



Istio : Maillage de Services sur Kubernetes

David THIBAU – 2023

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
- Installation
- Fonctionnalités
- Le proxy Envoy

Istio in Action

- Gateways
- Routing
- Résilience
- Observabilité
- Sécurité

Autres

- Installation personnalisée
- Résolution de problèmes
- Monitoring de istod



Introduction

Architectures micro-services

Services transverses

Convictions d'Istio



Introduction

Le terme « ***micro-services*** » décrit un nouveau 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
- Développés par des équipes full-stack indépendantes.

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



Caractéristiques

Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

DURS (Deploy, Update, Replace, Scale) indépendants : Chaque service peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket,



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...)

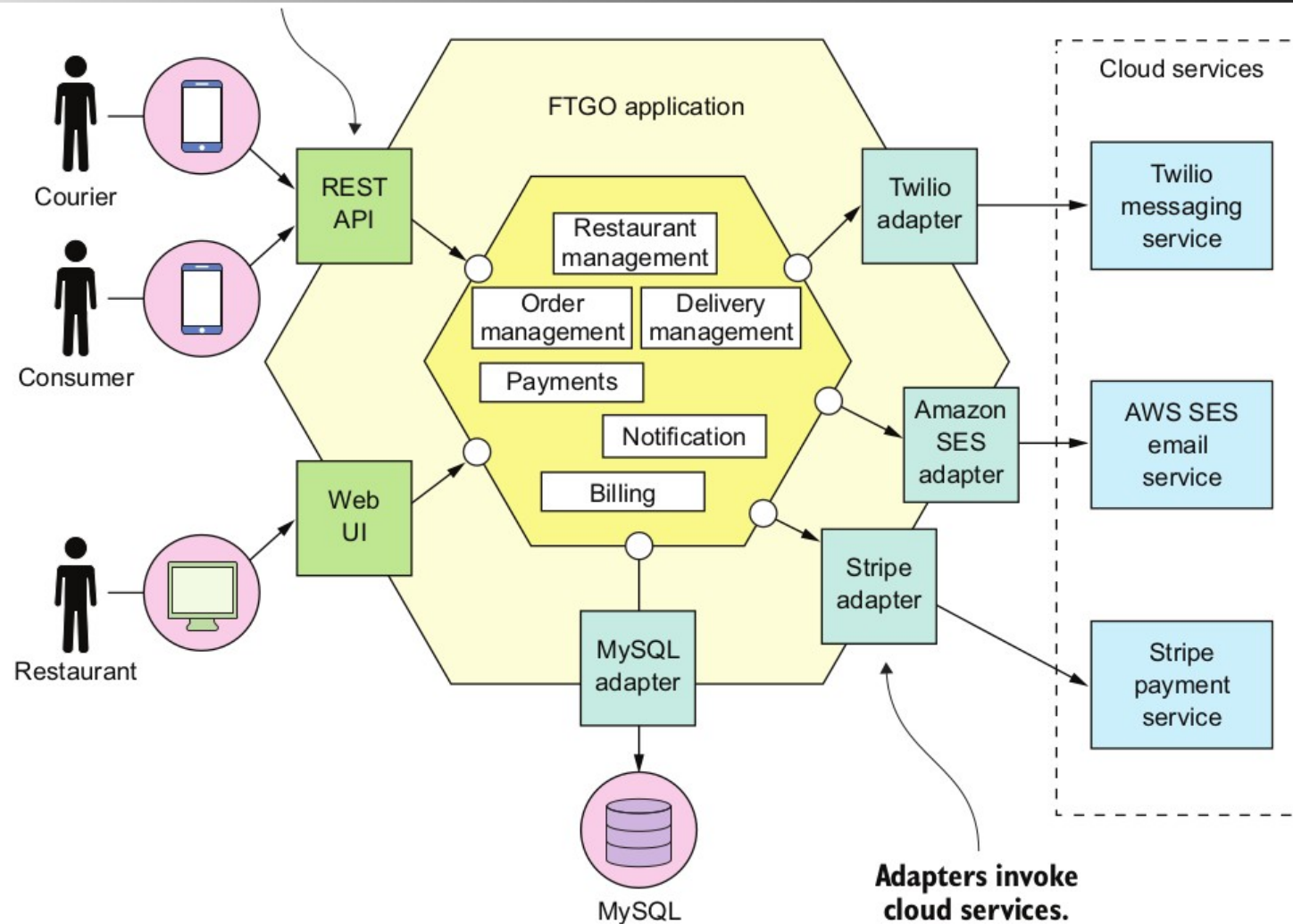
Découverte automatique : Les services sont typiquement distribués dans l'environnement d'exécution, le scaling peut être automatisé.
Les points d'accès doivent pouvoir être localisés automatiquement

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

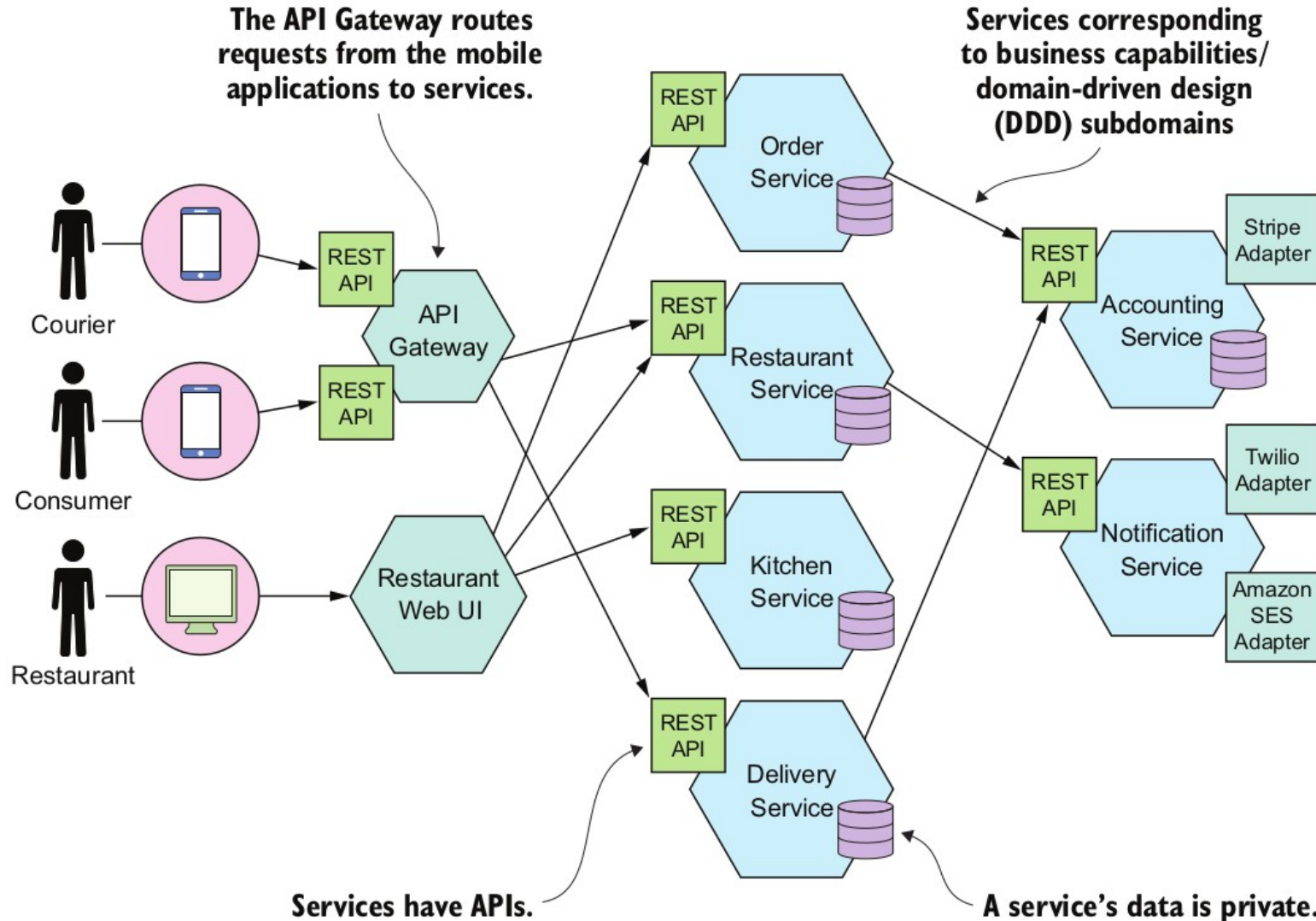
Résilience : Plus de services peuvent être en erreur.
L'application doit pouvoir résister aux erreurs.

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

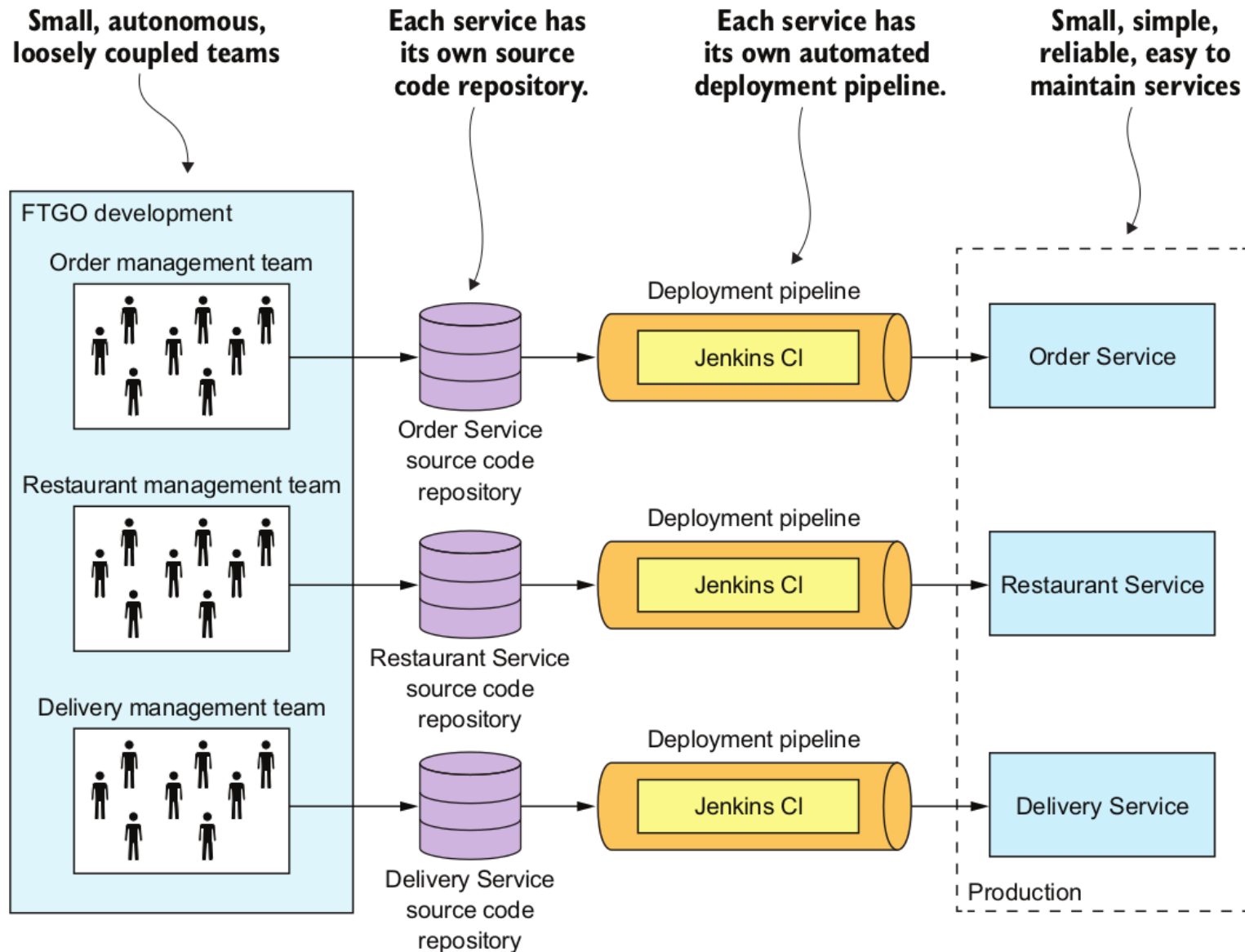
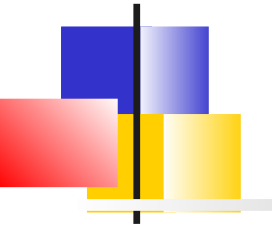
Architecture monolithique Hexagonale

