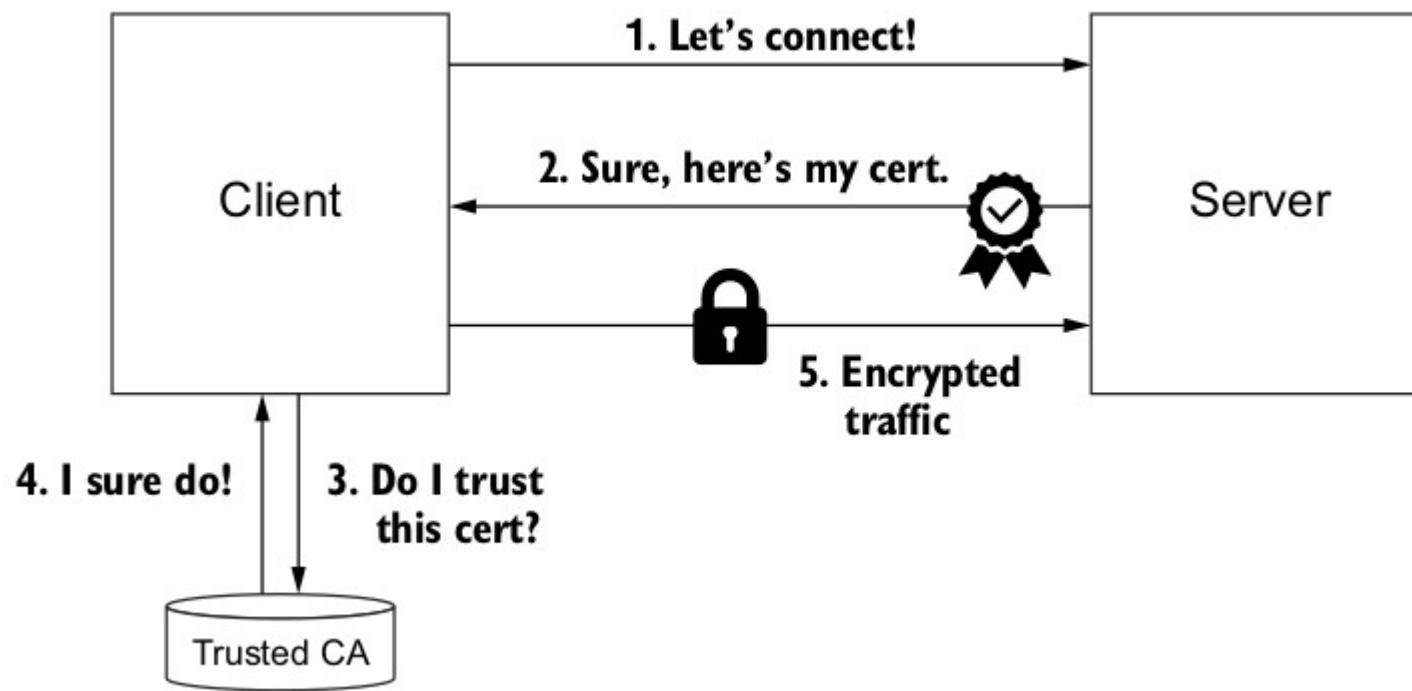


Sécurisation des gateways

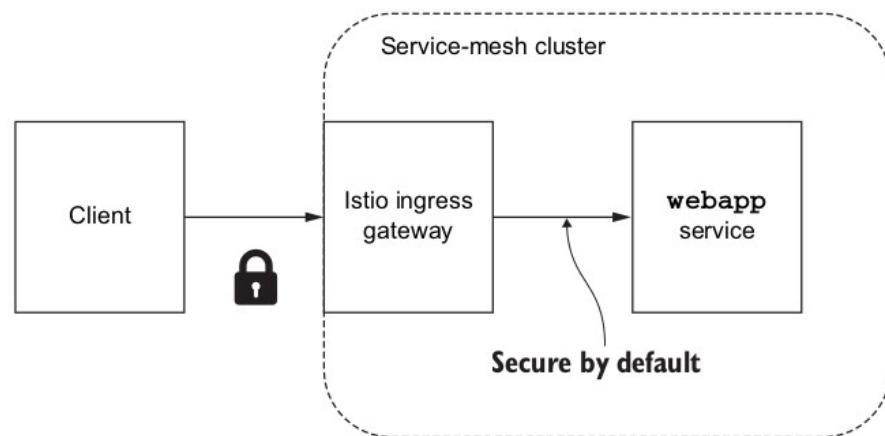
Les gateways d'Istio permettent :

- D'utiliser TLS/SSL pour le trafic entrant,
- De rediriger tout trafic non TLS vers les ports TLS appropriés
- De mettre en œuvre le TLS mutuel (*mTLS*).

Rappel TLS

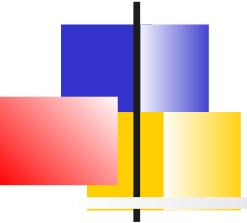


Traffic TLS



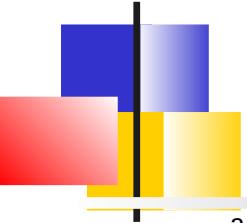
La gateway utilise la clé privée et le certificat du service *webapp*

- les certificats sont installés dans Kubernetes via des *secrets*



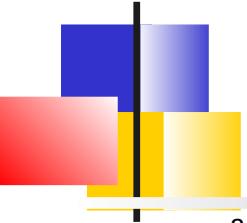
Configuration port https

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: httpbin-credential # must be the same as secret
  hosts:
  - httpbin.example.com
```



Configurer plusieurs hôtes virtuels avec TLS

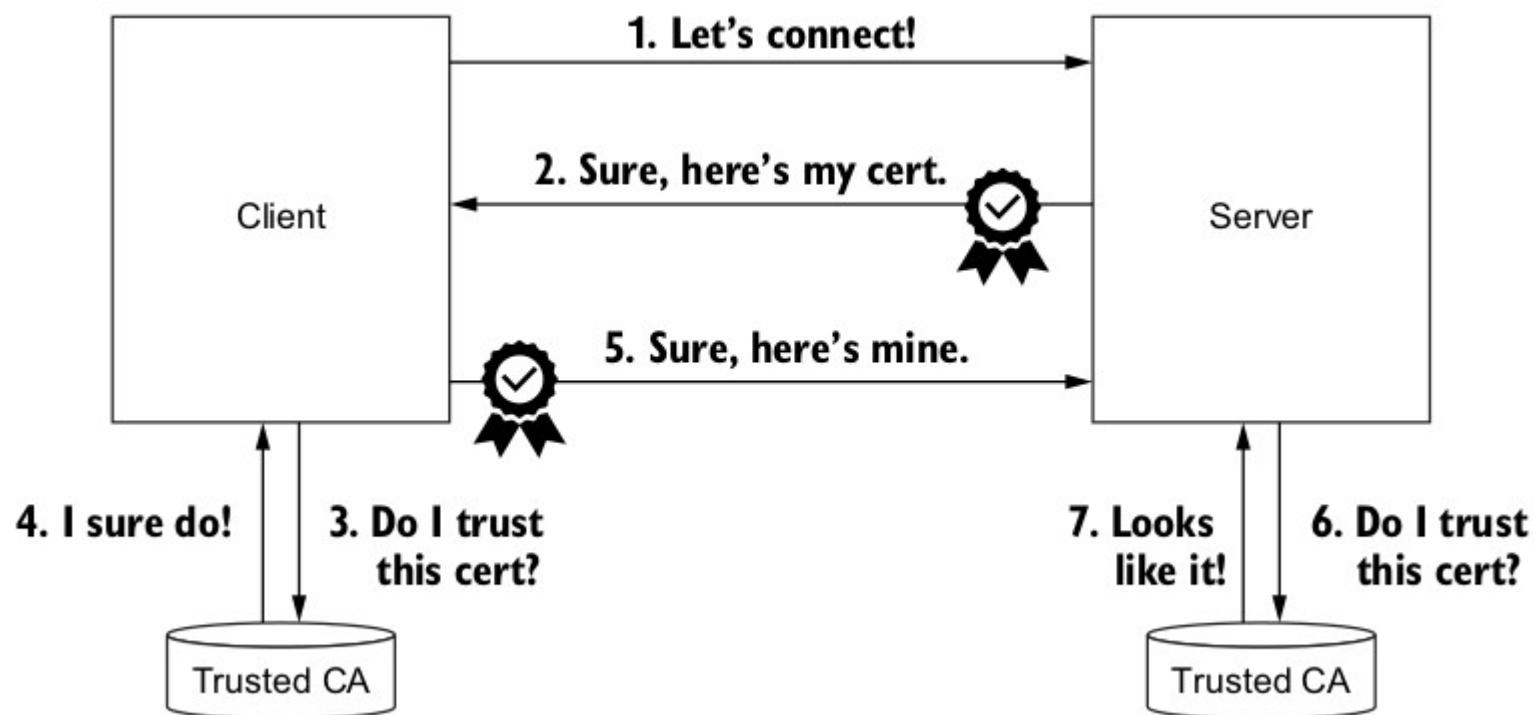
```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
    Istio: ingressgateway
  servers:
    - port:
        number: 443                      # 1ère entrée
        name: https-webapp
        protocol: HTTPS
        tls:
          mode: SIMPLE
          credentialName: webapp-credential
      hosts:
        - "webapp.formation.io"
    - port:                                # 2ème entrée
        number: 443
        name: https-catalog
        protocol: HTTPS
        tls:
          mode: SIMPLE
          credentialName: catalog-credential
      hosts:
        - "catalog.istioinaction.io"
```

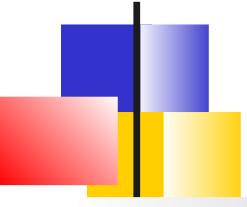


Configuration redirection vers https

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "webapp.formation.io"
      tls:
        httpsRedirect: true          # Redirection vers https
    - port:
        number: 443
        name: https
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: webapp-credential
      hosts:
        - "webapp.formation.io"
```

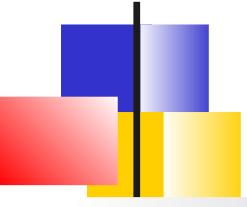
TLS mutuel (mTLS)





Configuration mTLS

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: mygateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: MUTUAL
      credentialName: httpbin-credential # must be the same as secret
  hosts:
  - httpbin.example.com
```



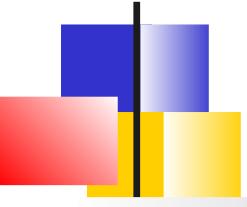
Istio gateway SDS

Une gateway Istio récupère les certificats via le ***secret discovery service (SDS)*** présent dans le processus *istio-agent* utilisé pour démarrer le proxy *istio*.

SDS est une API dynamique qui propage automatiquement les mises à jour.

Il est possible de vérifier le statut des certificats fournis par SDS avec :

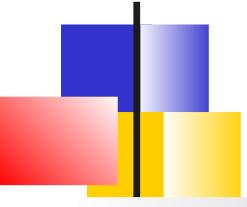
```
istioctl pc secret -n istio-system deploy/istio-ingressgateway
```



TCP

Istio peut travailler avec TCP mais n'offre pas les fonctionnalités de retry, de circuit-breaker, etc.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: echo-tcp-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 31400          # Port TCP à exposer
      name: tcp-echo
      protocol: TCP         # Protocole TCP
    hosts:
    - "*"                  # Pour tous les hôtes
```



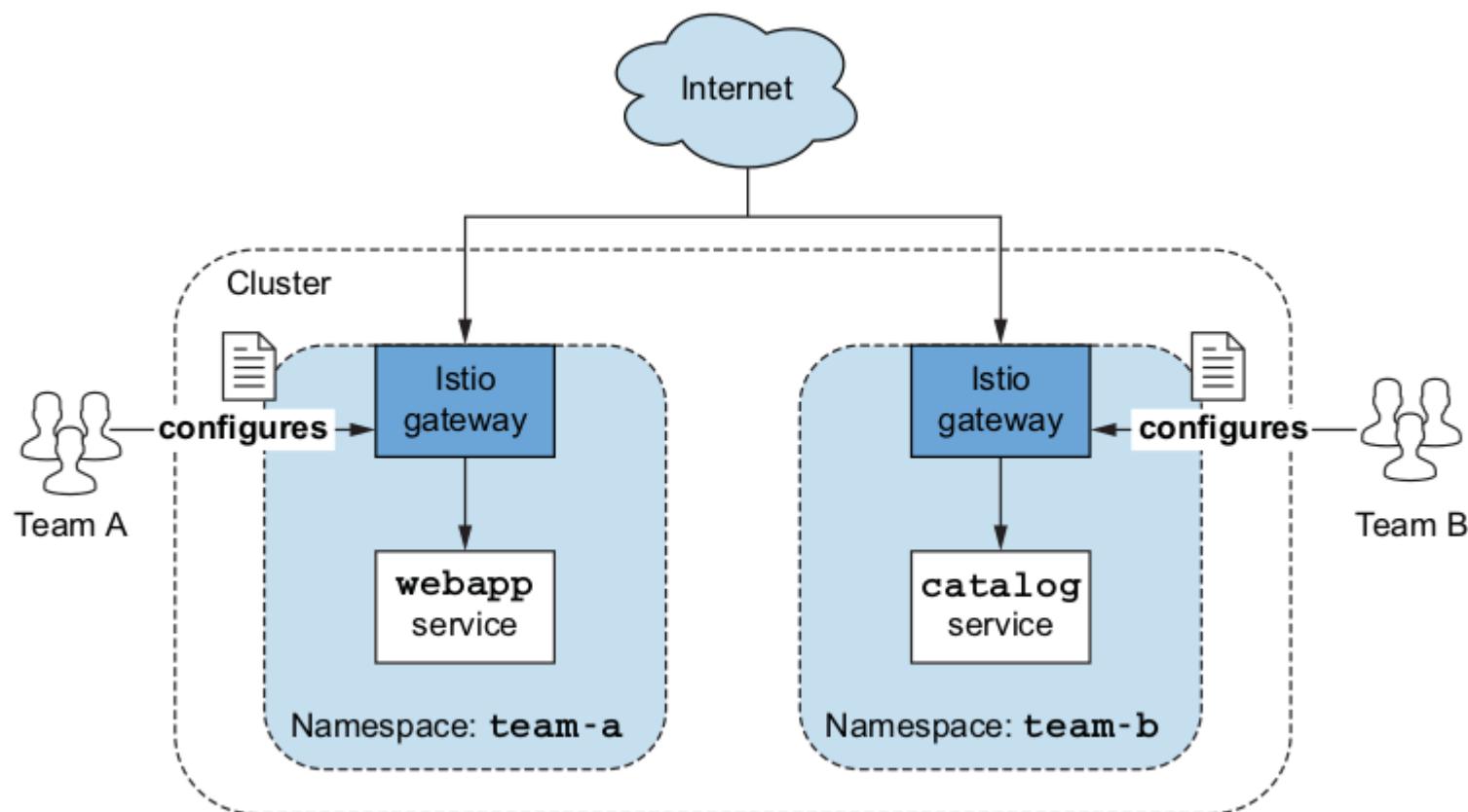
Usages de la Gateway

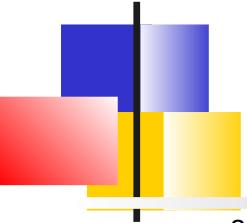
Une gateway peut être utilisée pour des différents usages : ingress, egress, shared-gateway, multi-cluster proxying.

On peut mettre en place plusieurs points *ingress* pour séparer le trafic des différents services (différentes contraintes de sécurité, de performance)

Chaque équipe maintient la configuration de leur gateways sans impacter les autres

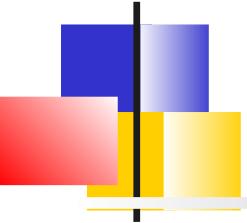
Multi Gateway





Exemple Multi-gateway

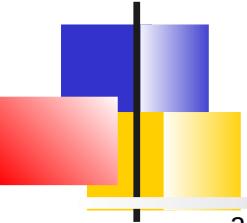
```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: my-user-gateway-install
  namespace: formation
spec:
  profile: empty
  values:
    gateways:
      istio-ingressgateway:
        autoscaleEnabled: false
  components:
    ingressGateways:
      - name: istio-ingressgateway          # Est déjà présent
        enabled: false
      - name: my-user-gateway
        namespace: formation
        enabled: true
        label:
          istio: my-user-gateway
-----
# Installation d'une nouvelle gateway pour le namespace formation
istioctl install -y -n formation -f my-user-gateway.yaml
```



Injection de gateway

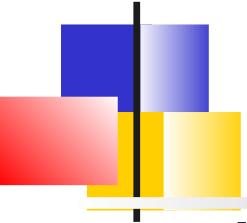
Afin que les équipes puissent mettre en place leur gateway en toute indépendance, on peut mettre en place de l'injection de gateway.