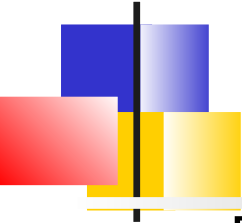


Une architecture micro-service



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 :

- Hôtes uniques avec différents processus
- 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 :

- Jetons d'accès, *oAuth*, ...



Les 12 facteurs de réussite

- I. Outil de scm** : Unique source de vérité
- II. Dépendances** : Déclare et isoler les dépendances
- III. Configuration** : Configuration séparée du code, stockée dans l'environnement
- IV. Services** d'appui (backend) : Considère les services d'appui comme des ressources attachées,
- V. Build, release, run** : Permet la coexistence de différentes releases en production
- VI. Processes** : Exécute l'application comme un ou plusieurs processus stateless.
Déploiement immuable
- VII. Port binding** : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP
- VIII. Concurrency** : Montée en charge grâce au modèle de processus
- IX. Disposability** : Renforce la robustesse avec des démarrages et arrêts rapides
- X. Dev/prod parity** : Garder les environnements de développement, de pré-production et de production aussi similaires que possible
- XI. Logs** : Traiter les traces comme un flux d'événements
- XII. Processus d'Admin** : Considérer les tâches d'administration comme un processus parmi d'autres



Introduction

Architectures micro-services
Services transverses
L'approche Istio



Services Transverses

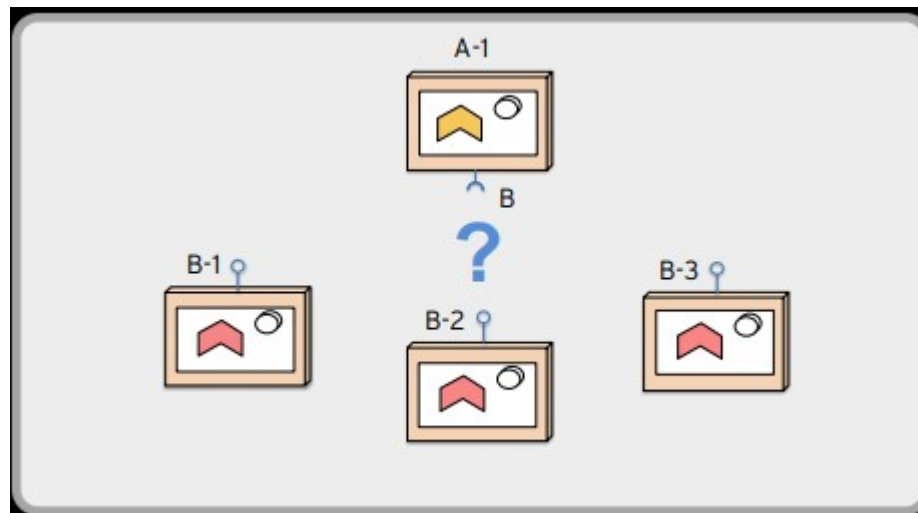
Les architectures distribuées doivent s'appuyer sur des services transverses.

L'implémentation pouvant être un framework ou l'infrastructure

Discovery

La réplication nécessite un service de découverte :

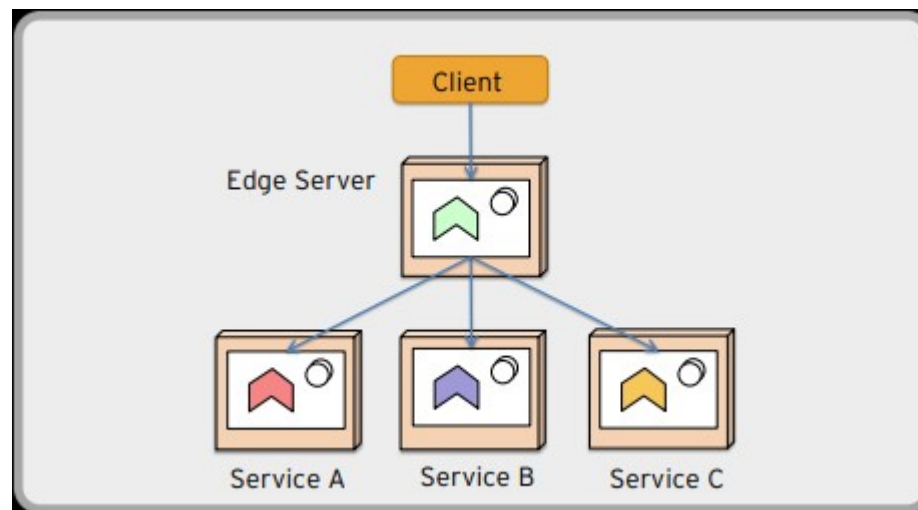
- Ou sont les services
- Quelles répliques appeler (Load Balancing)



Edge server

Définir les frontières du cluster

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

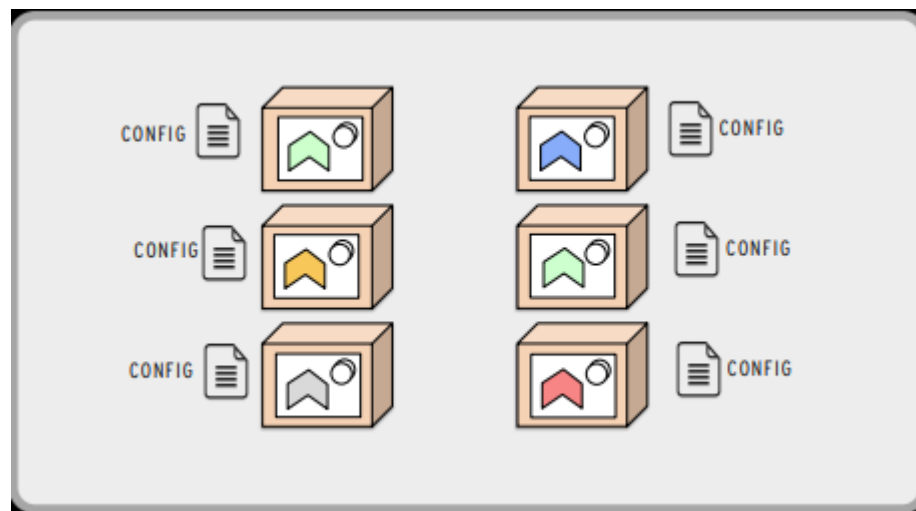




Configuration centralisée

Configuration centralisée

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





- Ou réside t elles ?
- Comment corréler les traces des différents services ?

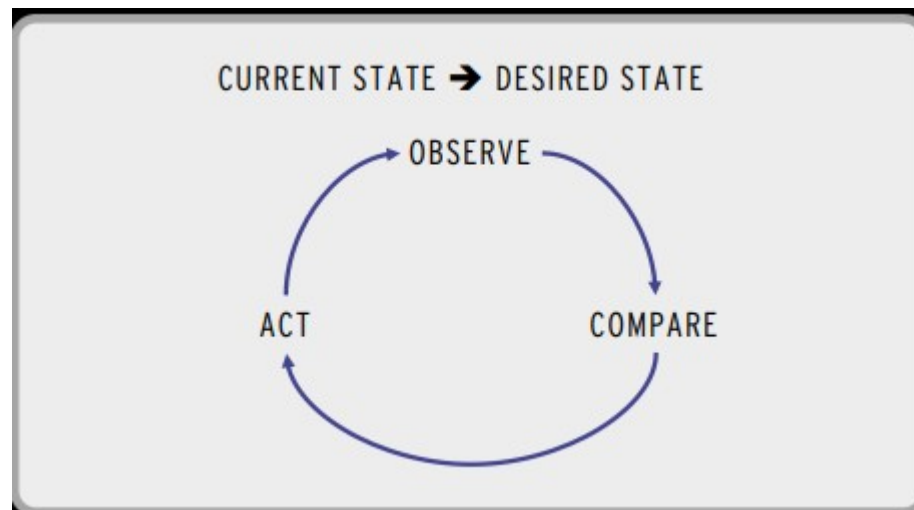




Gestion des services

Comment :

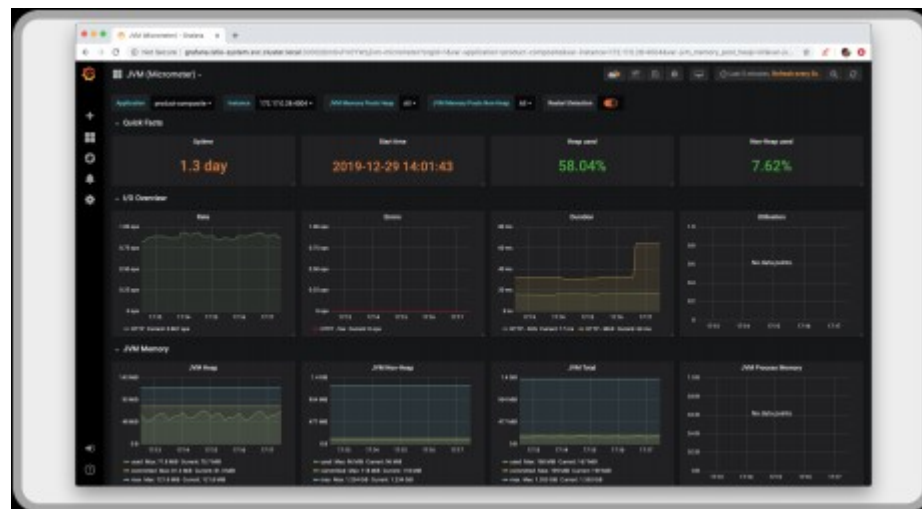
- Déployer les service
- Les scaler
- Les mettre à jour
- Redémarrer les services défaillants



Observabilité

Observer pour comprendre, résoudre les problèmes, optimiser

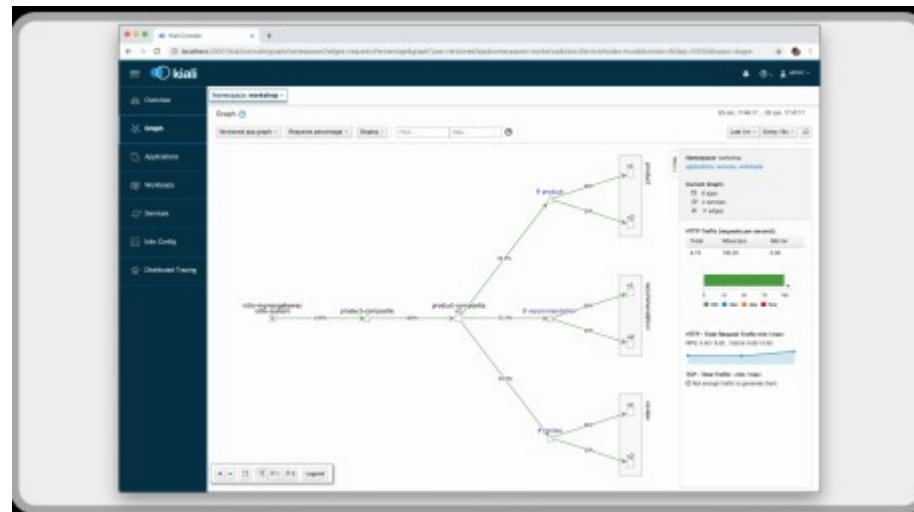
- Détecter les défaillances
- Quelles ressources sont utilisées ?
- Quel est l'usage des service ?
- Les services sont-ils performant



Gestion de trafic

Comment contrôler le routing ?

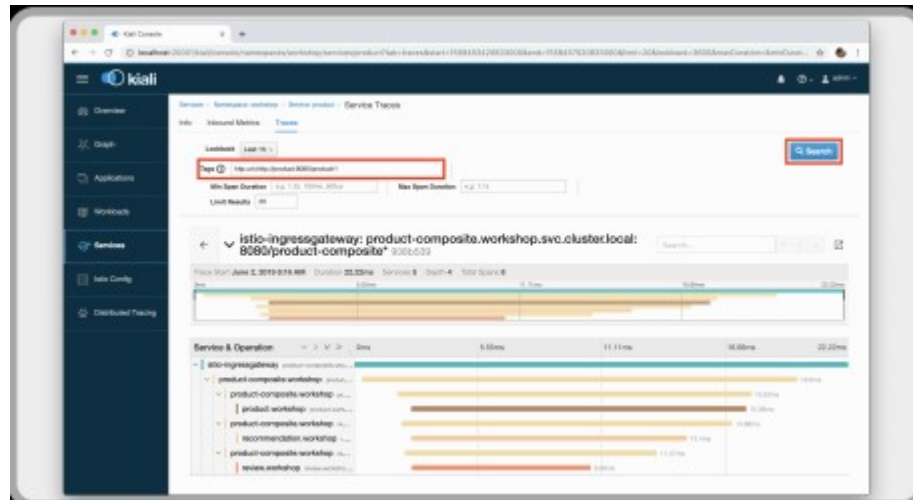
- Répartir la charge
- Limiter pour protéger
- Déploiement Blue/Green ou Canary



Tracing distribué

Qui appelle qui ?

- Suivre une requête dans l'architecture

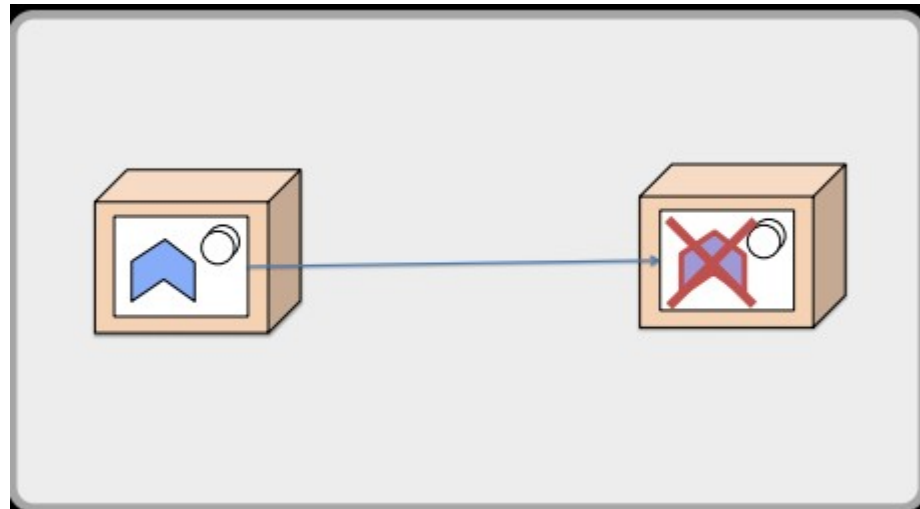




Résilience

Comment gérer les défaillances inévitables ?