Au lieu d'utiliser *istioctl* pour déployer leur gateway, ils peuvent le faire via un manifeste Kubernetes



Injection de gateway

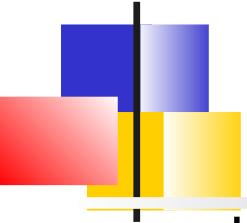
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-user-gateway-injected
  namespace: formation
spec:
  selector:
    matchLabels:
      ingress: my-user-gateway-injected
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true"
        inject.istio.io/templates: gateway      # Gabarit de la ConfigMap istio-sidecar-injector
    labels:
      ingress: my-user-gateway-injected
  spec:
    containers:
      - name: istio-proxy
        image: auto
```



Logs de la gateway

Le proxy d'Istio (Envoy) peut générer des traces d'accès.

- Dans le profil demo les accès sont affichés sur la console du pod
- Dans le profil default, les logs sont désactivés.
On peut les activer avec :
`istioctl install --set meshConfig.accessLogFile=/dev/stdout`

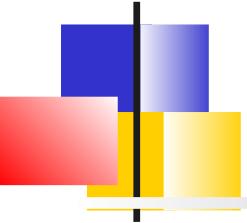


Activer les logs pour certaines gateway

Une meilleure approche dans un environnement de production est d'activer les logs seulement pour des gateways spécifiques.

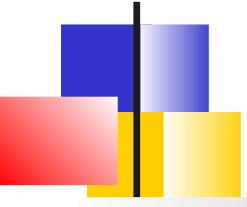
Cela peut être effectuer via des ressources **Telemetry**

```
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: ingress-gateway
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway      # Sélectionne la gateway voulue
  accessLogging:
  - providers:
    - name: envoy
  disabled: false                  # Active les traces
```



Ressources Istio

Gateway
Routing
Résilience
Observabilité
Sécurisation



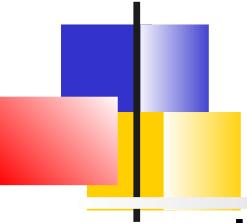
Routing et déploiement

Les règles de routing sont principalement utilisées lors d'un déploiement d'une nouvelle version d'un service

Sans interruption de service, les requêtes entrantes sont redirigées vers la nouvelle version.

Dans le mode natif Kubernetes, un roll-out est effectué. Les différentes répliques du service sont progressivement remplacées par la nouvelle version.

C'est un déploiement de type blue-green

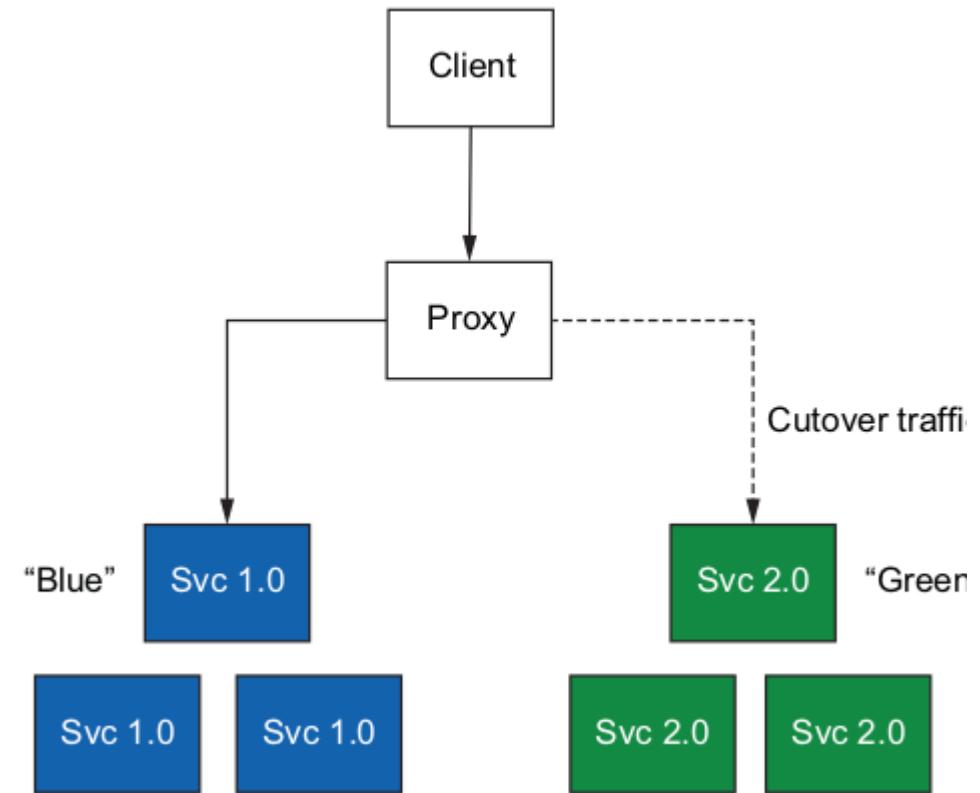


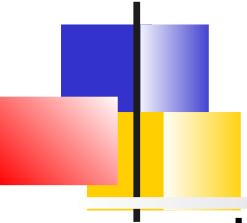
Déploiements Blue/Green

Les déploiements

Blue/green consistent à déployer une nouvelle release et lorsqu'elle est prête de basculer le trafic vers la nouvelle release.

Lors du basculement, on peut expérimenter un «big bang »





Distinction déploiement et release

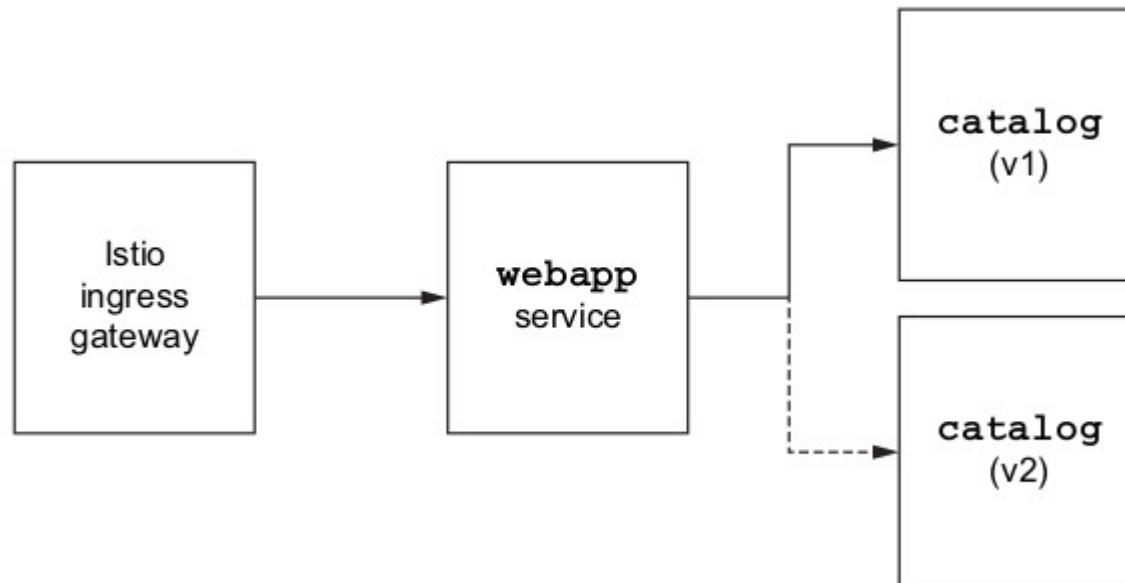
Les canary déploiement distinguent le déploiement de la release

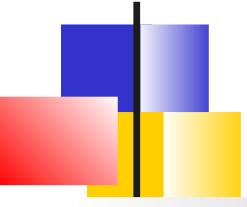
- Un déploiement consiste à installer le nouveau code en production mais aucun trafic réel ne lui est envoyé
Il est alors possible d'exécuter des tests afin de vérifier que tout marche comme prévu
- La release consiste à router du trafic vers le nouveau déploiement. Avec une canary release, seul un trafic partiel est routé (Utilisateurs interne, Beta-testeurs, ...) Il est encore possible de revenir en arrière à la version précédente
- Si les tests sont concluants, tout le trafic est routé vers le nouveau service

Subset et label

Pour effectuer une canary release, Nous devons indiquer à Istio comment identifier les charges de travail v1 et v2.

Dans un contexte Kubernetes, les labels **app** et **version** sur les ressources déploiement sont utilisés et des subsets du service sont définis en fonction de ces labels





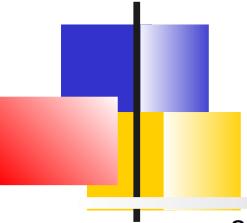
DestinationRule

Les ressources Istio **DestinationRule** permettent de spécifier des sous-ensembles de services, en regroupant par exemple toutes les instances d'un service d'une version donnée.

- Elles précisent également pour chaque sous-ensemble le load balancing, le mode de sécurité TLS ou la configuration des circuit-breaker

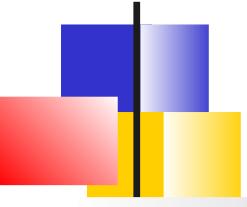
Ces sous-ensembles sont ensuite utilisés dans les règles de routage des *VirtualService*.

Les DestinationRule sont appliquées après l'évaluation des règles de routage d'un *VirtualService*
=> Elles déterminent la destination « réelle » du trafic.



Exemple *DestinationRule*

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews-destination-rule
spec:
  host: reviews.formation.svc.cluster.local      # Nom du service (Kubernetes, ServiceEntry)
  trafficPolicy:
    loadBalancer:
      simple: RANDOM                         # Algorithme de répartition entre sous-ensemble
  exportTo : '*'                                # Exporté à tous les namespaces (défaut)
  subsets:
    - name: v1                                  # Définition d'un subset
      labels:
        version: v1                            # Label identifiant le subset
    - name: v2
      labels:
        version: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN                  # Algorithme de répartition à l'intérieur sous-ensemble
    - name: v3
      labels:
        version: v3
```

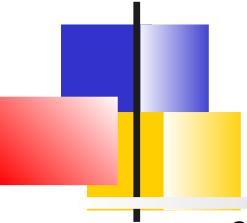


Utilisation dans un VirtualService

Le virtual Service peut définir les critères sur la requête qui route vers un des subsets définis via la directive `match`.

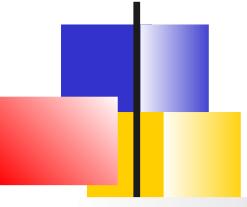
Dans le cas d'un trafic HTTP

- Les headers
- L'uri
- Le scheme
- La méthode
- Les paramètres de requêtes
- Le port
- Des labels du client
- Les namespaces du client
- ...



Routage avec les entêtes

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:                      # Services, IP de destination
  - reviews.formation.io
  http:                         # Règles de routage
  - match:
    - headers:
        end-user:
          exact: jason
    route:
      - destination:
          host: reviews
          subset: v2                  # Un sous-ensemble du service (worloads avec label)
      - route:                      # Route par défaut
          - destination:
              host: reviews
              subset: v3
```

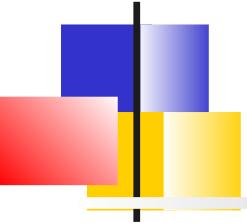


Pondération

Les règles de routage peuvent également être exprimées via la directive **weight**. La répartition entre subset est alors faite via de la pondération

Ex : 10 % du trafic vers v2 et 90 % vers v1

```
spec:  
  hosts:  
    - catalog  
  gateways:  
    - mesh  
  http:  
    - route:  
      - destination:  
          host: catalog  
          subset: version-v1  
          weight: 90  
      - destination:  
          host: catalog  
          subset: version-v2  
          weight: 10
```

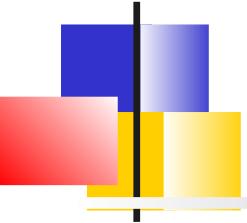


Automatisation des canary deployment

Certains outils permettent d'automatiser les canary deployment.

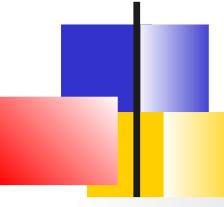
Flagger¹ s'intègre particulièrement bien avec Istio

Il s'appuie sur le métrique *health* pour les canary release



Exemple configuration

```
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: catalog-release
  namespace: formation
spec:
  targetRef:                      # Quel déploiement pour le canary
    apiVersion: apps/v1
    kind: Deployment
    name: catalog
  progressDeadlineSeconds: 60
  service:                          # Configuration du service
    name: catalog
    port: 80
    targetPort: 3000
    gateways:
    - mesh
  hosts:
  - catalog
```



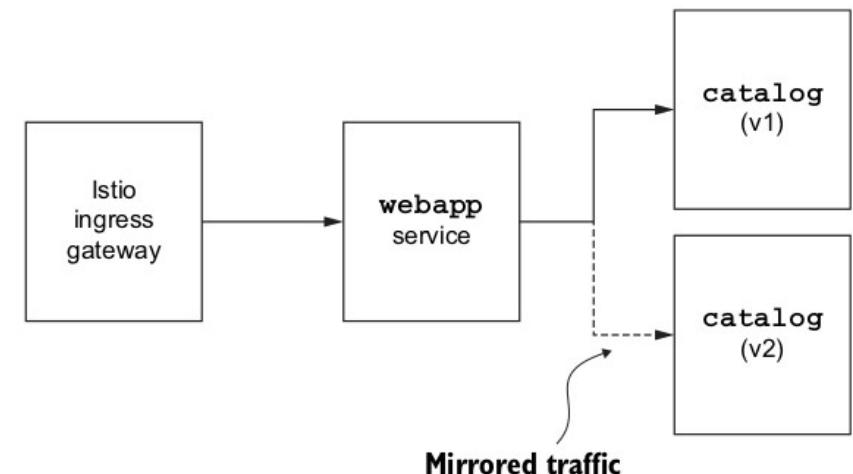
Exemple configuration (2)

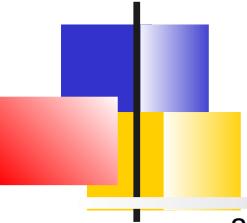
```
# Paramètre de progression du canary : à quelle vitesse promouvoir le canari,  
# quelles mesures pour déterminer la viabilité, les seuils pour déterminer le succès.  
# Evaluation toutes les 45 secondes, augmentation du trafic de 10% à chaque étape  
# A 50 % de trafic, on passe le trafic à 100 %.  
  
analysis:  
    interval: 45s  
    threshold: 5  
    maxWeight: 50  
    stepWeight: 10  
    match:  
        - sourceLabels:  
            app: webapp  
metrics:  
    - name: request-success-rate  
        thresholdRange:      # 99 % de succès sur une période d'une minute  
            min: 99  
        interval: 1m  
    - name: request-duration  
        thresholdRange:      # Maximum 500ms  
            max: 500  
        interval: 30s  
# Si ces seuils ne sont pas atteint le canari est roll-back
```

Trafic mirroring

Une autre approche consiste à mettre en miroir le trafic de production sur un nouveau déploiement hors d'accès des clients de production.

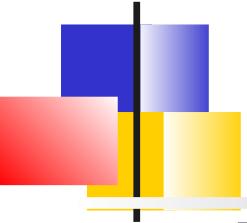
Cela permet d'obtenir des feedback sur le comportement d'un nouveau déploiement sans impacter les utilisateurs.





Configuration

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
    - catalog
  gateways:
    - mesh
  http:
    - route:
        - destination:
            host: catalog
            subset: version-v1
        weight: 100
      mirror:
        host: catalog
        subset: version-v2
```



Trafic externe

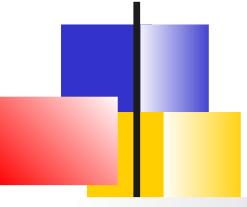
Par défaut, Istio autorise tout trafic sortant du maillage de services.

Cependant, puisque tout le trafic passe d'abord par le sidecar proxy, on peut contrôler le trafic externe :

- Bloquer tout trafic externe

```
istioctl install --set profile=demo \  
--set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY
```

- Contrôler le routage du trafic

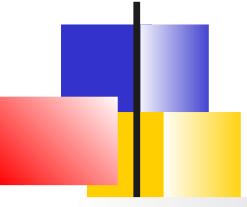


ServiceEntry

Istio crée un registre de services interne de tous les services connus par le maillage et accessibles au sein du maillage.