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



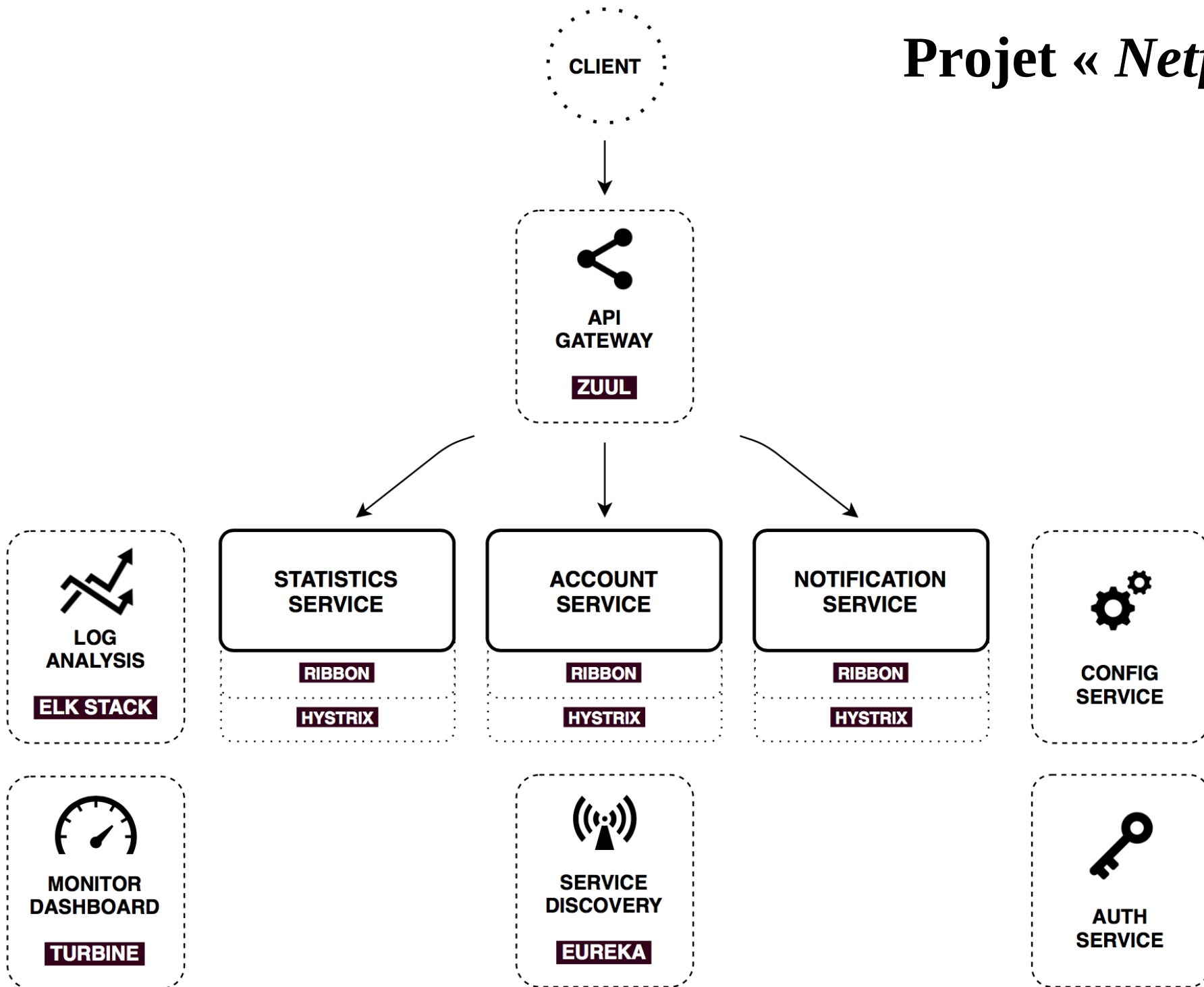


Services techniques vs Infra

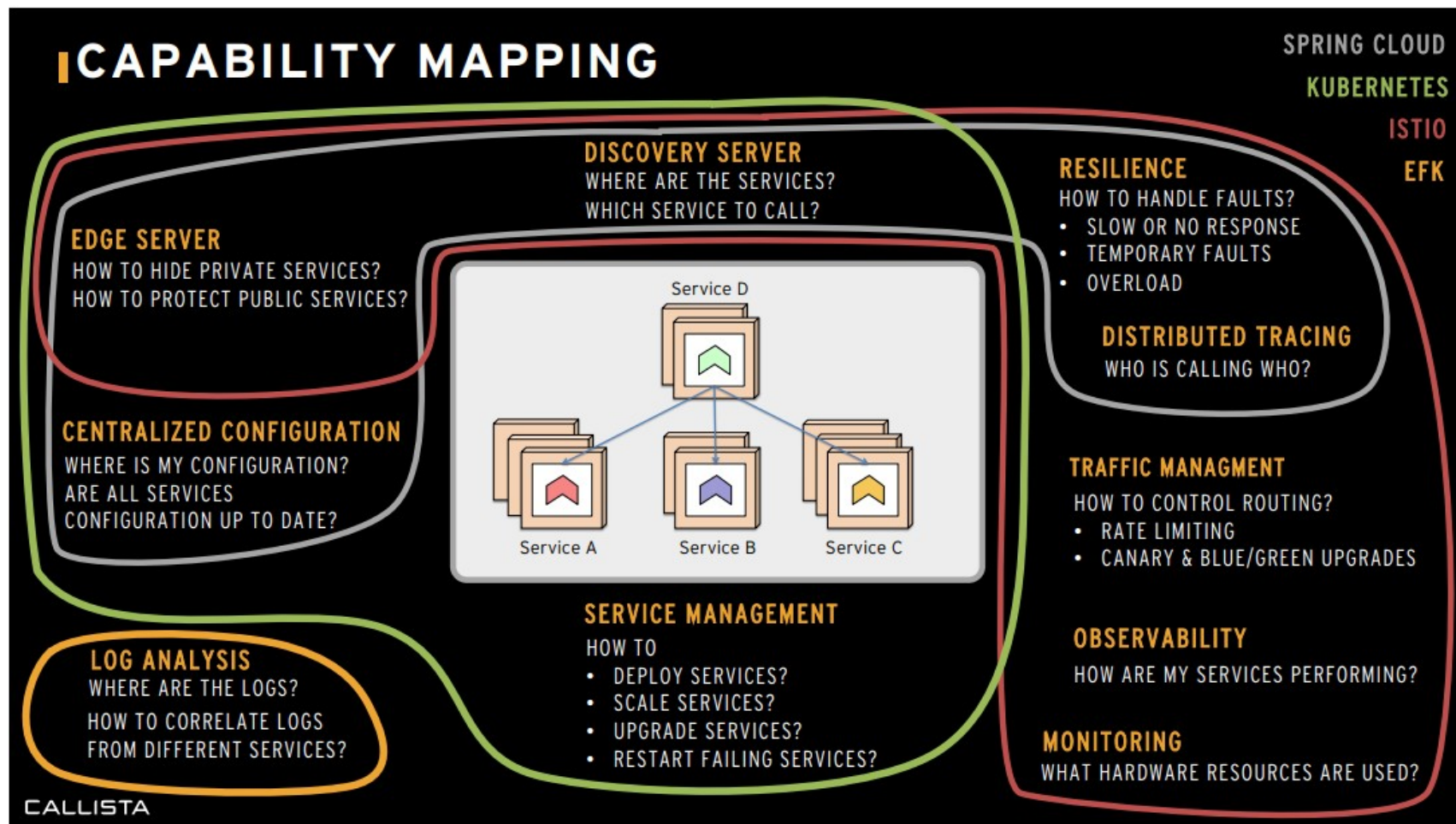
Qui fournit les services techniques ?

- 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 contrainte par les décisions d'implémentation effectuées par des équipes externes au projet
- Collés à 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 redirigez-les 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) d'Istio est composé de proxys de service, basés sur le proxy Envoy, qui cohabitent avec les applications.
Les proxys affectent le comportement du réseau en fonction de la configuration envoyée par le plan de contrôle.
- 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.

- 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



Démarrer avec Istio

Concepts
Installation
Fonctionnalités
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



Fonctionnalités

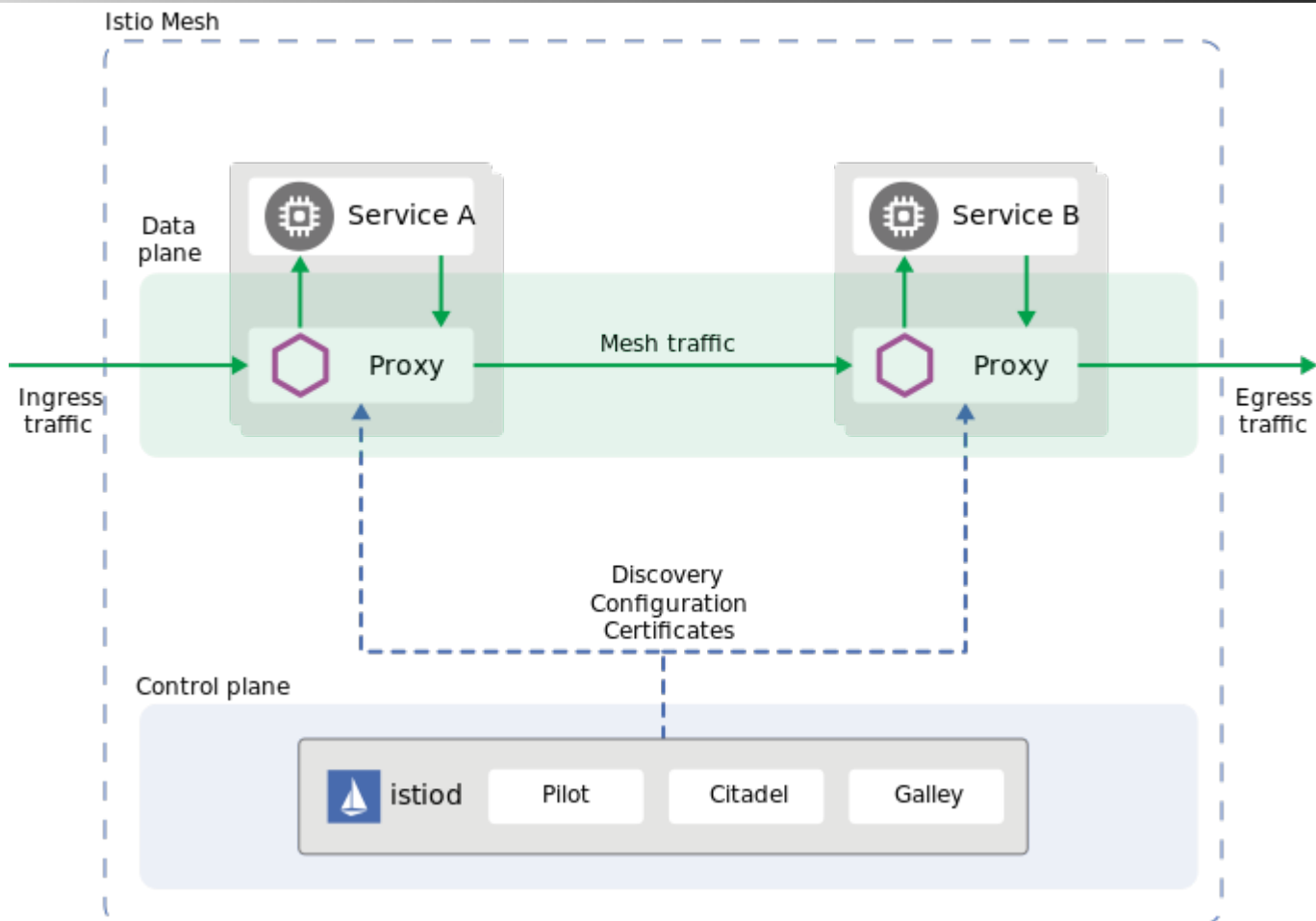
Istio est un ***service mesh*** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

- Un *sidecar proxy* est déployé à côté de chaque micro-service et intercepte toutes les communications.
- Un tableau de contrôle permet de gérer de façon centralisée.

Les fonctionnalités apportées sont nombreuses :

- Équilibrage de charge automatique
- Contrôle du trafic avec des règles de routage
- Résilience avec des politique de ré-essai, du failover, de disjoncteur
- Sécurisation via des contrôles d'accès, des limites de taux et des quotas.
- Sécurisation des communications via TLS et certificats.
- Collecte de métriques et tracing.

Architecture





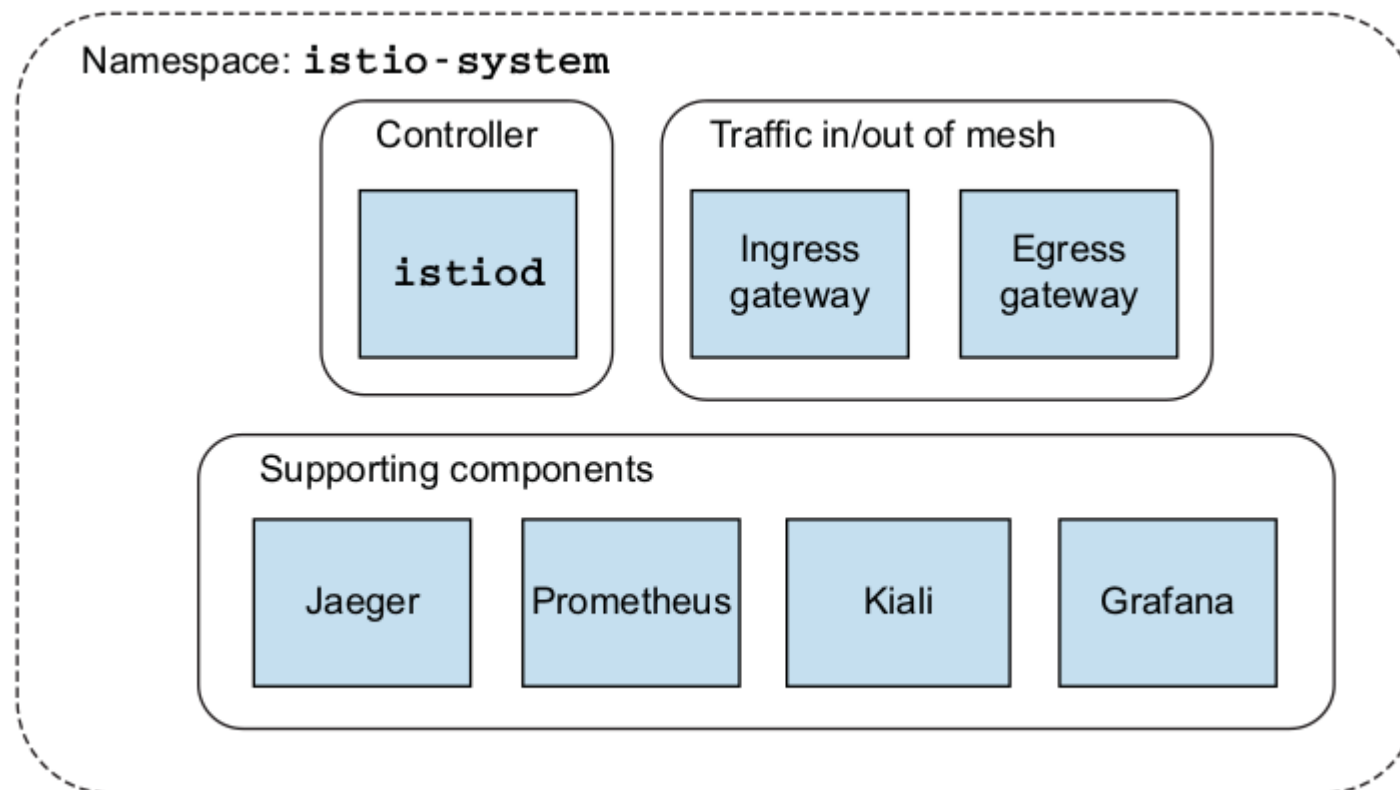
Control Plane

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

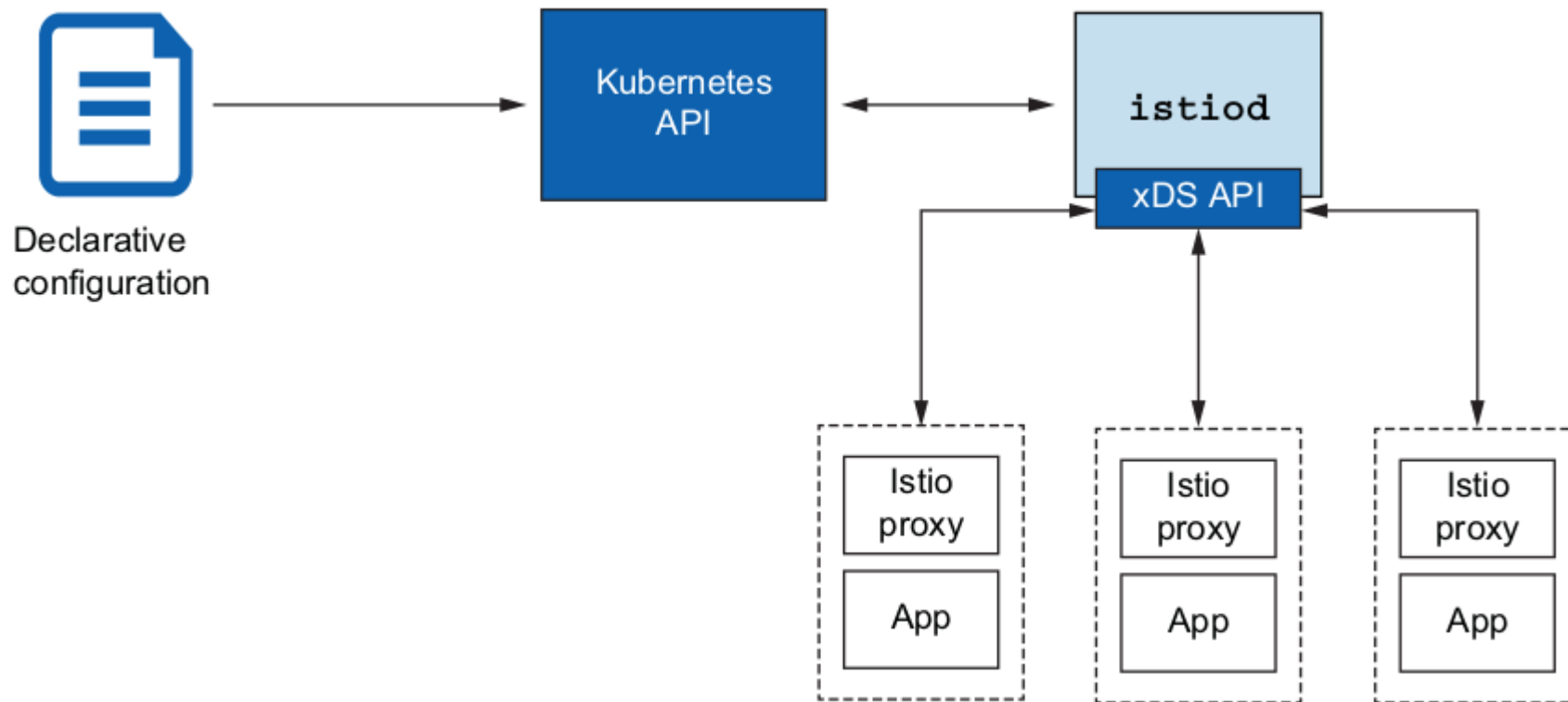
- API permettant de spécifier le comportement de routage/résilience souhaité
- API pour que les proxy Envoy consomment leur configuration
- Un service de découverte
- Identification des workloads via des certificats TLS
- Délivrance et rotation des certificats
- API pour spécifier les autorisations d'accès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



Istiod et composants supportés



Ressources Kubernetes





Ressources Istio

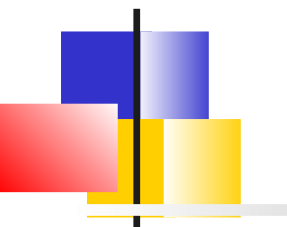
Gateways : Trafic extérieur au service mesh.
(proxy envoy autonome)

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

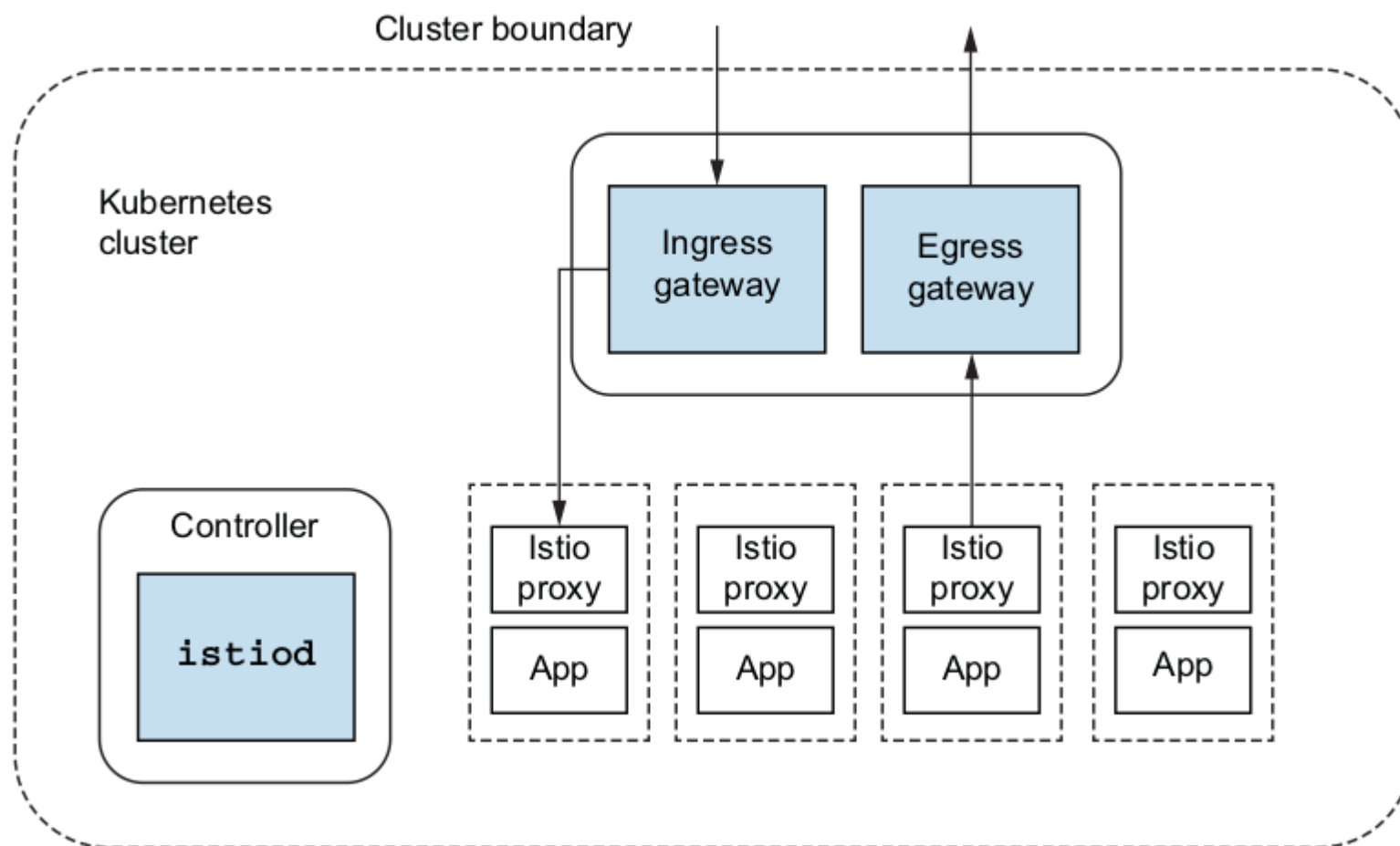
Destination rules : Stratégie de répartition de
charge, Sécurisation, Circuit breaker

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

Sidecars : Configuration du proxy *envoy* sidecar



Gateways





Exemple *DestinationRule*

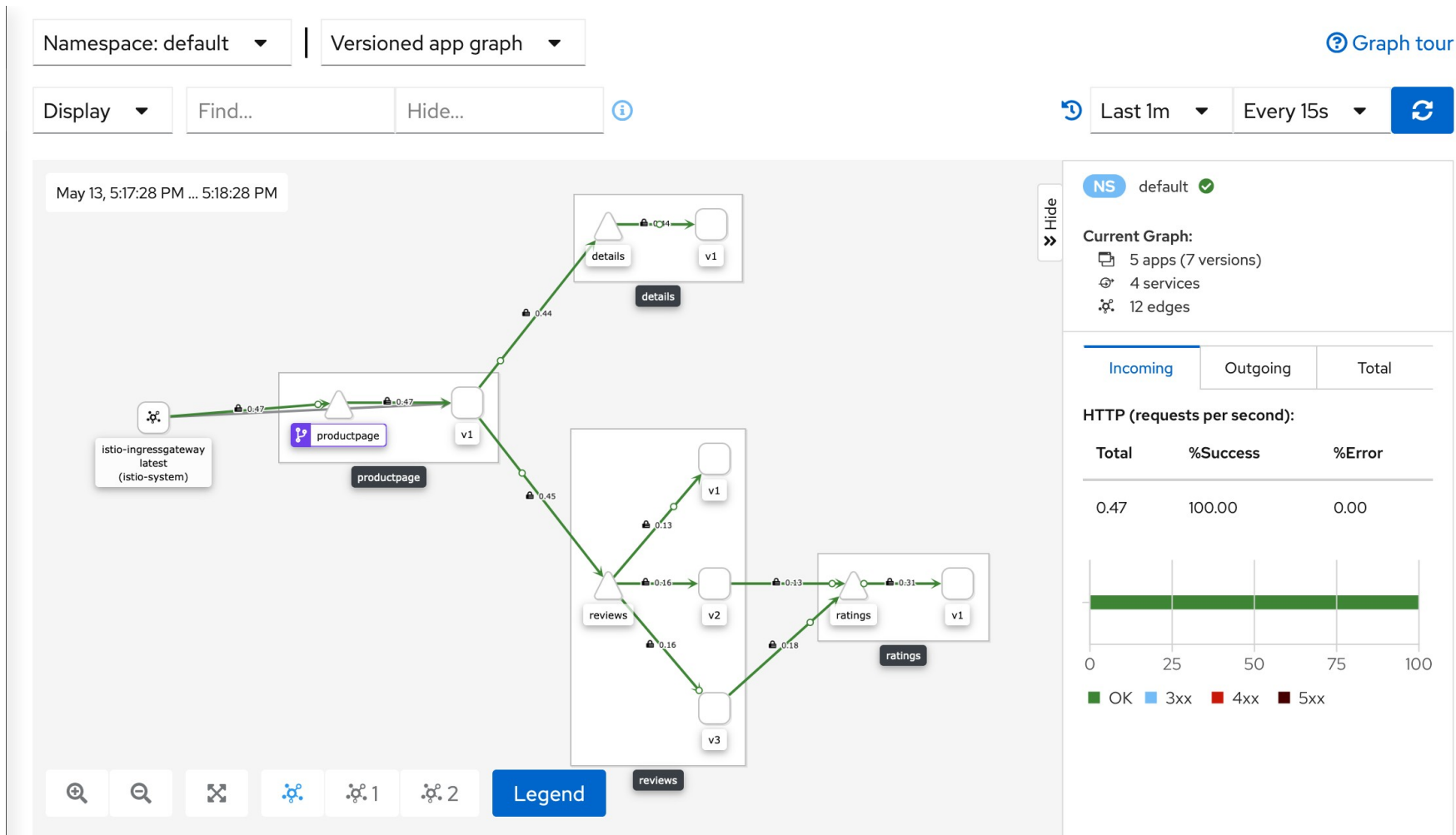
```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews-destination-rule
spec:
  host: reviews.formation.svc.cluster.local          # Service Kubernetes associé
  trafficPolicy:
    loadBalancer:
      simple: RANDOM                                # Algorithme de 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:
    - reviews.formation.io          # Services, IP de destination
  http:
    - match:
        - headers:
            end-user:
              exact: jason          # Règles de routage
                                   # 1 règle sur les entêtes http
      route:
        - destination:
            host: reviews
            subset: v2              # Un sous-ensemble du service (workloads avec label)
                                   # Route par défaut
    - route:
        - destination:
            host: reviews
            subset: v3
```