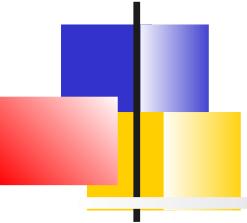
- Dans un environnement Kubernetes, le registre est alimenté par les services Kubernetes

Avec une ressource de type ***ServiceEntry***, il est possible d'ajouter un service externe au registre et d'autoriser le trafic



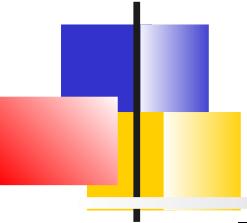
Configuration

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: jsonplaceholder
spec:
  hosts:
    - jsonplaceholder.typicode.com
  ports:
    - number: 80
      name: http
      protocol: HTTP
      resolution: DNS
      location: MESH_EXTERNAL
```



Ressources Istio

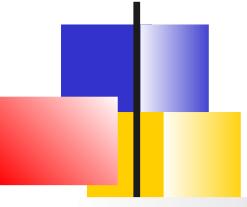
Gateway
Routing
Résilience
Observabilité
Sécurisation



Introduction

Istio implémente différents patterns de résilience :

- Équilibrage de charge côté client
- Équilibrage de charge sensible à la localité
- Délais d'expiration et retry
- Court-circuit (Circuit-beaker)

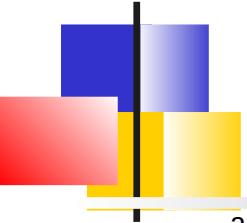


Répartition de charge côté client

L'équilibrage de charge côté client consiste à informer le client des différents points d'accès disponibles et le laisser choisir des algorithmes d'équilibrage de charge spécifiques pour une répartition optimale.

Ceci est défini via une ***DestinationRule trafficPolicy.loadBalancer*** :

- Round robin (défaut)
- Au hasard
- LEAST_REQUEST : répartition vers le pod qui a le moins de requêtes en attente
- Sticky session
- Localité
- ...



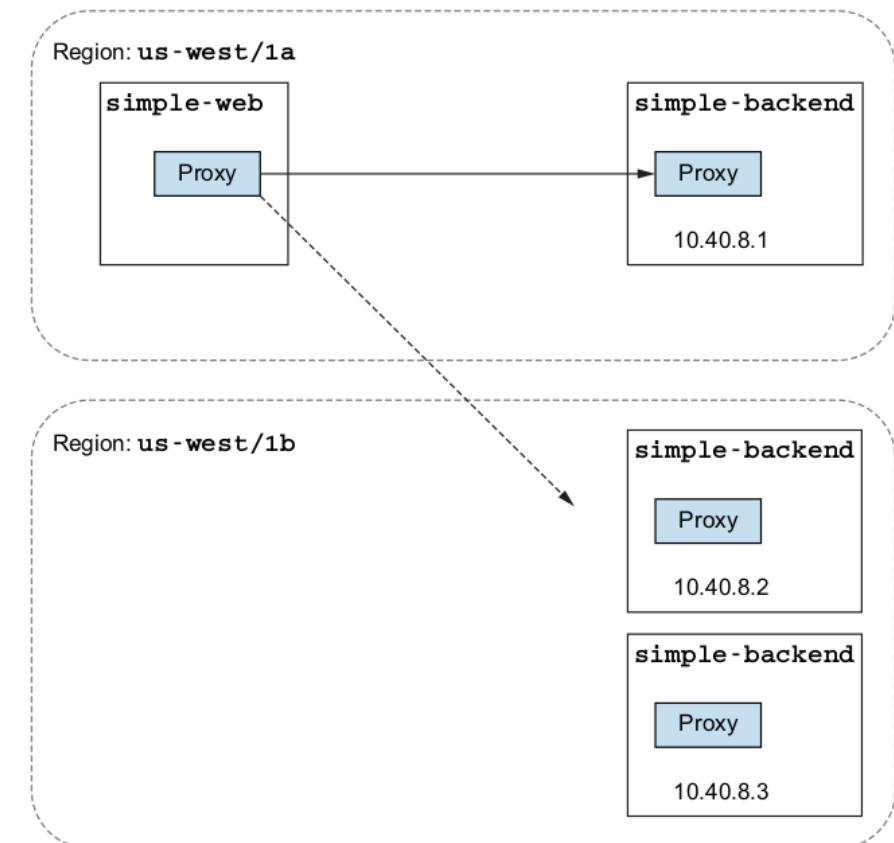
Configuration – Moins chargé

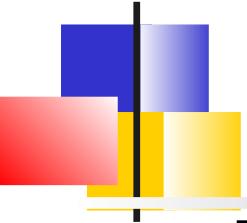
```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: LEAST_REQUEST
```

Répartition en fonction de la localité

Des labels peuvent identifier la région et la zone de disponibilité où une instance est déployée.

Istio se sert de ses informations pour donner la priorité à l'instance la plus proche



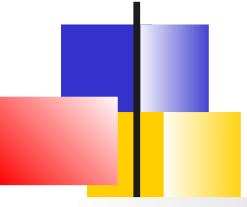


Timeouts

Pour se prémunir contre des échecs en cascade, on se doit d'implémenter des timeouts sur les temps connexions et/ou les requêtes

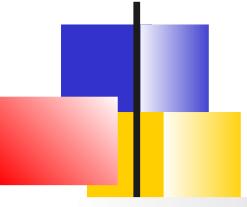
- Généralement, des délais d'attente plus longs en entrée de l'architecture que dans les couches plus basses

Les timeouts sont configurés via la ressource
VirtualService timeout



Configuration

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-backend-vs
spec:
  hosts:
    - simple-backend
  http:
    - route:
        - destination:
            host: simple-backend
  timeout: 0.5s
```



Ré-essais

Sans retry, les services sont vulnérables aux défaillances courantes.

D'un autre côté, trop de retry peuvent contribuer à la dégradation du système en provoquant des défaillances en cascade.

- Chaque retry a son propre délai (*perTryTimeout*)
=> *perTryTimeout *nbRetry* doit être plus petit que timeout global

Par défaut, Istio réessaie jusqu'à deux fois sur des erreurs 503¹. Si l'on veut désactiver :

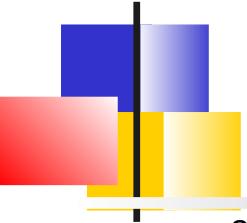
```
istioctl install --set profile=demo --set meshConfig.defaultHttpRetryPolicy.attempts=0
```

Le délai par défaut est de 25ms.

Les retry sont configurés via une ressource

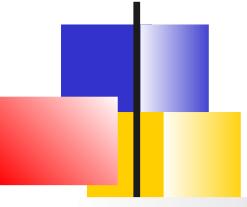
VirtualService.retries

1. En tout cela fait 3 essais



Configuration

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-backend-vs
spec:
  hosts:
    - simple-backend
  http:
    - route:
        - destination:
            host: simple-backend
  retries:
    attempts: 2
    retryOn: gateway-error,connect-failure,retriable-4xx
    perTryTimeout: 300ms
    retryRemoteLocalities: true
```



Circuit-breaker

La fonctionnalité de *circuit-breaker* sert à se prémunir contre les défaillances partielles pouvant mener en des défaillances en cascade.

L'idée est de réduire le trafic vers des systèmes défectueux, afin de ne pas continuer à les surcharger et les empêcher de se récupérer

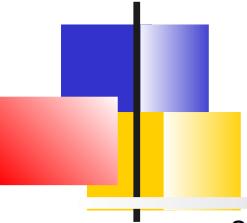
Istio ne propose pas de configuration explicite de type "disjoncteur", mais il fournit 2 moyens pour limiter la charge sur les services backend :

- Contrôler le nombre de connexions et de requêtes en attente pour un service spécifique

DestinationRule.connectionPool

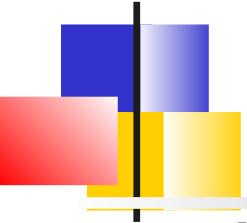
- Observer la santé des points d'accès dans un équilibrage de charge et expulser pendant un certain temps ceux qui se comportent mal

DestinationRule.trafficPolicy.outlierDetection



Exemple configuration connexion / requête

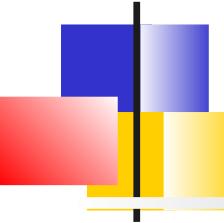
```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.formation.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1    # Le seuil pour un débordement de connexion.
      http:
        http1MaxPendingRequests: 1    # Nombre autorisé de requêtes en attente de connexion .
        maxRequestsPerConnection: 1
        maxRetries: 1
        http2MaxRequests: 1    # Nombre maximal de requête en // sur tous les endpoints
```



Gestion des surcharge

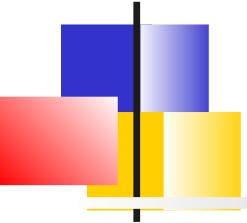
Lorsqu'une requête échoue à cause d'une ouverture de circuit, *Istio* ajoute une en-tête ***x-envoy-overloaded***

C'est à la charge du client de surveiller cette entête et de prendre les mesures adéquates : *fall-back*



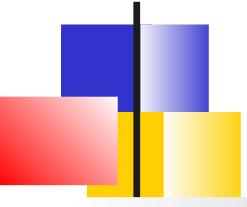
Exemple configuration instance en mauvaise santé

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.formation.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      consecutive5xxErrors: 3 # Détection déclenchée pour 3 5xx consécutives
      interval: 5s           # Vérification et prise de décision toutes les 5s
      baseEjectionTime: 5s   # La durée d'éjection nbEjections*baseEjectionTime
      maxEjectionPercent: 100 # La part d'instances éligibles pour l'éjection
```



Ressources Istio

Gateway
Routing
Résilience
Observabilité
Sécurisation



Introduction

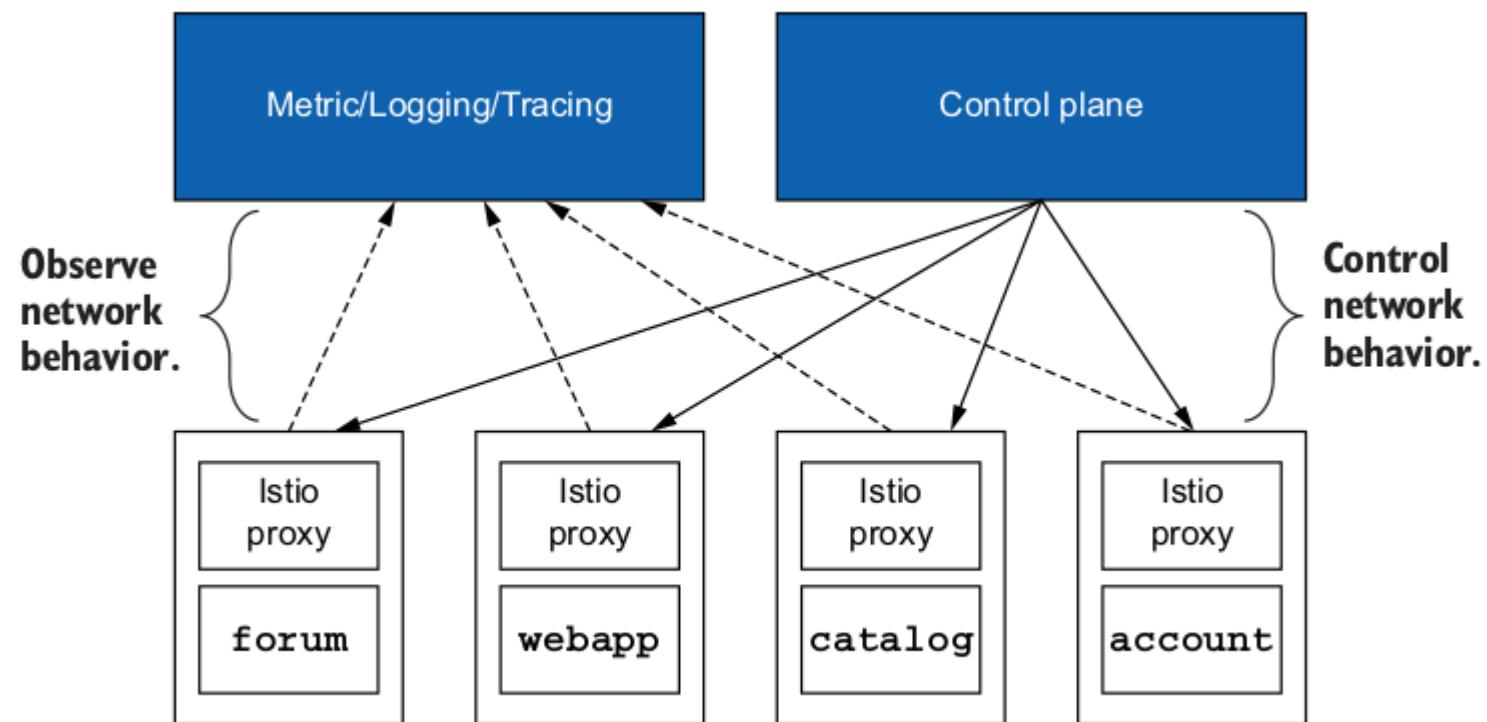
Istio, via ses proxy, est dans une position idéale pour aider à créer un système observable

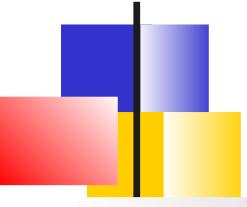
Il peut capturer des métriques importantes telles que le nombre de demandes par seconde, la durée des demandes, le nombre de demandes ayant échoué, etc.

Il peut également aider à ajouter dynamiquement de nouvelles métriques

Istio est fourni avec des exemples d'outils prêts à l'emploi, tels que Prometheus, Grafana et Kiali qui permettent les démonstrations

Position idéale





Métriques Data Plane

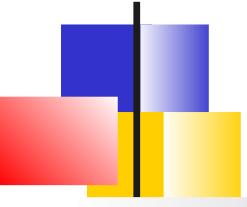
Le proxy associé à chaque POD expose un ensemble important de métriques au format Prometheus sur le port 5000

Il est accessible soit par un client web (curl) ou par le processus **pilot-agent**

```
kubectl exec -it deploy/webapp -c istio-proxy -- pilot-agent request GET stats
```

Les statistiques les plus importantes pour les développeurs d'application sont :

- ***istio_requests_total*** : Compteur de requêtes
- ***istio_request_bytes*** : Distribution des tailles de requêtes
- ***istio_response_bytes*** : Distribution des tailles de réponses
- ***istio_request_duration_milliseconds*** : Distribution des durées de requêtes



Personnalisation

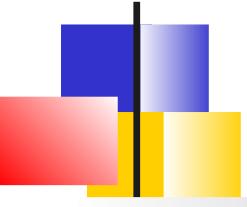
Les proxies peuvent être configurés afin qu'ils envoient plus de métriques¹.

Cette configuration peut être fait

- au niveau du mesh ... mais attention à la surcharge !
- Au niveau d'une workload via des annotations

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
...
spec:
  meshConfig:
    defaultConfig: # Default config pour tous les services
    proxyStatsMatcher: # Personnalisation des métriques
      inclusionPrefixes:
        - "cluster.outbound|80|catalog.formation" # Les métriques qui matchent
```

1. Voir : <http://mng.bz/9K08>



Types de métriques

<cluster_name>.internal.* : Les requêtes internes au mesh

<cluster_name>.ssl.* : Permet de voir toutes les métriques relatives à SSL

upstream_cx et **upstream_rq** : donnent toutes les informations réseau des connexions entrantes

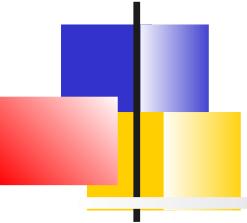
cluster.<name>.health_check.* : Métriques de santé

cluster.<name>.outlier_detection.* : Métriques sur les éjections

cluster.<name>.circuit_breakers.<priority>. : Métriques par rapport aux seuils des circuit breaker

cluster.<name>.lb* : Métriques sur les load balancer

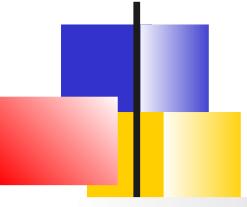
...



Connaissance du mesh

Chaque Proxy possède également des métriques sur les autres pods du mesh :

```
kubectl exec -it deploy/webapp -c istio-proxy \
-- curl localhost:15000/clusters
```

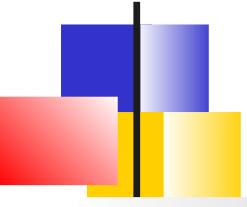


Métriques du Control Plane

istiod conserve beaucoup d'informations sur son activité
(Configurations des proxys, l'émission/rotation des certificats, etc.)