Tableau de bord Kiali





Démarrer avec Istio

Concepts
Installation
Fonctionnalités
Proxy Envoy



Installation

Télécharger une distribution qui contient :

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

Ajouter le client au *\$PATH*

Exécuter le déploiement d'Istio sur Kubernetes en utilisant un profil (demo par exemple)

Éventuellement ajouter un label à un *namespace* pour profiter de l'injection automatique de Istio et Envoy

Vérifier l'installation

Éventuellement, installez les composants du plan de contrôle (Prometheus, Grafana, Jaeger, etc).



Démarrer avec Istio

Concepts
Installation
Fonctionnalités
Proxy Envoy



Observabilité

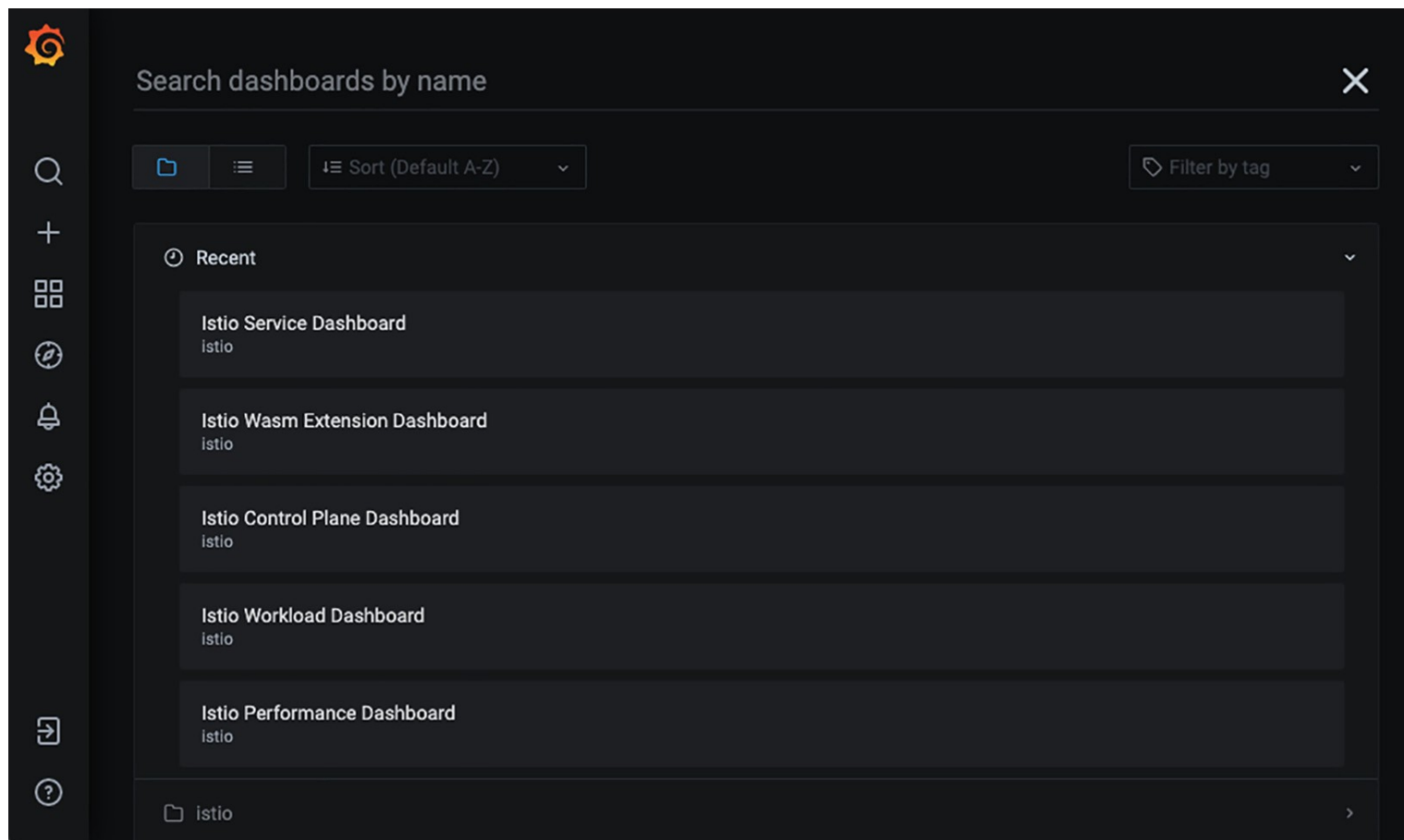
Istio crée 2 types de métriques d'observabilité :

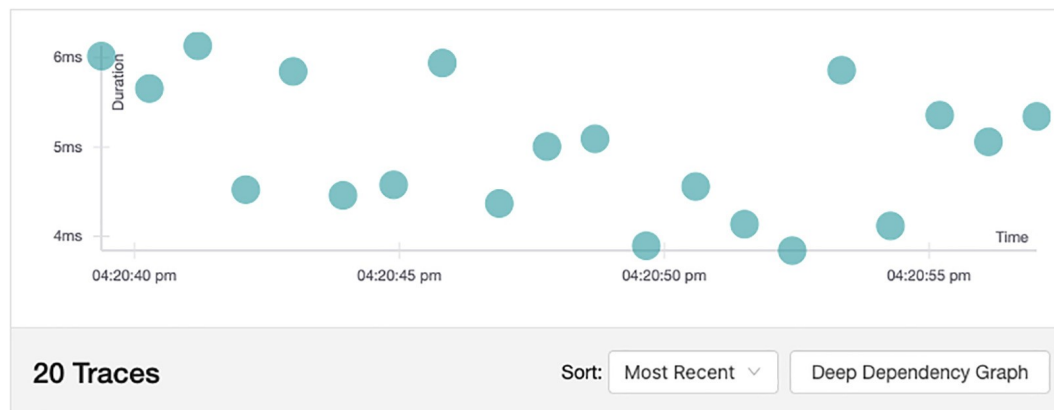
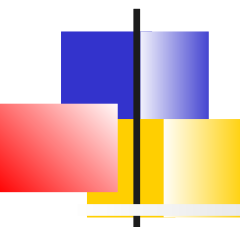
- Les métriques principales tels que le nombre de requêtes par seconde, d'échecs et les percentiles de latence.
- Le traçage distribué comme OpenTracing.io.
Istio peut envoyer des spans à des backends de tracing



Grafana

Istio a des tableaux de bord Grafana prédéfini

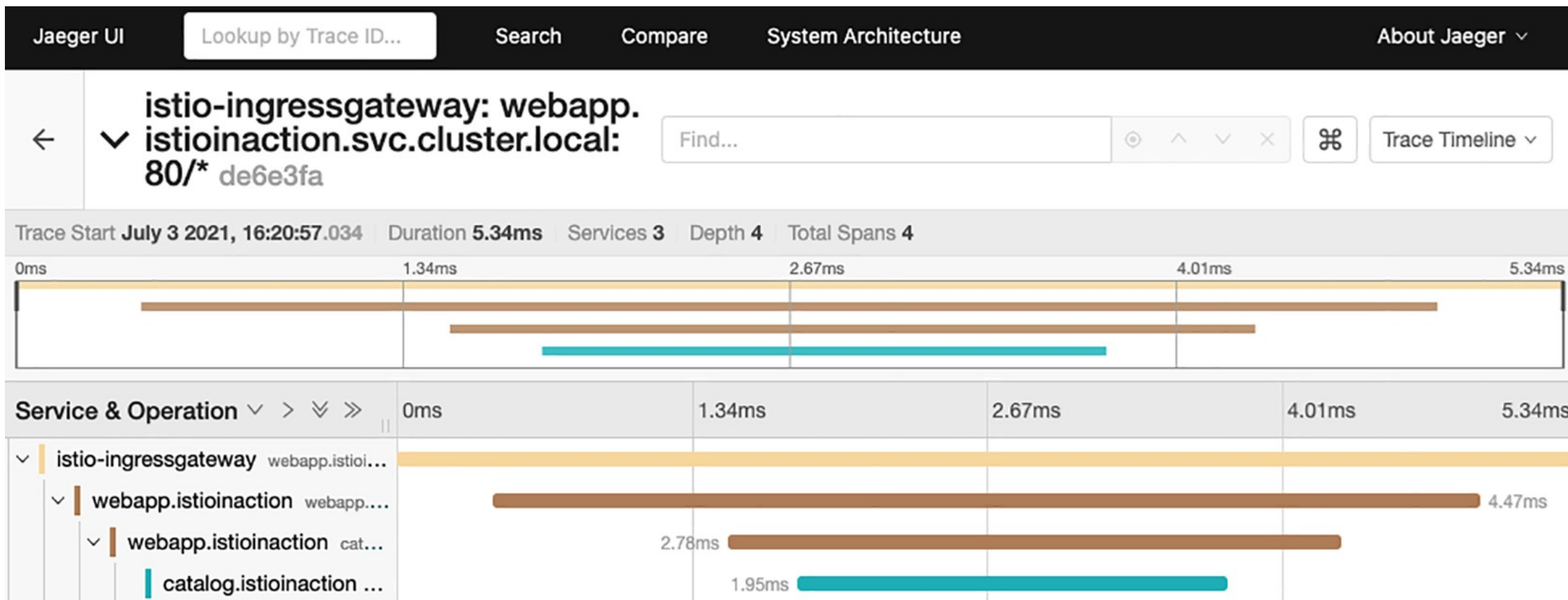




Compare traces by selecting result items

<input type="checkbox"/>	istio-ingressgateway: webapp.istioinaction.svc.cluster.local:80/* de6e3fa	5.34ms
4 Spans	<div><div>catalog.istioinaction (1)</div><div>istio-ingressgateway (1)</div><div>webapp.istioinaction (2)</div></div>	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	<div><div>catalog.istioinaction (1)</div><div>istio-ingressgateway (1)</div><div>webapp.istioinaction (2)</div></div>	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	<div><div>catalog.istioinaction (1)</div><div>istio-ingressgateway (1)</div><div>webapp.istioinaction (2)</div></div>	Today 4:20:55 pm a few seconds ago

Vue détaillée





Résilience

Certains patterns sont définis pour obtenir de la résilience :

- Le ré-essai : Réessayer la même requête un certain nombre de fois pour pallier une défaillance temporaire
- Timeout : Ne pas bloquer une thread si une dépendance est en surcharge
- Circuit-breaking : Ne pas perdre son temps à faire des requêtes vers une dépendance défaillante ... et lui laisser le temps de ce remettre



Démarrer avec Istio

Concepts
Installation
Fonctionnalités
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 protocols comme HTTP 1.1, HTTP 2, gRPC, et d'autres

Il peut ainsi collecter de nombreuses 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
- Ou par service comme avec Istio.



Concepts Envoy

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

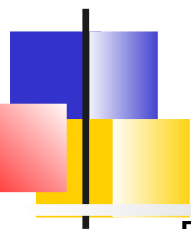
Un listener accepte du trafic et applique la configuration de ce trafic.

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 coeur 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

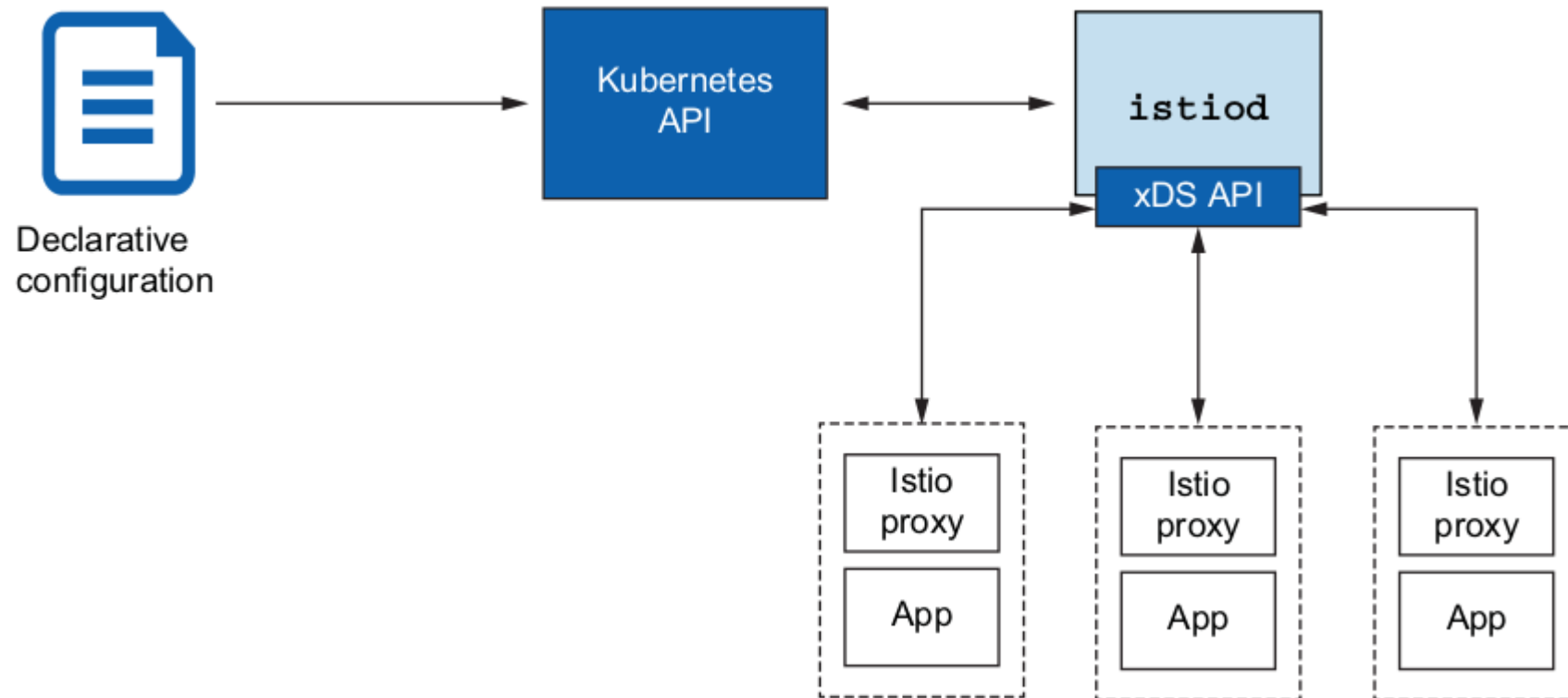
Fichiers JSON ou YAML qui spécifient listeners, routes, et clusters ainsi que d'autres points

L'API (v3) est construite sur gRPC (peut profiter du streaming et d'un mode push)

Envoy peut utiliser un ensemble d'APIs¹ 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*

Envoy et Istio





Istio in Action

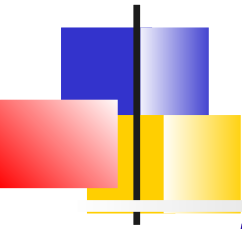
Gateway

Routing

Resilience

Observabilité

Sécurisation



Ingress

Ingress fait référence au trafic qui provient de l'extérieur du réseau et 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 forward sur le bon endpoint du réseau local.
- Sinon, il rejette le trafic.



Virtual IP ou Hosting

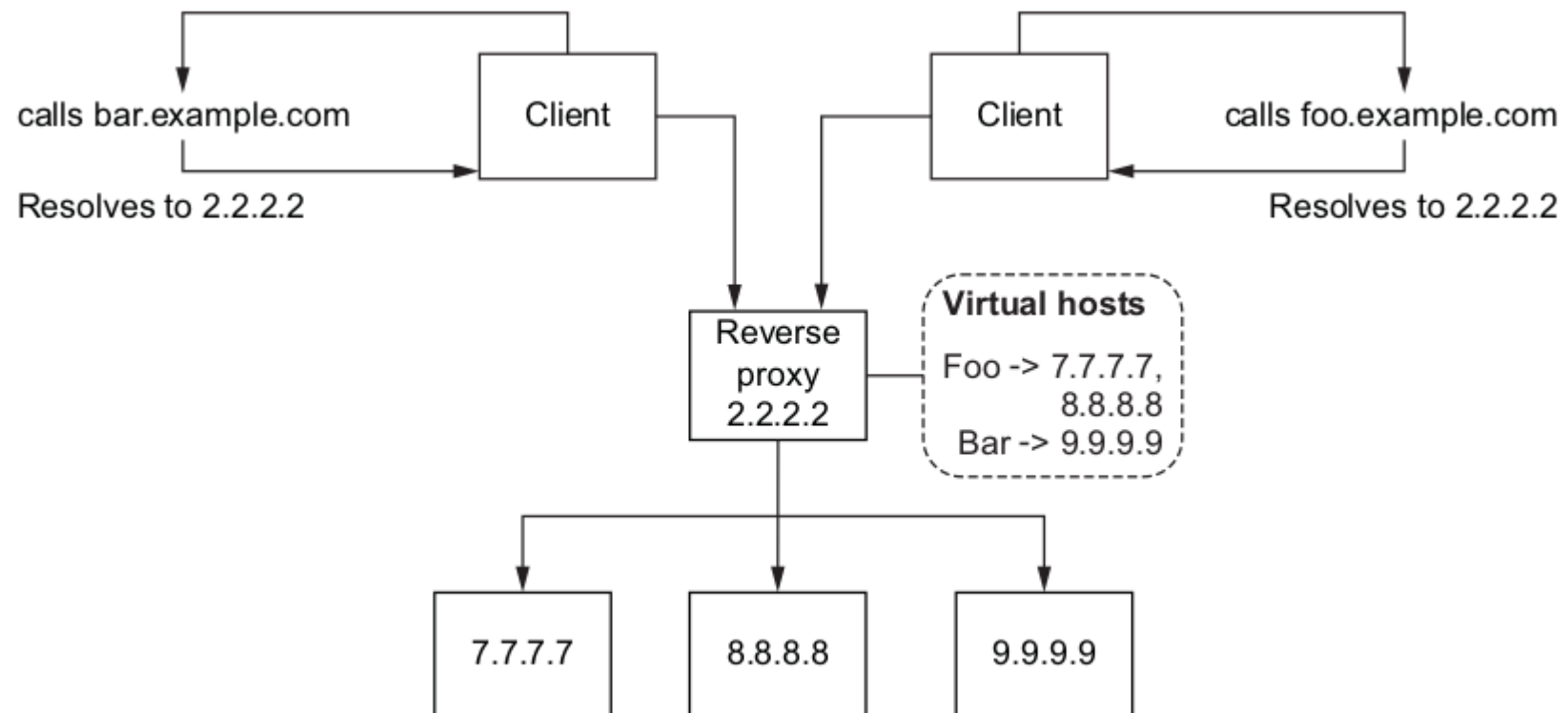
Les techniques de virtual IP ou virtual hosting permettent un routage plus robuste vers le service cible .