```
kubectl exec -it -n istio-system deploy/istiod -- curl localhost:15014/metrics
```

- Certificat racine servant à signer les CSR des workloads (***citadel_server***)
- Version du control plane (***istio_build***)
- Performance de la configuration (***pilot_proxy_convergence***)
- Combien de services sont connus, combien de VirtualService configurés, combien de proxys connecté (***pilot_****)
- Le nombre de mises à jour « poussées » par l'API xDS (***pilot_ds_pushes***)



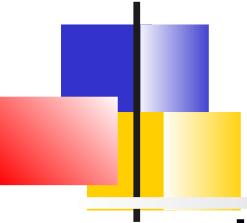
Intégration Prometheus

Prometheus tire les métriques des cibles via des endpoints

Le service proxy expose les métriques au format Prometheus

```
kubectl exec -it deploy/webapp -c istio-proxy \  
-- curl localhost:15090/stats/prometheus
```

Le système ***kube-prometheus*** est un système typique de Kubernetes hautement scalable et qui inclut *Prometheus*, *Prometheus operator*, *Grafana* et d'autres composants auxiliaires.

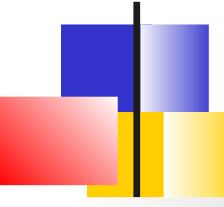


kube-prometheus

kube-prometheus s'installe typiquement avec Helm.

La configuration des cibles s'effectue grâce à un opérateur capable de déployer des ressources de types ***ServiceMonitor*** ou ***PodMonitor***

Dans le cas d'Istio, les cibles sont les proxy envoy (*/stats/prometheus*) ainsi que le service istiod (*/metrics*)



Personnalisation

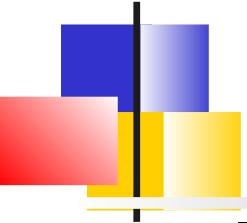
Par défaut, Istio fournit un ensemble de métriques mais ceci est configurable

La configuration est faite par le proxy stats via une ressource EnvoyFilter¹

Il est possible d'affiner la configuration

- Ajout de dimension à 1 métrique existant
- Création de nouveaux métriques
- Grouper des attributs existants (par exemple avoir une information pour les requêtes qui contiennent un path ou paramètre particulier)

1. <https://istio.io/latest/docs/reference/config/networking/envoy-filter/>



Grafana

Istio a des tableaux de bord Grafana prédéfinis mais ils ne font plus partie de la distribution¹

Dans l'environnement Kubernetes, le chargement des tableaux de bord Grafana s'effectue via une *ConfigMap* labellisé avec :
grafana_dashboard=1

1. Voir : <https://grafana.com/orgs/istio/dashboards>

datasource default

Deployed Versions

Pilot Versions



Resource Usage

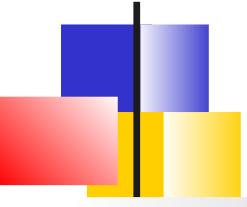


Pilot Push Information



Tdb Service



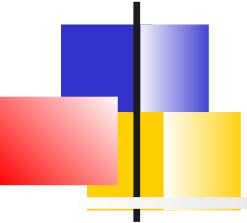


Tracing distribué

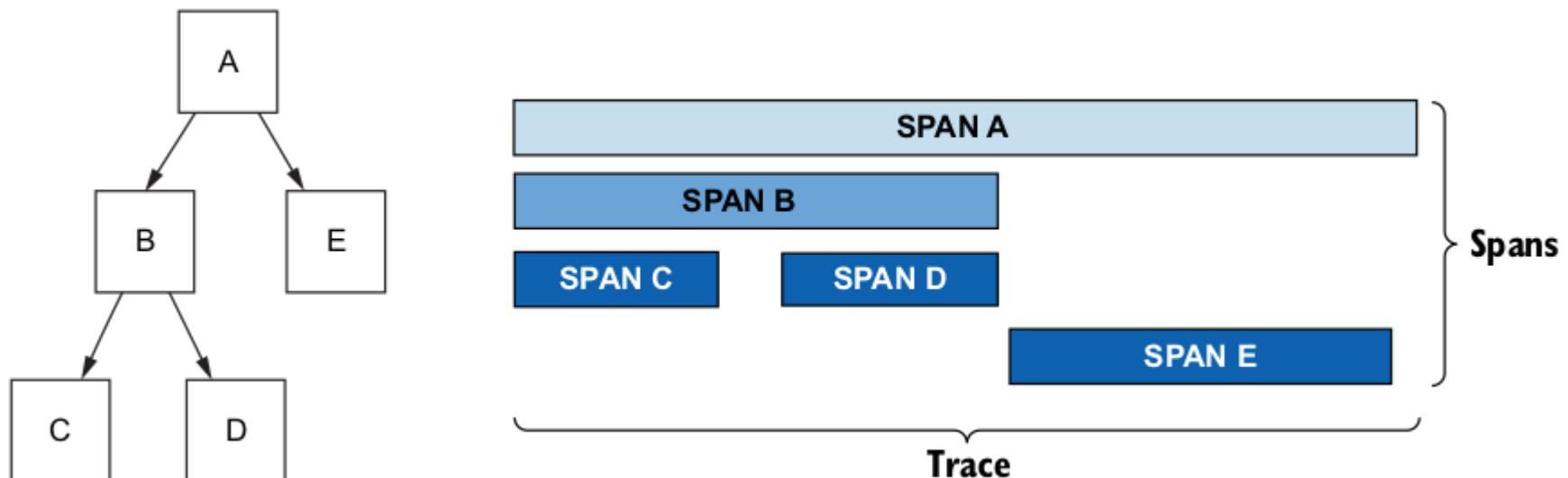
Le tracing distribué consiste à annoter les requêtes avec des ID de corrélation représentant les appels entre services et un ID de suivi représentant une requête.

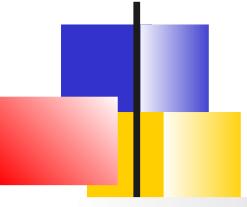
Le proxy Istio peut ajouter ces métadonnées et les envoyer vers une moteur d'agrégation implémentant la spécification OpenTracing

- Jaeger
- Zipkin
- Lightstep
- Instana



Trace et spans





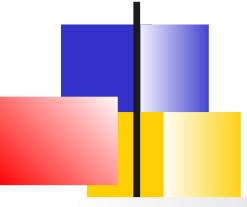
Mécanisme

Lorsqu'une requête traverse le proxy de service Istio, une nouvelle trace est démarrée s'il n'y en a pas une en cours, et les temps de début et de fin sont capturées dans le cadre du *Span*.

Istio ajoute à la requête des en-têtes HTTP, appelés en-têtes de suivi Zipkin, qui peuvent être utilisés pour corrélérer les objets Span suivants à l'ensemble Trace

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context.

Du point de vue applicatif, il faut s'assurer que ces entêtes sont propagées

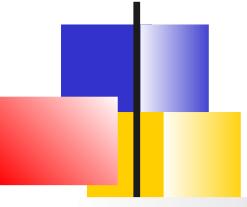


Configuration Istio

La configuration peut se faire à différents niveau : global au mesh, pour un espace de nom ou une workload spécifique.

Istio supporte Zipkin, Datadog, Jaeger et d'autres, leur activation s'effectue via une ressource ***IstioOperator***

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  namespace: istio-system
spec:
  meshConfig:
    defaultConfig:
      tracing:
        lightstep: {}
        zipkin: {}
        datadog: {}
        stackdriver: {}
```



Échantillonnage des traces

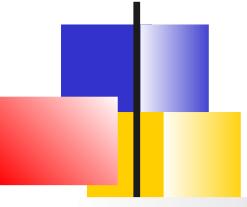
Le tracing distribué peuvent provoquer des dégradations de performance.

=> On peut choisir la fréquence de collecte des traces.

La fréquence est configurée dans la *ConfigMag*

MeshConfig qui peut être édité à tout moment

```
apiVersion: v1
data:
  mesh: |-
    accessLogFile: /dev/stdout
    defaultConfig:
      discoveryAddress: istiod.istio-system.svc:15012
      proxyMetadata: {}
    tracing:
      sampling: 10      # 10 % de collecte
    zipkin:
      address: zipkin.istio-system:9411
```

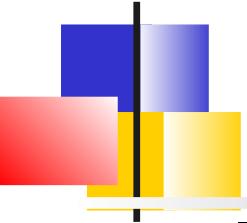


Configuration fine

La configuration de la fréquence d'échantillonage au niveau espace de nom ou workload est effectué via des annotations

```
spec:  
  template:  
    metadata:  
      annotations:  
        proxy.istio.io/config: |  
          tracing:  
            sampling: 10  
        zipkin:  
          address: zipkin.istio-system:9411
```

Il est même possible d'activer les traces pour des requêtes spécifiques, il suffit d'ajouter l'entête **x-envoy-force-trace**



Personnalisation des tags d'une trace

Il est possible d'attacher des métadonnées supplémentaires à une trace.

Le tag peut être valué à :

- Une valeur en dur
- Une variable d'environnement
- Une entête HTTP

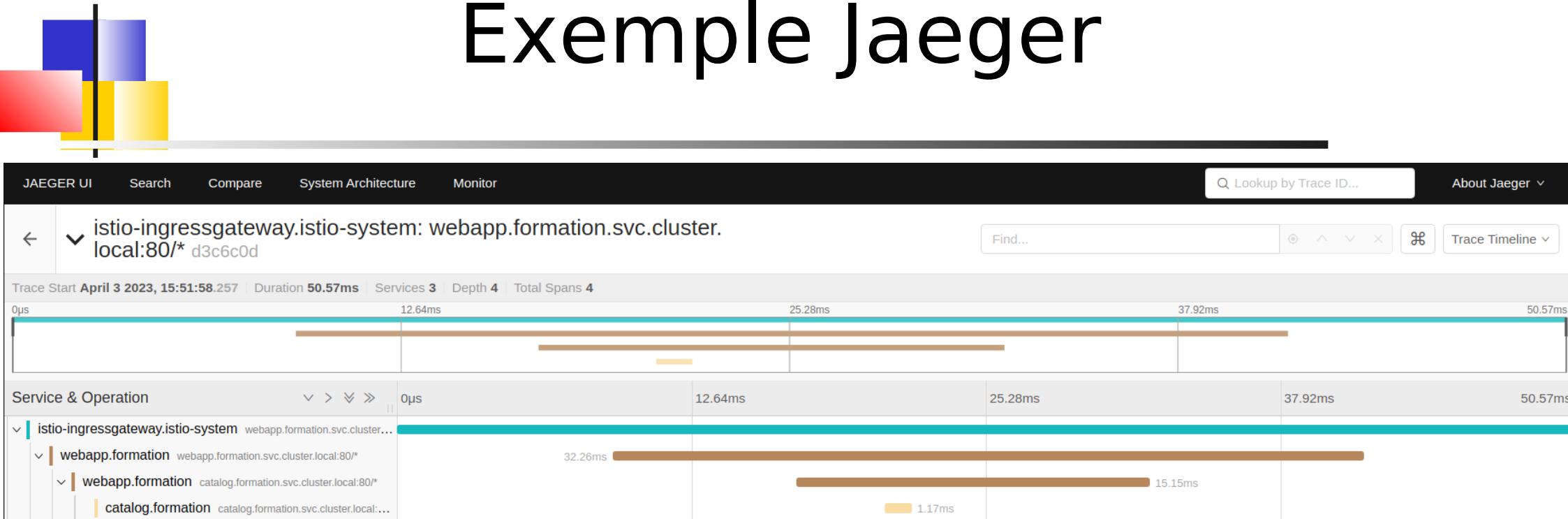
Exemple jaeger

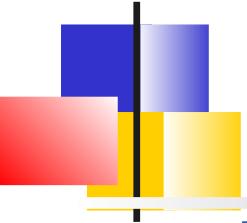
The screenshot shows the Jaeger UI interface. At the top, there is a navigation bar with tabs: JAEGER UI (selected), Search, Compare, System Architecture, and Monitor. To the right of the navigation bar is a search bar labeled "Lookup by Trace ID..." and a link "About Jaeger".

The main area is divided into sections:

- Search:** Contains fields for Service (set to "istio-ingressgateway.istio-system"), Operation (set to "all"), and Tags (set to "http.status_code=200 error=true"). It also includes a "Lookback" dropdown set to "Last Hour" and "Max Duration" and "Min Duration" input fields.
- JSON File:** A button to upload JSON trace files.
- Timeline:** A chart showing the duration of two traces over time. The Y-axis represents Duration (ms) with marks at 500ms and 1s. The X-axis shows times from 03:46:40 pm to 03:51:40 pm. Two green dots represent the start of each trace.
- Traces Summary:** Displays "2 Traces" with sorting options "Most Recent" and "Deep Dependency Graph".
- Compare traces:** A section titled "Compare traces by selecting result items" showing two trace results. The first result is for "istio-ingressgateway.istio-system: webapp.formation.svc.cluster.local:80/* d3c6c0d" with a duration of 50.57ms, involving 4 spans across three services: catalog.formation (1), istio-ingressgateway.istio-system (1), and webapp.formation (2). The second result is for "istio-ingressgateway.istio-system: httpbin.org:80/* 24abd44" with a duration of 1.41s, involving 1 span for istio-ingressgateway.istio-system (1).

Exemple Jaeger





Kiali

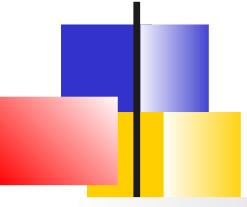
Kiali est un projet OpenSource de visualisation.

Il permet de visualiser le mesh d'Istio pendant son exécution

Il récupère ses données de Prometheus pour proposer des graphes interactifs

Il existe un Kiali Operator¹ permettant de l'installer dans un environnement de production

1. <https://github.com/kiali/kiali-operator>



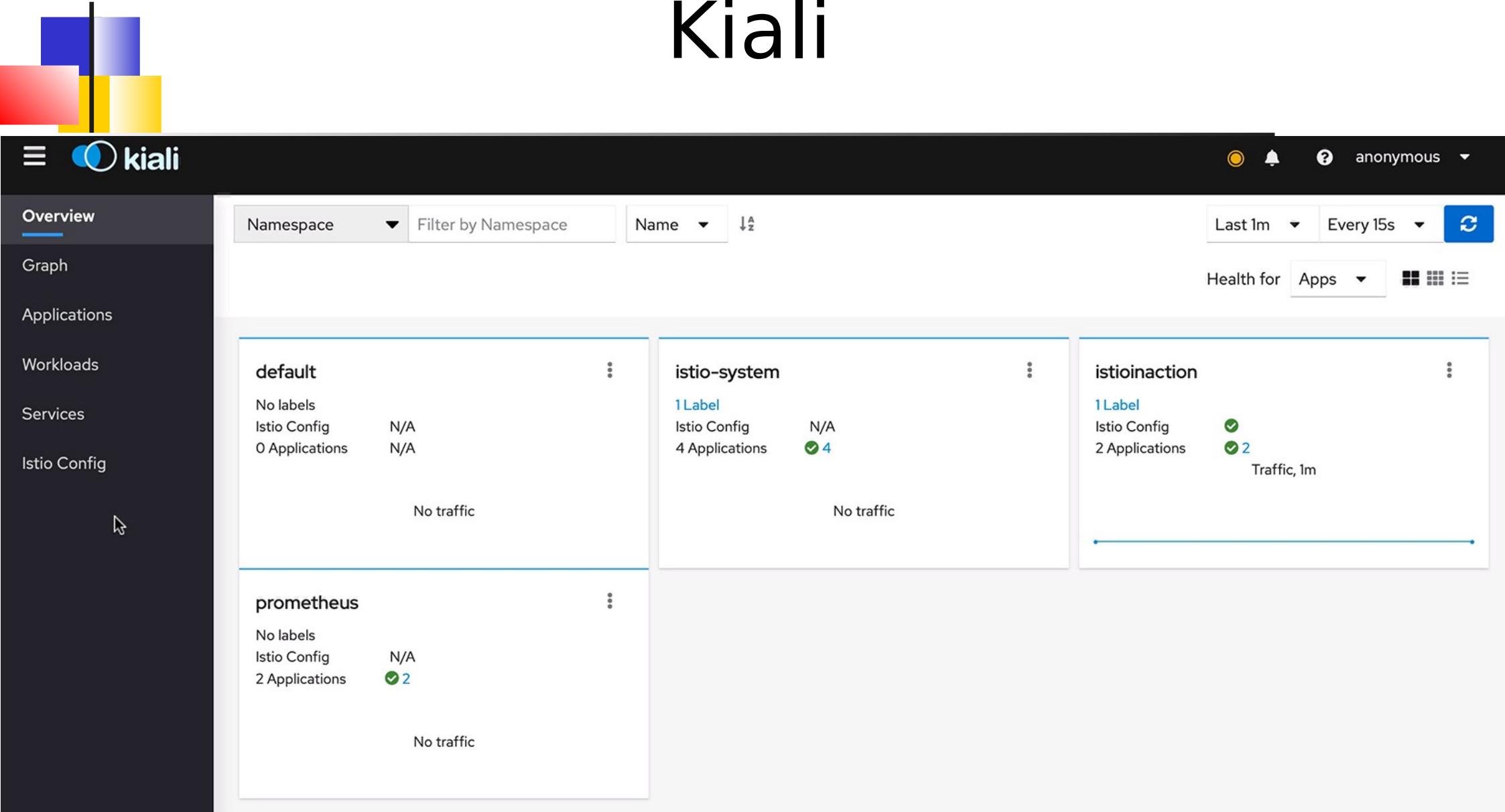
Operator kiali

L'operator ***kiali-operator*** s'installe typiquement via Helm.

L'opérateur s'appuie sur une ressource CRD de type *kiali* pour le déploiement du service kiali

En fonction de son environnement, il faut passer ses propres valeurs des URLs Prometheus, Grafana, Tracing à la commande d'installation Helm ou éditer à posteriori la CRD

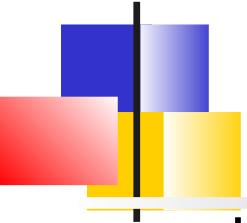
Kiali



The image shows the Kiali user interface, a tool for monitoring and managing Istio traffic. The top navigation bar includes a logo, a search bar, and user authentication information. The left sidebar lists navigation options: Overview (selected), Graph, Applications, Workloads, Services, and Istio Config. The main content area displays four service meshes in cards:

- default**: No labels, Istio Config N/A, 0 Applications, N/A. Status: No traffic.
- istio-system**: 1 Label, Istio Config N/A, 4 Applications (4 green checkmarks). Status: No traffic.
- istioinaction**: 1 Label, Istio Config (green checkmark), 2 Applications (2 green checkmarks). Status: Traffic, 1m.
- prometheus**: No labels, Istio Config N/A, 2 Applications (2 green checkmarks). Status: No traffic.

Time range controls at the top right allow filtering by "Last 1m" and "Every 15s". A refresh button and a "Health for Apps" dropdown are also present.



Overview et Graph

Le **tableau de bord principal** affiche les différents espaces de noms et le nombre d'applications en cours d'exécution avec une indication visuelle pour la santé globale

- On peut accéder au détail d'un espace de nom

L'onglet **Graph** permet de visualiser les flux : Nombre d'octets, de requêtes, les flux par versions (Canary ou load balancing), Requêtes/seconde, pourcentage du trafic total par versions, Santé des applications basée sur le trafic réseau, Trafic HTTP/TCP, les pannes de réseau.

On peut accéder au détail d'un workload

On peut accéder à la config Istio

Graph

kiali

Overview Graph Applications Workloads Services Istio Config

Namespace: formation | Traffic | Versioned app graph

Last 1m | Every 15s

Display Find... Hide... i

Graph tour

Apr 3, 04:24:49 PM ... 04:25:49 PM

istio-ingressgateway latest (istio-system) 2 hosts

webapp latest catalog v1

A webapp A catalog

Legend

S webapp ✓ health

Has Request Routing > 1 host

Traffic Traces

HTTP (requests per second):

	Total	%Success	%Error
In	33.33	100.00	0.00
Out	33.33	100.00	0.00

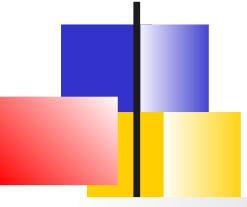
Out In

OK 3xx 4xx 5xx NR

HTTP - Request Traffic min / max:
Not enough traffic to generate chart.

No gRPC traffic logged

```
graph LR; EG[istio-ingressgateway latest (istio-system)] --> Webapp[webapp latest]; Webapp --> Catalog[catalog v1]; Catalog --> Webapp
```

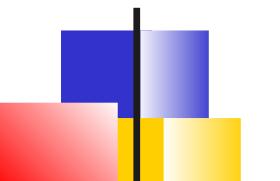


Workload

L'onglet Workload propose plusieurs sous-menus :

- **Overview** : Pods du service, sa configuration Istio et un graphe des flux amont et aval
- **Trafic** : Taux de réussite du trafic entrant et sortant
- **Logs** : Logs applicatifs, Logs d'accès Envoy et les spans corrélées
- **Inbound Metrics** et **Outbound Metrics** : Corrélés avec les spans
- **Traces** : Les traces vues par Jaeger
- **Envoy** : La configuration Envoy configuration appliquée à la workload comme les clusters, listeners et routes

Workload



☰ kiali

Overview

Graph

Applications

Workloads

Services

Istio Config

Workloads > Namespace: formation > webapp

Last 10m ▾ Every 15s ▾

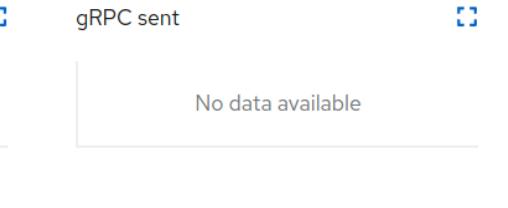
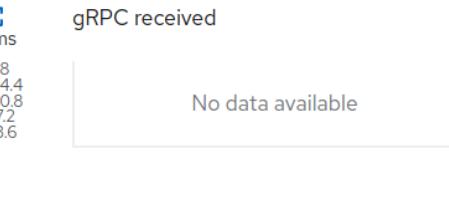
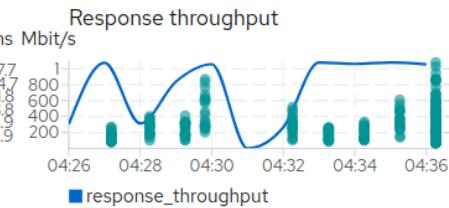
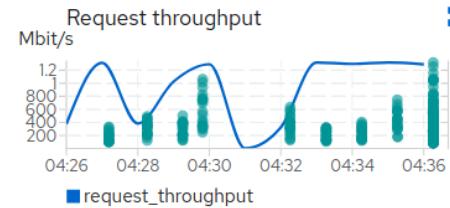
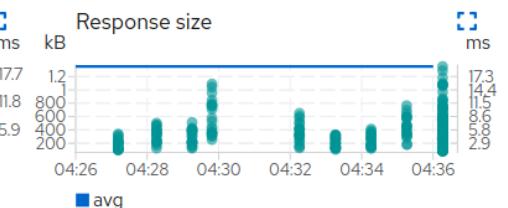
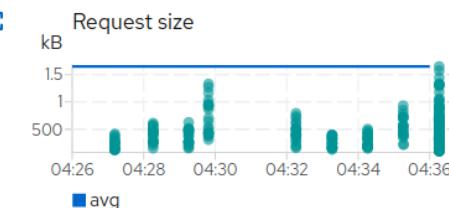
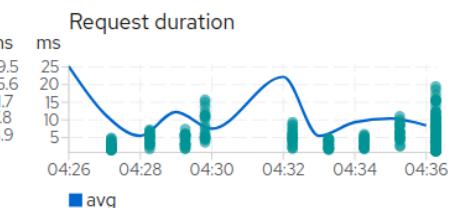
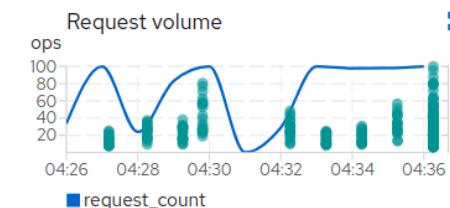
Overview Traffic Logs Inbound Metrics Outbound Metrics Traces Envoy

Metrics Settings ▾

Reported from

Destination ▾

Spans



TCP opened

TCP closed

TCP received

TCP sent

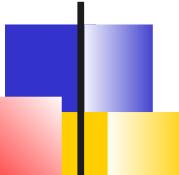
No data available

No data available

No data available

No data available

Config Istio



The Kiali interface showing the Istio Config page for the 'formation' namespace.

Navigation bar:

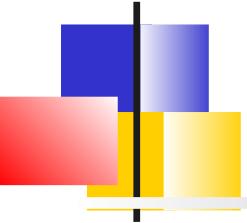
- Overview
- Graph
- Applications
- Workloads
- Services
- Istio Config (selected)

Header:

- Namespace: formation
- Actions

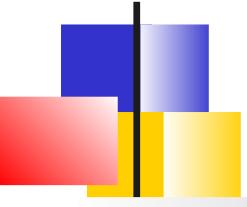
Istio Config Table:

Istio Type	Name	Namespace	Type	Configuration
G	coolstore-gateway	NS formation	Gateway	✓
SE	external-httpbin-org	NS formation	ServiceEntry	✓
SE	jsonplaceholder	NS formation	ServiceEntry	✓
VS	thin-httbin-virtualservice	NS formation	VirtualService	✓
VS	webapp-virtualservice	NS formation	VirtualService	✓



Ressources Istio

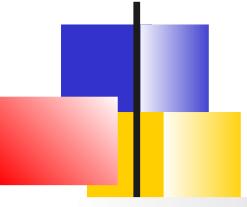
Gateway
Routing
Résilience
Observabilité
Sécurisation



Introduction

Afin de protéger les données utilisateur, il est nécessaire de :

- Authentifier et vérifier les permissions avant d'autoriser l'accès à une ressource
 - Authentication service vers service
 - Authentication utilisateur final
 - Vérifier les permissions de l'entité authentifiée
- Chiffrer les données en transit



Istio et SPIFFE

Istio utilise la spécification SPIFFE afin de fournir une identité aux workloads

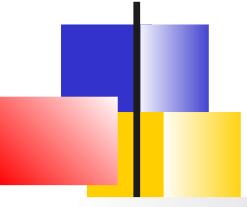
L'identité SPIFFE¹ est une URI

spiffe:/trust-domain/path

- **trust-domain** représente l'émetteur des identités
- **path** : identifie de manière unique une charge de travail dans le domaine de confiance.

Istio renseigne le chemin avec le compte de service exécutant la workload.

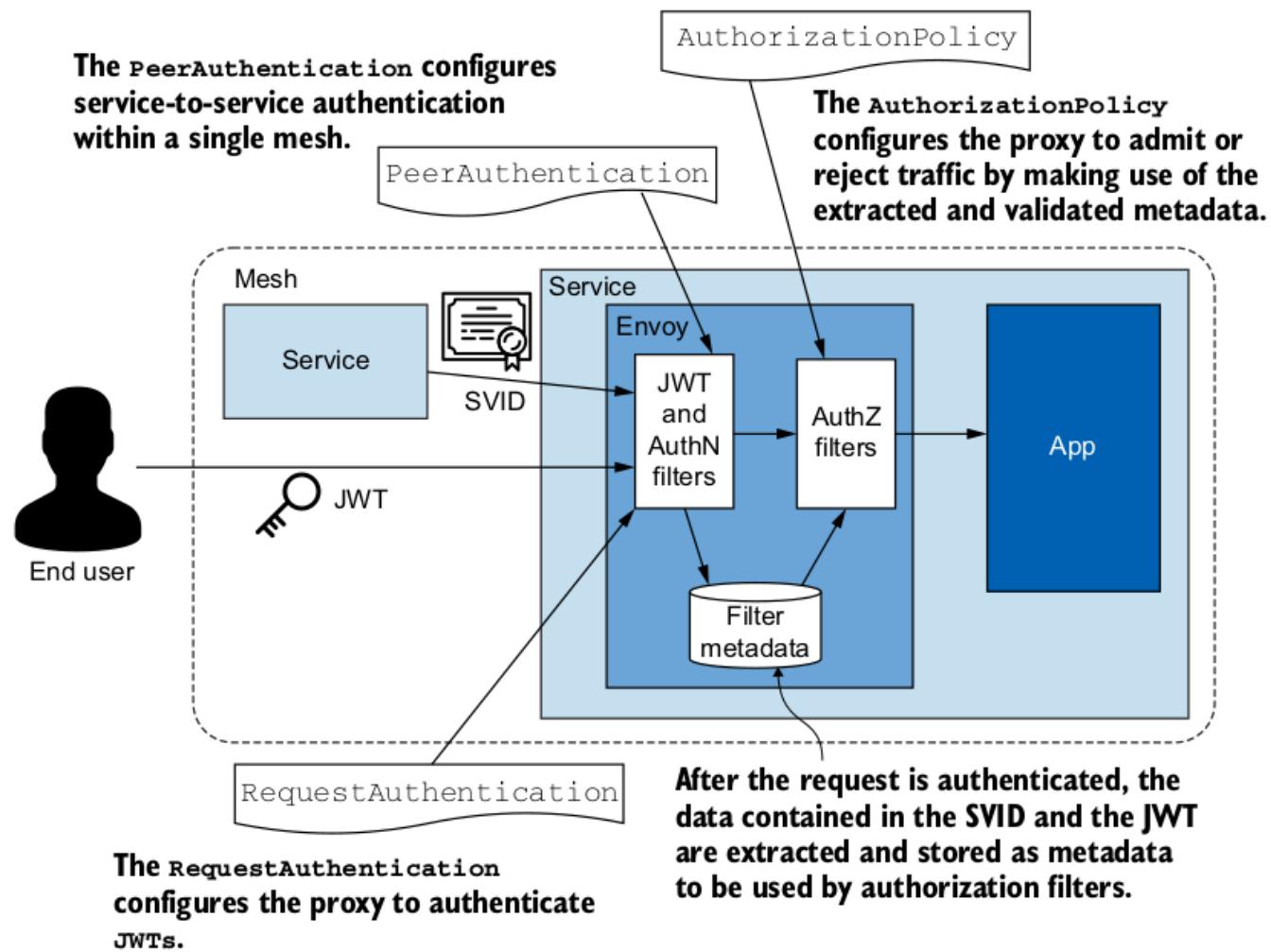
Cette identité est encodée dans un certificat X.509 nommée SVID², que le plan de contrôle d'Istio crée pour les charges de travail.



Ressources

L'exploitant Istio utilise différentes ressources pour la sécurité :

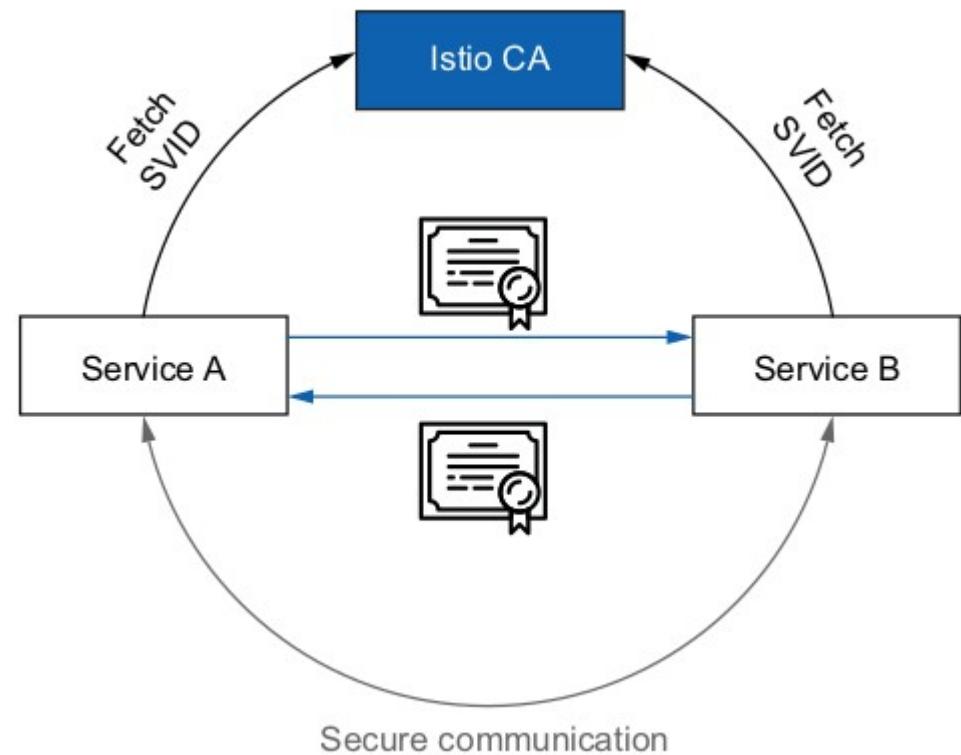
- **PeerAuthentication** : configure le proxy pour authentifier le trafic de service à service. En cas de succès, le proxy met à disposition les informations d'identification pour autoriser la requête.
- **RequestAuthentication** : configure le proxy pour authentifier l'utilisateur final. En cas de succès, le proxy met à disposition les informations d'identification utilisateur pour autoriser la requête.
- **AuthorizationPolicy** : configure le proxy pour autoriser ou rejeter les requêtes en fonction des informations mises à disposition.

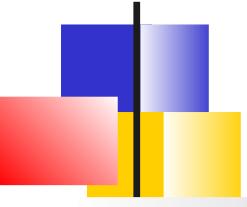


Automatique *mTLS*

Le trafic entre les services est chiffré et les 2 proxys sont mutuellement authentifiés.

Un processus automatique génère les certificats et les renouvelle



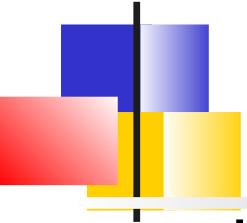


Sécurisation

L'authentification mutuelle ne suffit pas à sécuriser le réseau.

2 actions à faire :

- Interdire le trafic non authentifié
- Définir les permissions afin de ne fournir que les accès dont les services ont besoin

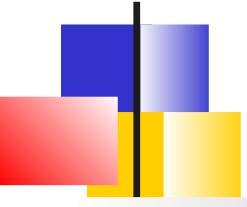


Identification et Chiffrement entre services

La ressource *PeerAuthentication* permet de configurer le trafic autorisé entre services : STRICT (mTLS exclusivement) ou PERMISSIVE

Comme d'habitude, ceci peut être configuré au niveau global, espace de nom ou workload

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
```

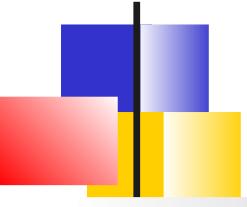


Autorisation entre services

AuthorizationPolicy définit les permissions au niveau global, namespace ou workload

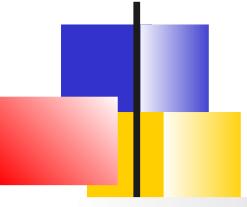
3 champs doivent être configurés pour une stratégie d'autorisation :

- **selector** définit le sous-ensemble de workloads pour lequel la stratégie s'applique
- **action** spécifie ALLOW , DENY , ou CUSTOM
Si une ou plusieurs stratégies ALLOW sont appliquées, l'accès à la workload est rejeté pour tout trafic par défaut
- **rules** : Une liste de règles pour identifier la requête . La stratégie d'autorisation est appliquée si une des règles est vérifiée.



Exemple

```
kind: "AuthorizationPolicy"
...
spec:
  selector:
    matchLabels:
      app: webapp
  action: ALLOW
  rules:
    - to:
        - operation:
            paths: ["/api/catalog*"]
```

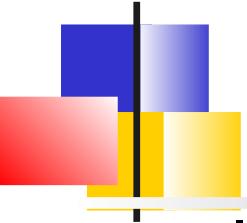


Rule

Les règles spécifient la source de la connexion et (éventuellement) l'opération lorsque les 2 correspondent, la règle est activée

Les champs d'une règle sont :

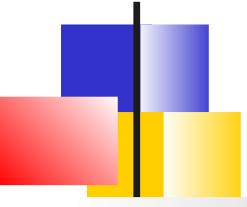
- **from** : spécifie la source de la requête.
 - **principals** : Une liste d'identité SPIFFE
 - **namespaces** : Une liste d'espaces de nom
 - **ipBlocks** : Une liste d'adresse Ips ou d'intervalle
- **to** : spécifie les opérations de la requête comme le host ou la méthode
- **when** : Une liste de conditions devant être remplies



Expression des règles

Istio supporte différents types d'expression des règles :

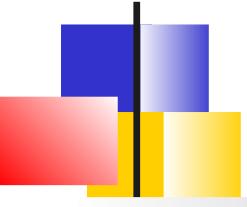
- Correspondance exacte.
Par exemple, *GET*
- Correspondance de préfixe.
Par exemple, */api/catalog**
- Correspondance de suffixe
Par exemple, **.formation.io*
- Correspondance de présence, qui correspond à toutes les valeurs et est indiquée par ***. Le champ doit être présent quelque soit sa valeur



Activation d'une règle

Afin qu'une règle s'active, La requête doit correspondre :

- à une des sources listées par la clause *from*
- ET à 1 des opérations listées dans la clause *to*
- ET à TOUTES les conditions de *when*

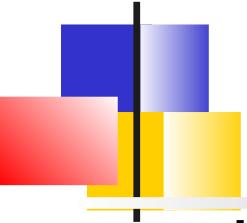


Deny catch-all

Il est recommandé d'ajouter une stratégie ***deny catch-all*** qui s'active lors qu'aucune règle s'active.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all          # Namespace d'installation
  namespace: istio-system  # => Toutes les workloads du mesh
spec: {}                  # Spec vide => Tout match
```

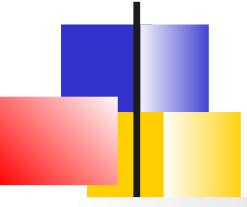
N.B Si on définit une règle vide, on obtient un allow all



Ordre d'évaluation

L'ordre d'évaluation des stratégies est :

- 1) Stratégies CUSTOM
- 2) Stratégies DENY
- 3) Stratégies ALLOW
- 4) Sinon :
 - 1) Si stratégie de *catch-all* est présente, elle détermine si la requête est approuvée
 - 2) Si pas de stratégie de catch-all, la requête est :
 - 1) Autorisée si il n'y pas de stratégies ALLOW
 - 2) Rejetée si il y a des stratégies ALLOW mais aucune ne match

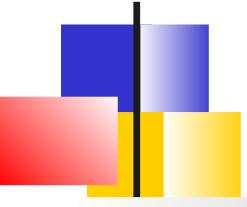


Espace de nom

Généralement on veut autoriser le trafic pour tous les services appartenant au même espace de nom.

Cela peut être effectué via la propriété ***source.namespace***

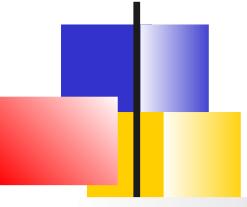
```
kind: "AuthorizationPolicy"
metadata:
  name: "webapp-allow-view-default-ns"
  namespace: formation
spec:
  rules:
    - from:
        - source:
            namespaces: ["default"]
      to:
        - operation:
            methods: ["GET"]
```



Service non identifié

Pour permettre l'accès de service non identifié (pas de proxy Istio), il faut supprimer le champ *from*

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "webapp-allow-unauthenticated-view-default-ns"
  namespace: formation
spec:
  selector:
    matchLabels:
      app: webapp
  rules:
    - to:
        - operation:
            methods: ["GET"]
```

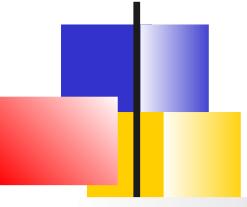


Conditions

Le champ **when** permet de faire une règles conditionnelles en fonction des attributs Istio (Entêtes HTTP, source IP, l'espace de nom, le principal, revendications JWT, etc.)¹

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-mesh-all-ops-admin"
  namespace: istio-system
spec:
  rules:
  - from:
    - source:
        requestPrincipals: ["auth@formation.io/*"] # Jeton issu
  when:
  - key: request.auth.claims[group]    # Attribut Istio, revendication JWT
    values: ["admin"]
```

1. <https://istio.io/latest/docs/reference/config/security/conditions>

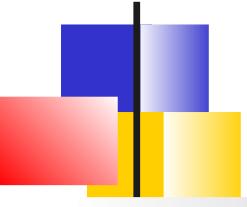


Identification utilisateur final

L'authentification et les autorisations liées à l'utilisateur final est supporté par *Istio* si on utilise JWT

Bien que l'autorisation de l'utilisateur final puisse être effectuée à n'importe quel niveau, les contrôles d'accès sont typiquement faits sur la gateway Istio.

La gateway supprime éventuellement le jeton ... pour ne pas le laisser traîner dans le mesh

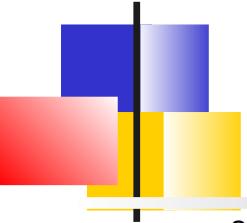


RequestAuthentication

La ressource ***RequestAuthentication*** sert à valider les jetons JWT, extraire les revendications des jetons valides et les stocker dans les méta-données de filtre afin que les stratégies d'autorisation puissent s'appuyer dessus.

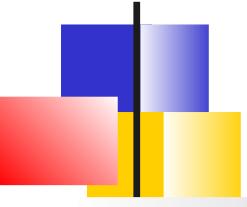
Les conséquence d'une ressource *RequestAuthentication* pour une requête :

- Jeton valide => revendications disponibles
- Jetons invalides (expiration, validation de signature) => requête rejetée
- Sans jetons => requête acceptée mais pas d'identification



Exemple

```
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "jwt-token-request-authn"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  jwtRules:
  - issuer: "auth@istioinaction.io"
    jwks: |
      { "keys": [ {"e": "AQAB", "kid": "CU-
ADJJEBH9bXl0tpsQWYuo4EwlkxFUHbeJ4ckkakCM", "kty": "RSA", "n": "z19VRDbmVvyXNdyoGJ5uhuTSRA265
3KHEi3XqITfJISvedYHVNGoZZxUCoiSEumxqrPY_Du7IMKzmT4bAuPnEalbY8rafuJNXnxVmqqjTrQovPIerkGW5h
59iUXIz6vCzn07F61RvJsUEyw5X291-3Z3r-9RcQD9sYy7-
8fTNmcXcdG_nNgYCnduZUJ3vFvhmQCwHFG1idwni8PJ09NH6aTZ3mN730S6Y1g_lJf0bju7lwYWT8j2Sjrwt6EES
55oGimkZHkktKjDYjRx1rN4dJ5PR5zh1Q4k0RWg1PtllWy1s5TSp0Uv840PjEohEo0WH0-
g238zI0YA83gozgbJfmQ"} ]}
```



Forcer JWT

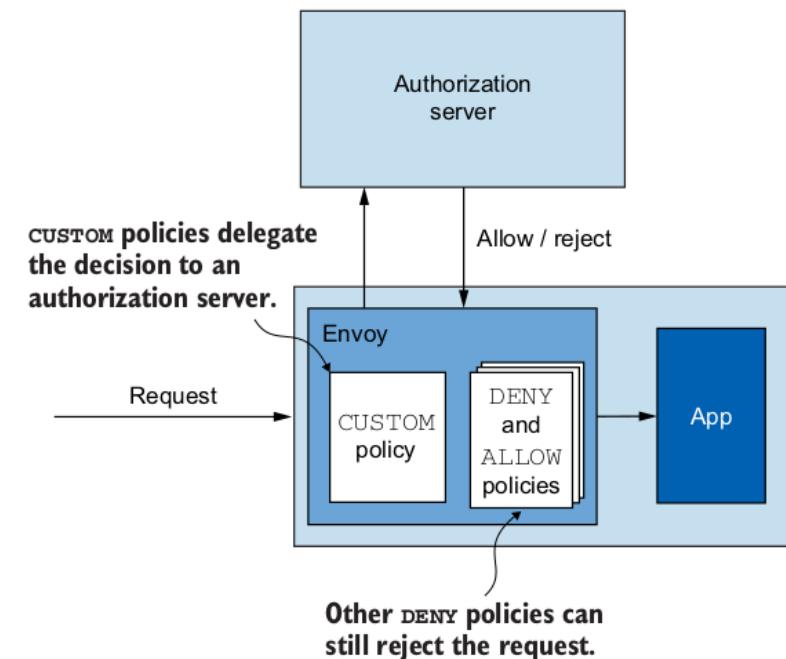
Pour interdire les requêtes sans jeton, il faut une ressource
AuthorizationPolicy

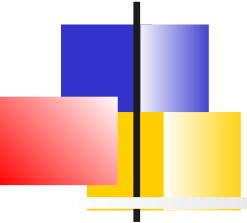
```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: app-gw-requires-jwt
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  action: DENY
  rules:
  - from:
    - source:
        notRequestPrincipals: ["*"]
  to:
  - operation:
    hosts: ["webapp.formation.io"]
```

Intégration avec un service d'autorisation externe

Il est possible de configurer le proxy *Istio* afin qu'il appelle un service d'autorisation externe pour valider l'accès à la ressource.

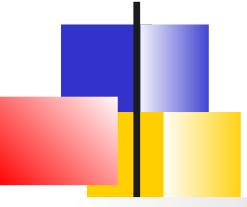
Le service d'autorisation peut être dans le mesh, comme sidecar de l'application ou même à l'extérieur du mesh.





Compléments

Installation personnalisée
Résolution de problèmes
Monitoring de istod

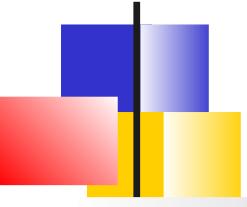


Introduction

L'installation d'Istio est plutôt simple, il suffit d'appliquer des ressources dans le cluster Kubernetes :

Différents moyens sont à disposition

- **helm** : La personnalisation est effectuée par des gabarits Helm
- **istioctl** : Expose une API plus simple et plus sûre pour installer et personnaliser Istio à l'aide de la définition de ressource personnalisée (CRD) *IstioOperator*¹
- **istio-operator** : Un opérateur s'exécutant côté cluster qui gère les Installations d'Istio à l'aide de l'API *IstioOperator*
- **kubectl**



IstioOperator

L'API **IstioOperator**¹ est un CRD Kubernetes qui spécifie l'état souhaité d'une installation Istio.

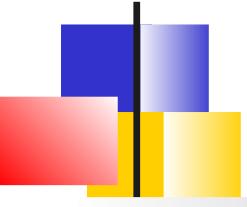
- Ensuite, *istioctl* et *istio-operator* font converger le système vers l'état souhaité

L'API offre 2 avantages :

- La validation des entrées utilisateur
- La documentation

Il existe cependant tellement de configurations qu'Istio met à disposition des profils d'installation

1. <http://mng.bz/PWXP>



Profils disponibles

default : Un point de départ pour les déploiements en production.

L'autoscaling est activé et davantage de ressources sont mises à la disposition d'istiod, des gateway et du proxy.

demo : Démonstration en local

empty : Point de départ pour une installation complètement personnalisée

external : Control plane externe

minimal : Idem default sans la gateway ingress

openshift : Profil défaut pour OpenShift

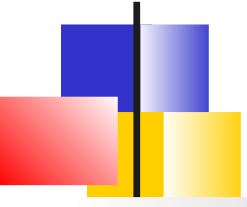
...

Pour visualiser les profils disponibles :

```
istioctl profile list
```

Pour visualiser la conf d'un profil :

```
istioctl profile dump demo
```



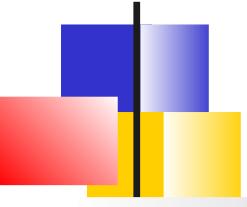
2 ressources *IstioOperator*

Une installation personnalisée consiste à personnaliser un profil de départ.

Il est recommandé de découpler les installations du control plane et du data plane

=> 2 ressources *IstioOperator*

Cela peut être fait par *istioctl* ou *istio-operator*



istio-operator

Un opérateur Kubernetes est un type de contrôleur Kubernetes qui expose la gestion d'un logiciel via des ressources personnalisées Kubernetes

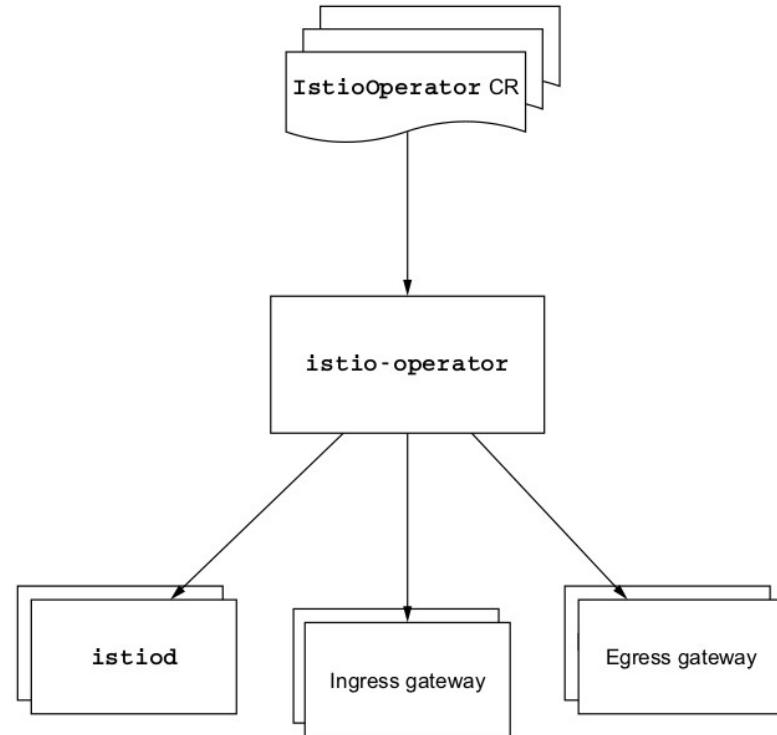
istio-operator gère les installations Istio dans un cluster et permet la personnalisation de l'installation via l'API *IstioOperator*

istio-operator est installé dans le cluster Kubernetes et s'identifie via son compte de service pour interagir avec l'API Kubernetes et surveiller les ressources de type *IstioOperator*

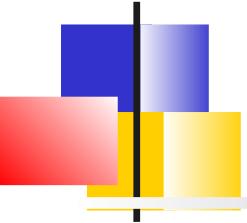
Si une ressource *IstioOperator* est modifiée, l'opérateur met à jour l'installation d'Istio

C'est une approche GitOps

=> les modifications apportées à Git sont propagées jusqu'au cluster via des pipelines CI/CD

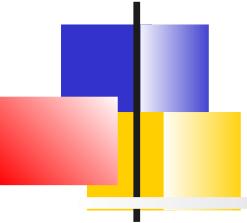


Atelier 3.2 Installation avec *istio-operator*



Compléments

Installation personnalisée
Résolution de problèmes
Monitoring de *istiod*

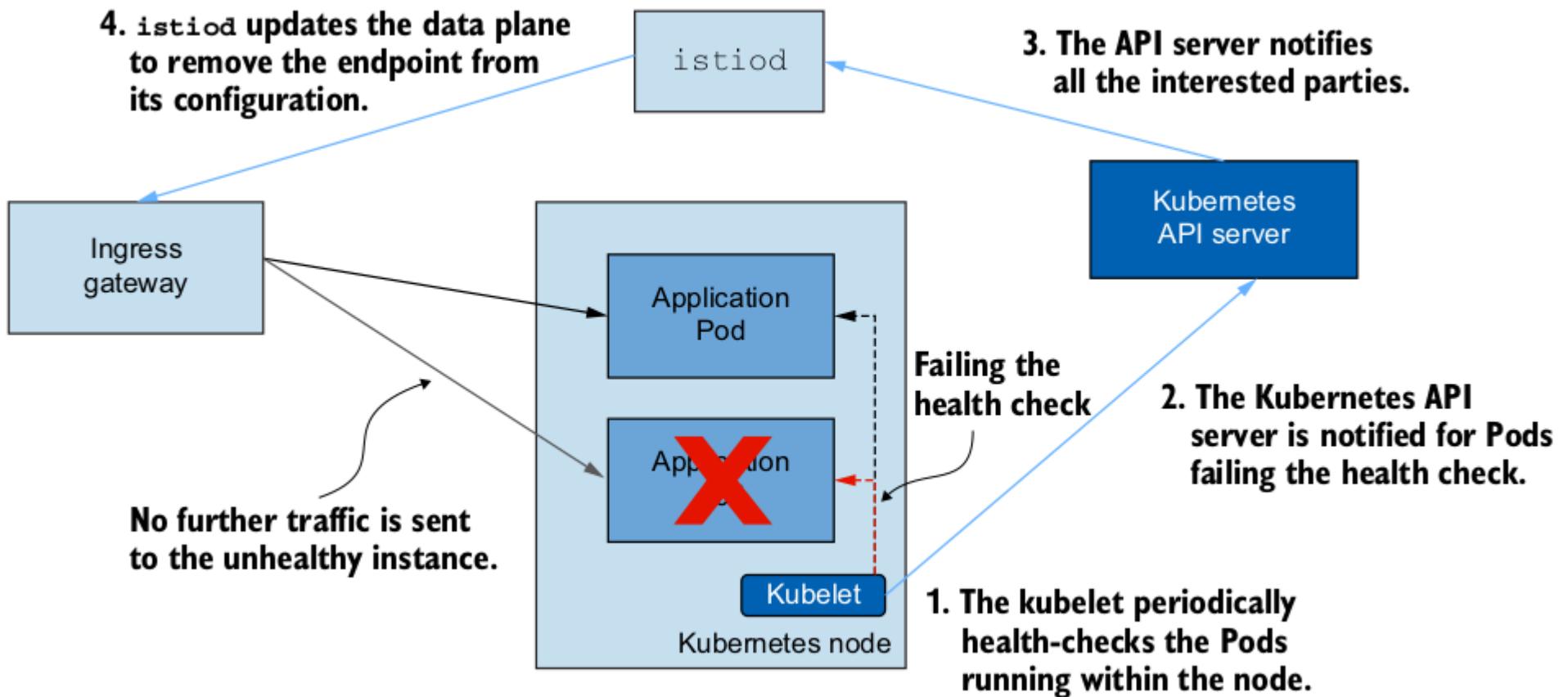


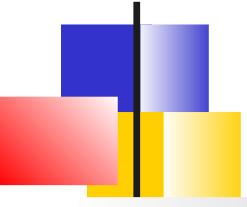
Identifier des problèmes de DataPlane

Considérant que la fonction première du *control plane* est de synchroniser le *DataPlane* avec la dernière configuration, la 1ère chose est de vérifier que la synchronisation s'est bien faite.

Dans de gros clusters, la synchronisation peut prendre du temps.

Propagation d'un service unhealthy jusqu'à la configuration



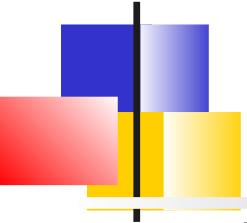


Etats de synchronisation

La synchronisation peut être contrôlée avec :
istioctl proxy-status

La commande affiche la liste des workloads et leur état de synchronisation pour les API xDS. 3 états possibles :

- **SYNCED** : Envoy a acquitté la dernière configuration envoyée
- **NOT SENT** : *istiod* n'a rien envoyé à Envoy. En général parce qu'il n'a rien à envoyer
- **STALE** : *istiod* a envoyé une mise à jour mais n'a pas été acquitté

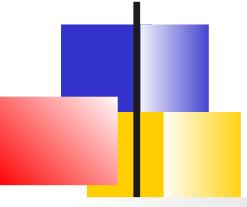


Kiali

Kiali est plutôt efficace pour détecter les erreurs.

- Des icônes avertissent en général d'un problème et permettent d'accéder à la configuration du *DataPlane*
- Chaque problème a un identifiant référencé¹

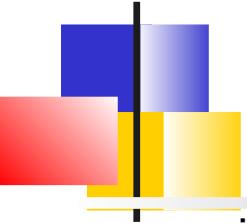
1. Voir <http://mng.bz/2jzX>



Commandes *Istioctl*

Pour la résolution de problèmes, *istioctl* propose les 2 commandes ***analyze*** et ***describe***

- *analyze* est un outil de diagnostic extensible qui peut s'appliquer à des clusters en cours d'exécution ou sur des configurations par encore appliquées
- *describe* affiche un résumé de la configuration d'une workload vis à vis d'Istio



Exemple *analyze*

```
istioctl analyze -n formation
```

```
Error [IST0101] (VirtualService catalog-v1-v2.formation)
```

- Referenced host+subset in destinationrule not found:
 - "catalog.istioinaction.svc.cluster.local+version-v1"

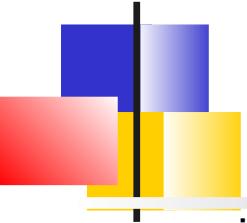
```
Error [IST0101] (VirtualService catalog-v1-v2.formation)
```

- Referenced host+subset in destinationrule not found:
 - "catalog.formation.svc.cluster.local+version-v2"

```
Error: Analyzers found issues when analyzing namespace: formation.
```

```
See https://istio.io/v1.13/docs/reference/config/analysis
```

- for more information about causes and resolutions.



Exemple *describe*

```
istioctl x describe pod catalog-68666d4988-vqhmb
```

```
Pod: catalog-68666d4988-q6w42
```

```
Pod Ports: 3000 (catalog), 15090 (istio-proxy)
```

```
-----
```

```
Service: catalog
```

```
Port: http 80/HTTP targets pod port 3000
```

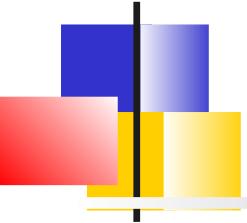
```
Exposed on Ingress Gateway http://13.91.21.16
```

```
VirtualService: catalog-v1-v2
```

```
WARNING: No destinations match pod subsets (checked 1 HTTP routes)
```

```
Warning: Route to subset version-v1 but NO DESTINATION RULE defining subsets!
```

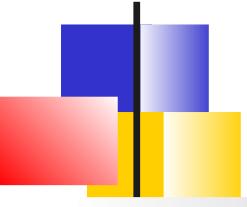
```
Warning: Route to subset version-v2 but NO DESTINATION RULE defining subsets!
```



Directement sur la config Envoy

Si les précédentes investigations ne donnent rien, il est possible de récupérer la configuration directement *d'Envoy*

- Via l'interface d'administration d'Envoy
- Via *istioctl*



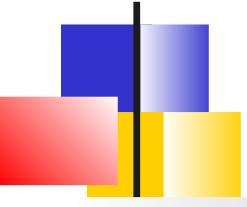
Interface d'administration Envoy

L'interface d'administration d'Envoy expose la configuration d'Envoy + d'autres fonctionnalités permettant de modifier certains aspects du proxy, comme le niveau de log.

Cet interface est accessible pour chaque proxy de service sur le port 15000.

```
istioctl dashboard envoy deploy/catalog -n istioinaction
```

Command	Description
certs	print certs on machine
clusters	upstream cluster status
config_dump ←	dump current Envoy configs (experimental)
contention	dump current Envoy mutex contention stats (if enabled)

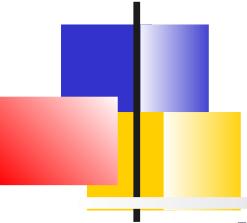


istioctl

istioctl fournit des outils pour filtrer le résultat d'un *config_dump* effectué avec l'administration d'Envoy

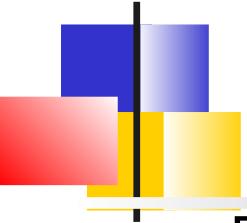
istioctl proxy-config permet de filtrer la configuration du proxy, il propose les sous-commandes :

- *cluster*
- *endpoint*
- *listener*
- *route*
- *secret*



Problèmes applicatifs

Les traces et les métriques générées par le proxy peuvent aider à la résolution de problèmes applicatifs.



Traces

Envoy enregistre toutes les requêtes traitées par le proxy si :

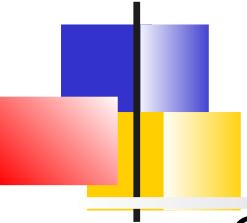
```
istioctl install --set  
  meshConfig.accessLogFile="/dev/stdout"
```

Le format par défaut est TEXT. En général, le format JSON est plus lisible

```
istioctl install --set profile=demo \  
--set meshConfig.accessLogEncoding="JSON"
```

A la recherche d'une 504 :

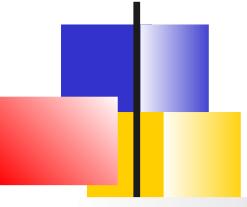
```
kubectl -n istio-system logs deploy/istio-  
ingressgateway \  
| grep 504
```



Format pour une requête

Chaque requête contient un nombre de champs dont les plus importants sont :

- ***Response_Code*** : Code du protocole
- ***response_flags¹*** :
 - UT : Le service était lent vis à vis de la configuration du timeout
 - UH : Aucune workload
 - NR : Pas de route configurée
 - ...
- ***upstream_cluster*** : Le cluster que l'on essaie d'atteindre
- ***upstream_host*** : L'hôte que l'on essaie d'atteindre
- ***downstream_remote_address*** : D'où la requête vient
- ***duration*** : Durée de la requête
- ***request_id*** : Id de la requête

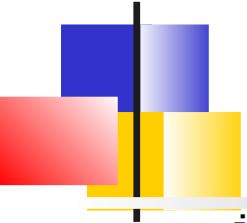


Niveau de trace

istioctl permet de lire et modifier les niveaux de trace des différents loggers

Les niveaux disponibles sont :

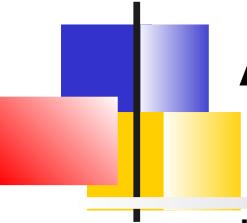
- none
- error
- warning
- info
- debug



Visualiser les niveaux de trace

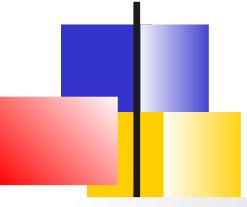
```
istioctl proxy-config log \
deploy/istio-ingressgateway -n istio-system
```

```
active loggers:
  connection: warning          # détails TCP
  conn_handler: warning
  filter: warning
  http: warning                # Layer 7
  http2: warning
  jwt: warning
  pool: warning                # Pool de connexion
  router: warning              # Routing
  stats: warning
```



Augmenter le niveau de trace

```
istioctl proxy-config log deploy/istio-ingressgateway \
-n istio-system \
--level http:debug,router:debug,connection:debug,pool:debug
```



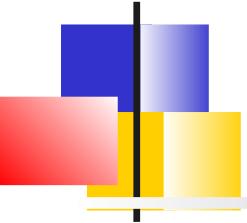
Grafana/Prometheus

Les tableaux de bord Grafana permettent de détecter des dysfonctionnement applicatifs :

- *Client Success Rate* (Non 5xx) est un bon indicateur même si il s'applique globalement à un service

Les requêtes Prometheus permettent d'être encore plus fin

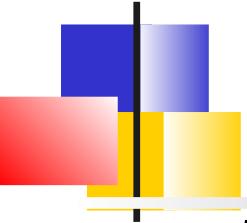
```
sort_desc(sum(irate(  
    istio_requests_total{  
        reporter="destination",  
        destination_service=~"catalog.istioinaction.svc.cluster.local",  
        response_flags="DC"}[5m]))  
by (response_code, kubernetes_pod_name, version))
```



Compléments

Installation personnalisée
Résolution de problèmes

Monitoring de istod



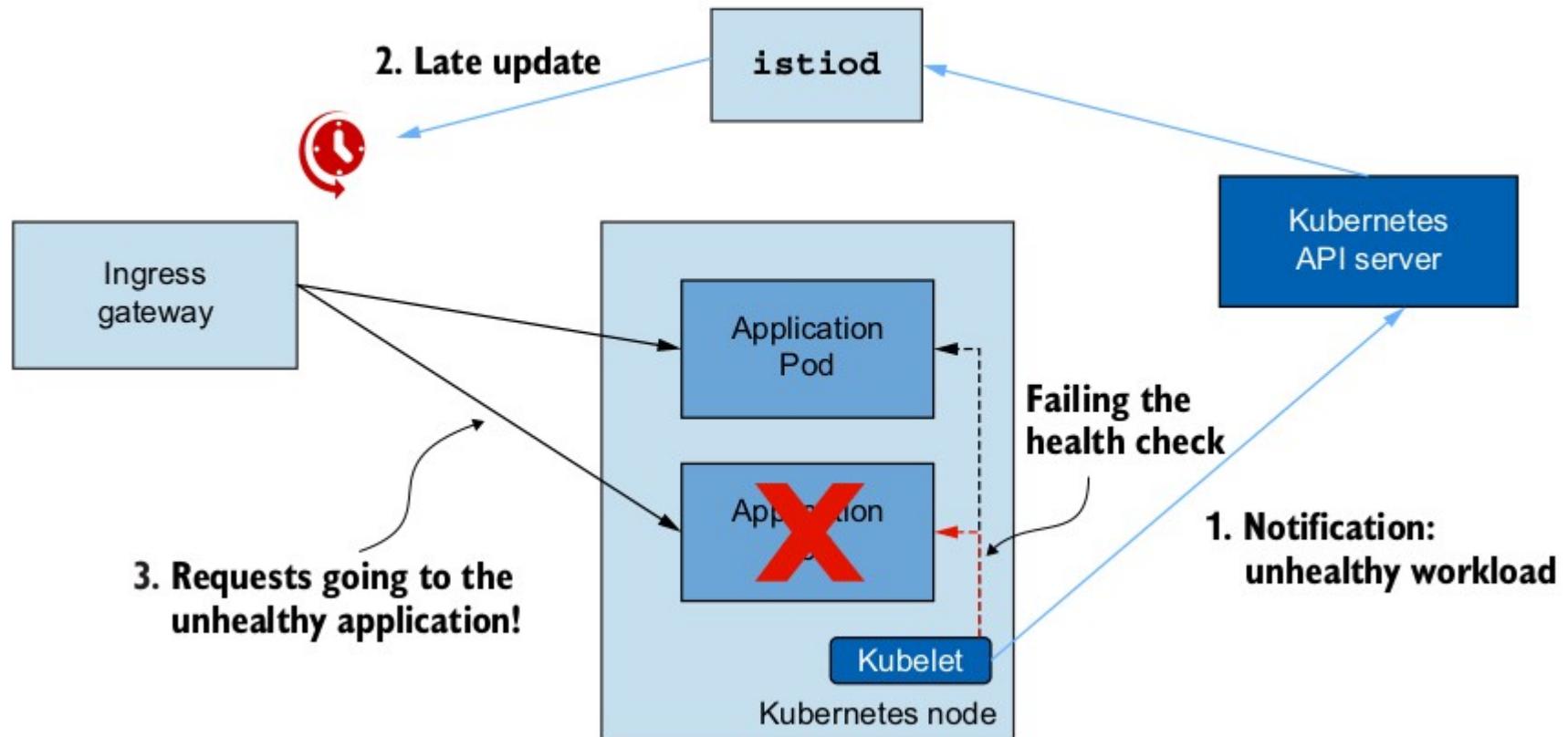
Introduction

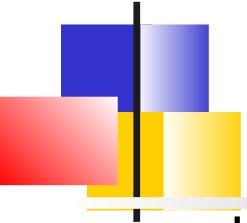
Istiod est le cerveau du service mesh

- Si il est affecté par des problèmes de performance cela a des conséquences sur tous les services
Par exemple, le routing de requêtes vers des workloads ayant disparues

Istiod écoute les évènements Kubernetes et met à jour les configurations pour refléter les changements

Illustration



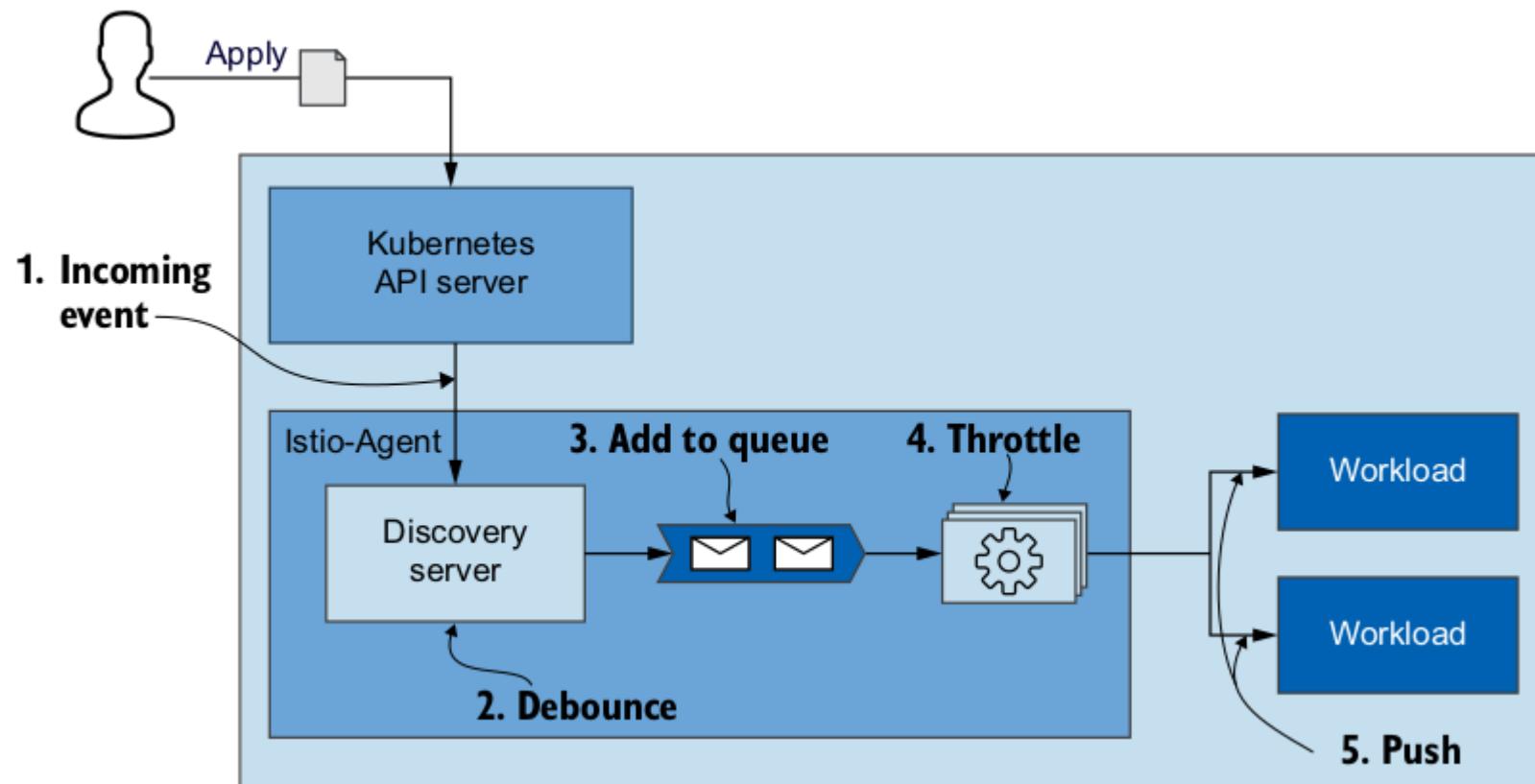


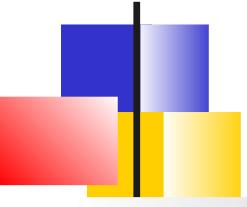
Séquence de synchronisation

La séquence d'étapes pour synchroniser le proxy vis à vis d'un changement est le suivant :

- 1) Un événement entrant déclenche le processus de synchronisation.
- 2) Le composant DiscoveryServer d'istiod écoute ces événements. Pour améliorer les performances, il retarde l'ajout de l'événement à la file d'attente d'envoi pendant une durée définie pour regrouper et fusionner les événements suivants pour cette période. (debouncing)
- 3) Une fois le délai expiré, il ajoute les événements fusionnés à la file d'attente push, qui gère une liste des push en attente de traitement.
- 4) Le serveur istiod limite (throttling) le nombre de requêtes push qui sont traitées simultanément, ce qui garantit une progression plus rapide des éléments en cours de traitement.
- 5) Les éléments traités sont convertis en configuration Envoy et poussés vers les charges de travail.

Illustration



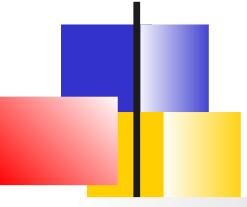


Facteurs

Les facteurs qui affectent les performances sont donc :

- La cadence des modifications : une cadence élevée nécessite davantage de traitement.
- Les ressources allouées : si la demande dépasse les ressources allouées à *istiod*, le travail doit être mis en file d'attente.
- Le nombre de workloads à mettre à jour : influe sur la temps CPU et de bande passante nécessaires
- La taille de la configuration : influe sur la temps CPU et de bande passante nécessaires

Les tableaux de bord Grafana peuvent permettre d'identifier les goulets d'étranglement



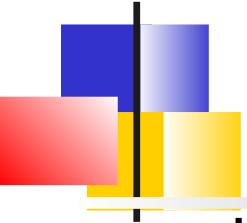
Gold Signals

Les 4 signaux à surveiller pour évaluer la performance d'un service sont :

- La **latence** : Le temps nécessaire pour mettre à jour la configuration
- La **saturation** : Comment les ressources sont utilisées
- Les **erreurs** : Taux d'erreur dans *istiod*
- Le **trafic** : Quelle est la charge sur *istiod*

Ils sont accessibles avec :

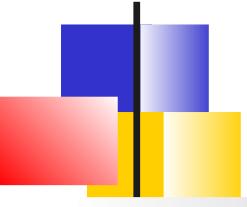
```
kubectl exec -it -n istio-system deploy/istiod -- curl localhost:15014/metrics
```



Latence

Les métriques à surveiller :

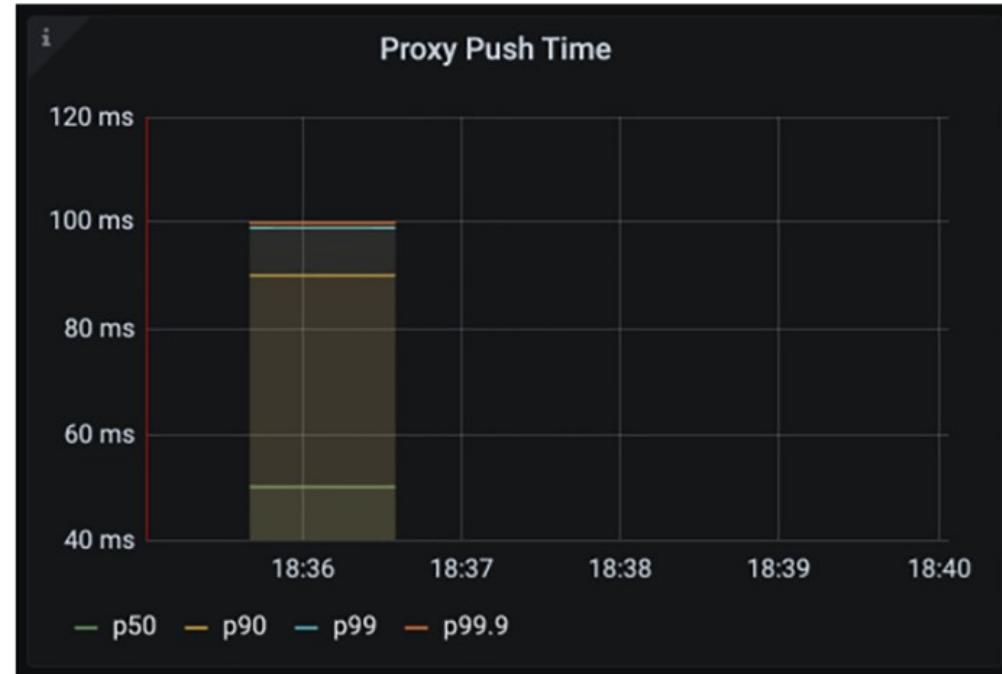
- **pilot_proxy_convergence_time** mesure la durée entre l'arrivée d'une requête push dans la file d'attente et sa distribution aux workloads.
- **pilot_proxy_queue_time** mesure le temps d'attente des requêtes push dans la file d'attente.
=> Si pb, Scaling vertical de *istiod*
- **pilot_xds_push_time** mesure le temps nécessaire pour pousser la configuration Envoy vers les charges de travail.
=> Si pb, cela montre une bande passante surchargée.
=> Réduire la taille et la fréquence des changements.

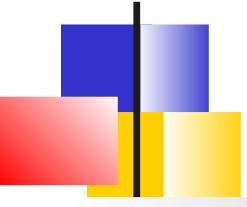


Grafana : Proxy Push Time

Dans Grafana, *pilot_proxy_convergence_time* est visualisé dans

Istio Control Plane Dashboard -> Pilot Push Information



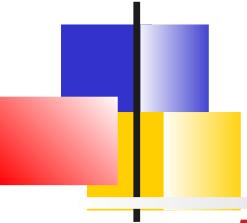


Saturation

La saturation est généralement causée par le CPU.

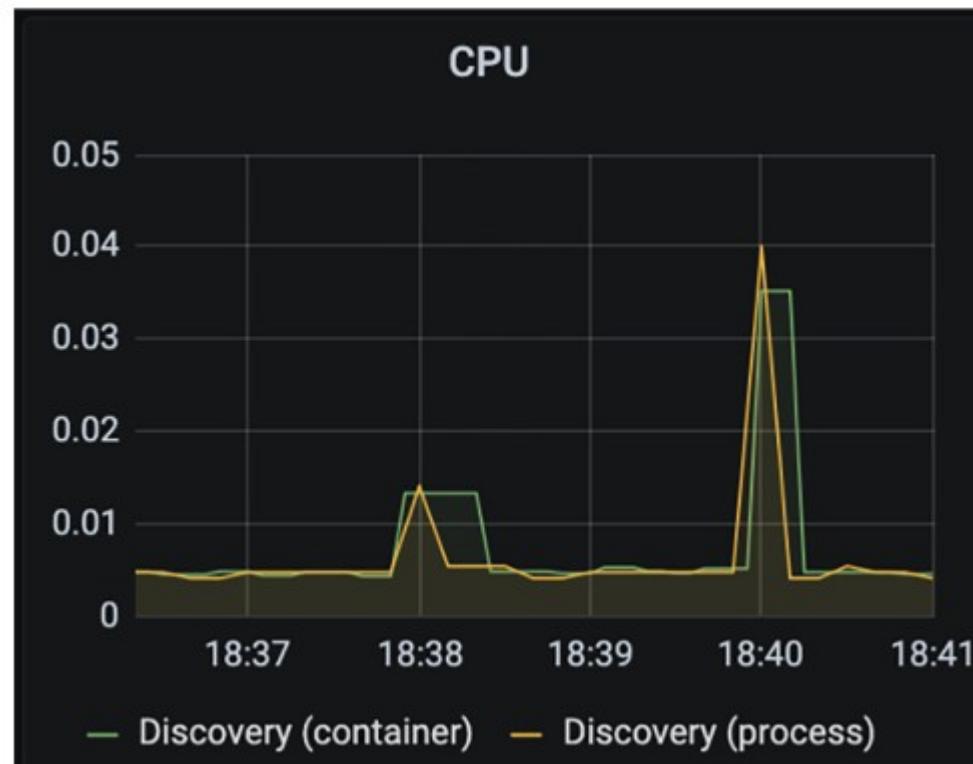
Les métriques sont :

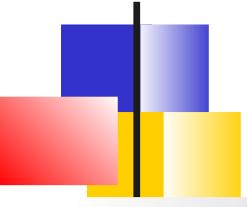
- ***container_cpu_usage_seconds_total*** : Mesure Kubernetes.
- ***process_cpu_seconds_total*** : Mesure via l'Instrumentation istiod.



Grafana : CPU

Istio Control Plane → Resource Usage





Trafic

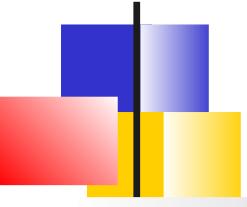
istiod reçoit du trafic entrant et produit du trafic sortant

Les métriques pour l'entrant sont :

- **pilot_inbound_updates** : Nbre de mises à jour reçues.
- **pilot_push_triggers**: Nbre d'événements ayant déclenché un push. (type : service , endpoint ou config)
- **pilot_services** : Nbre de services connus

Les métriques pour le sortant :

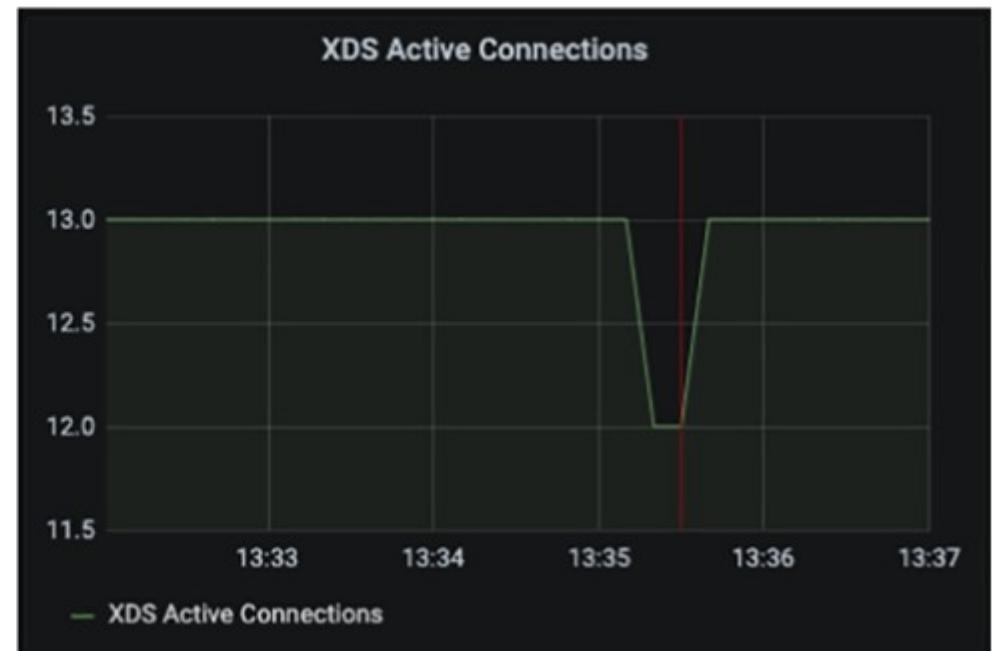
- **pilot_xds_pushes** mesure tous les push effectués .
- **pilot_xds** : Nbre total de connexions aux workloads.
- **envoy_cluster_upstream_cx_tx_bytes_total** : Taille de la configuration transférée sur le réseau.

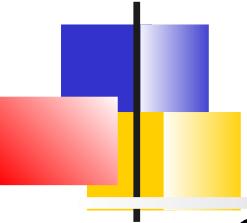


Grafana : output

Istio Control Plane → Pilot Pushes

Istio Control Plane → ADS Monitoring





Erreurs

Concernant les erreurs, les métriques à surveiller :

- **pilot_total_xds_rejects** : push rejetés
 - *pilot_xds_eds_reject,*
 - *pilot_xds_ids_reject,*
 - *pilot_xds_rds_reject,*
 - *pilot_xds_cds_reject*
- **pilot_xds_write_timeout** : Somme des erreurs et des timeouts lors d'un push
- **pilot_xds_push_context_errors** : Les erreurs lors de la génération de la config

Ces infos sont également disponible dans Grafana

Istio Control Plane → Pilot Errors