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





Istio-*gateway

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

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
```



Différence entre Ingress Istio et Ingress Kubernetes

Kubernetes Ingress v1 est une spécification très simple conçue pour HTTP (80 et 443), avec Istio on peut descendre au niveau TCP (utile pour Kafka par exemple)

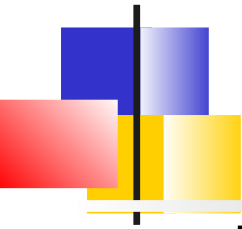
Kubernetes Ingress v1 est sous-spécifiée. Il ne définit pas comment doivent être spécifiées des règles de routage de trafic complexes, la répartition du trafic etc..

=> chaque fournisseur propose des implémentations différentes (HAProxy, Nginx, etc.)

=> Les méta-données de configuration sont également non portables

Istio sépare clairement les propriétés

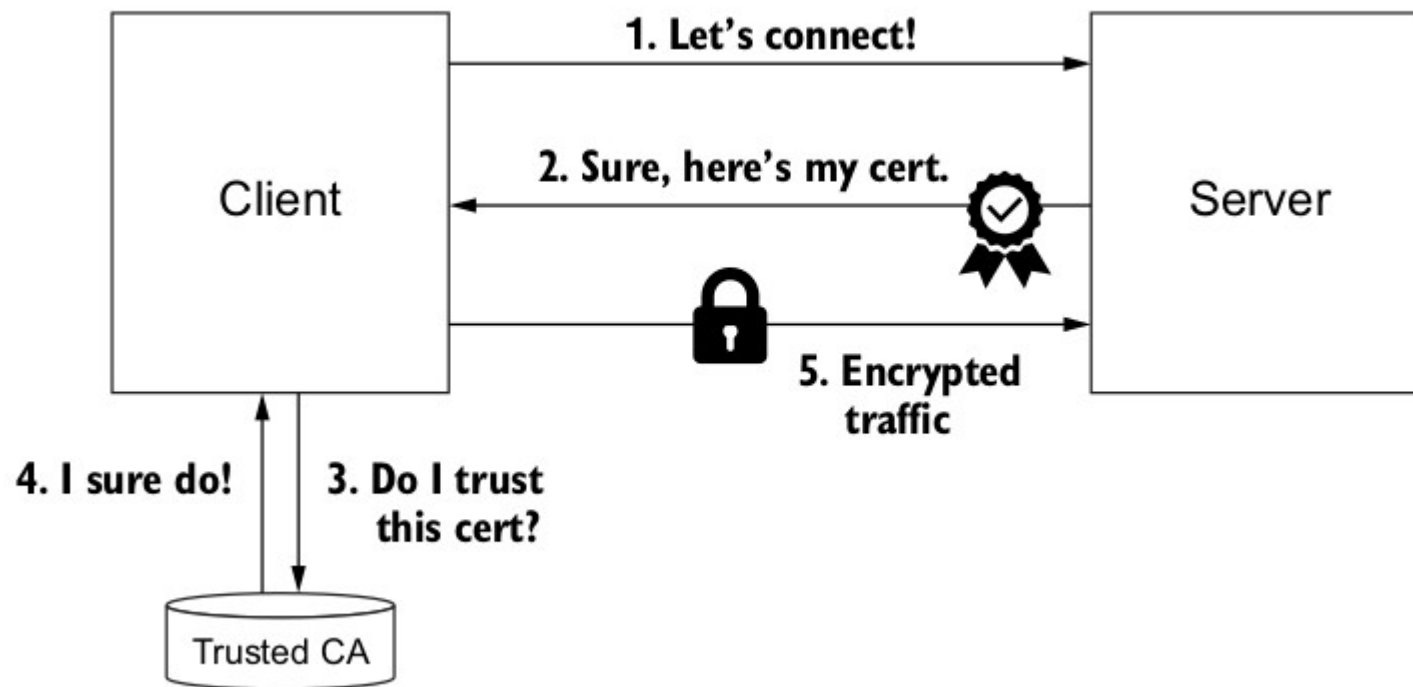
- de la couche 4 (transport) et de la couche 5 (session) : *Gateway*
- des problèmes de routage de la couche 7 (application). :
VirtualService



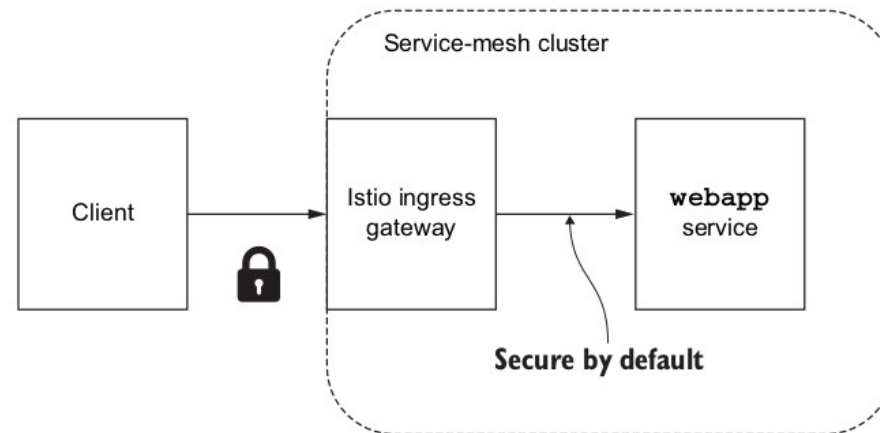
Sécurisation des gateways

Les gateways d'Istio permettent de transférer le trafic TLS/SSL entrant, de rediriger tout trafic non TLS vers les ports TLS appropriés ou de mettre en œuvre le TLS mutuel (*mTLS*).

Rappel TLS



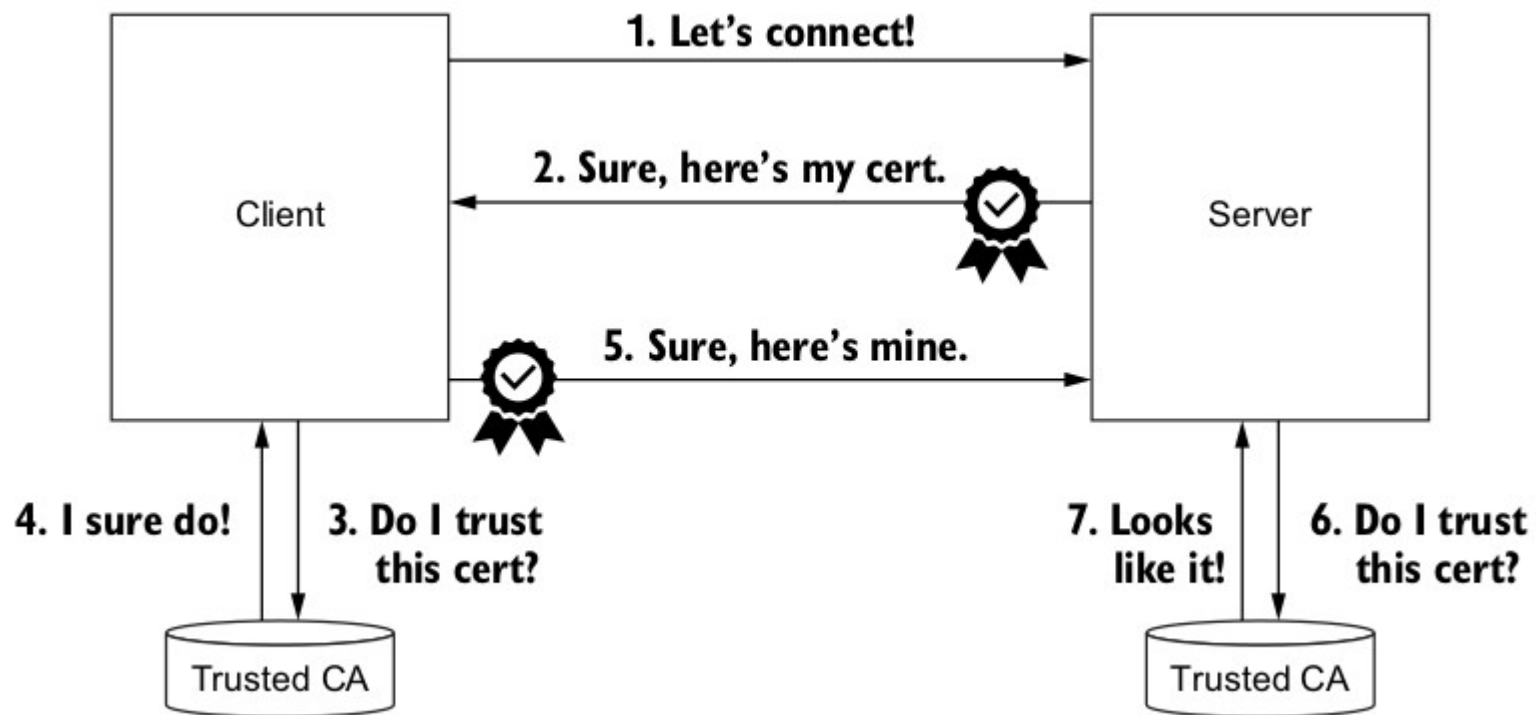
Traffic TLS



Pour autoriser HTTPS pour le trafic ingress, il faut spécifier la clé privé et le certificat que la gateway doit utiliser

- les certificats peuvent être installés dans Kubernetes via des *secrets*

TLS mutuel (mTLS)





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
        name: https-webapp
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: webapp-credential
      hosts:
        - "webapp.formation.io"
    - port:
        number: 443
        name: https-catalog
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: catalog-credential
      hosts:
        - "catalog.istioinaction.io"
```



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 statuts 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

On peut également mettre en place de l'injection de gateway facilitant la mise en place par les équipes

Activer les logs de la gateway pour la résolution de problème



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
        enabled: false
      - name: my-user-gateway
        namespace: formation
        enabled: true
        label:
          istio: my-user-gateway
-
# Installation d'une nouvelle gateway pour le namespace formation
istiocctl install -y -n formation -f my-user-gateway.yaml
```



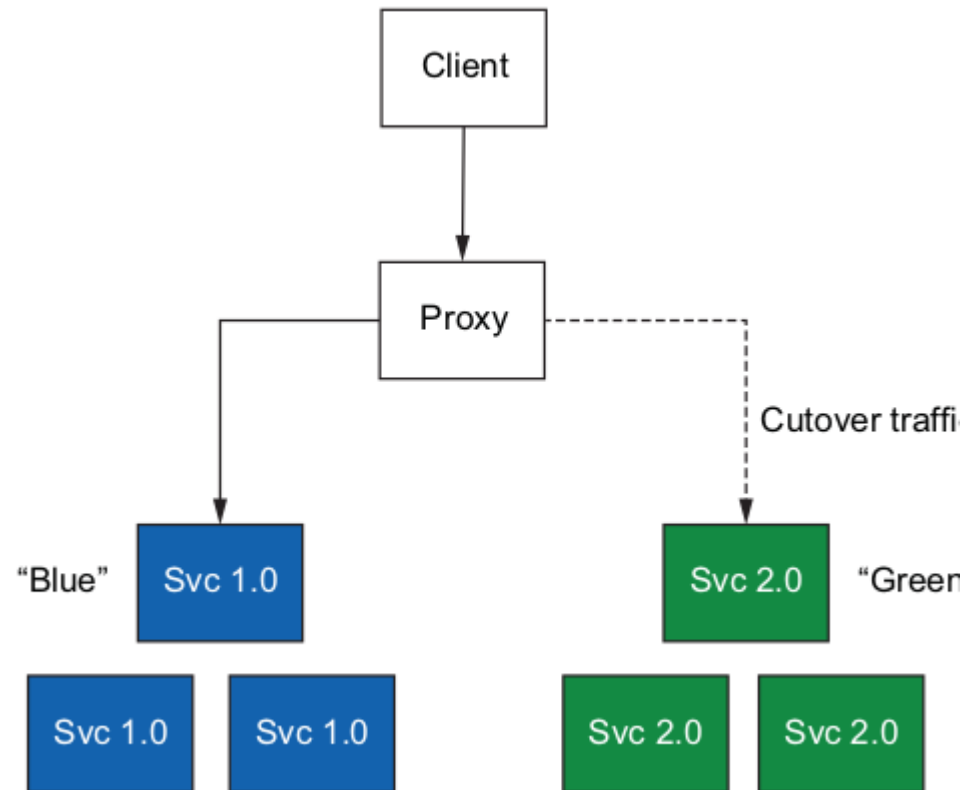
Istio in Action

Gateway
Routing
Résilience
Observabilité
Sécurisation

Déploiements Blue/Green

Les déploiements Blue/green consiste 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 distingues le déploiement de la release

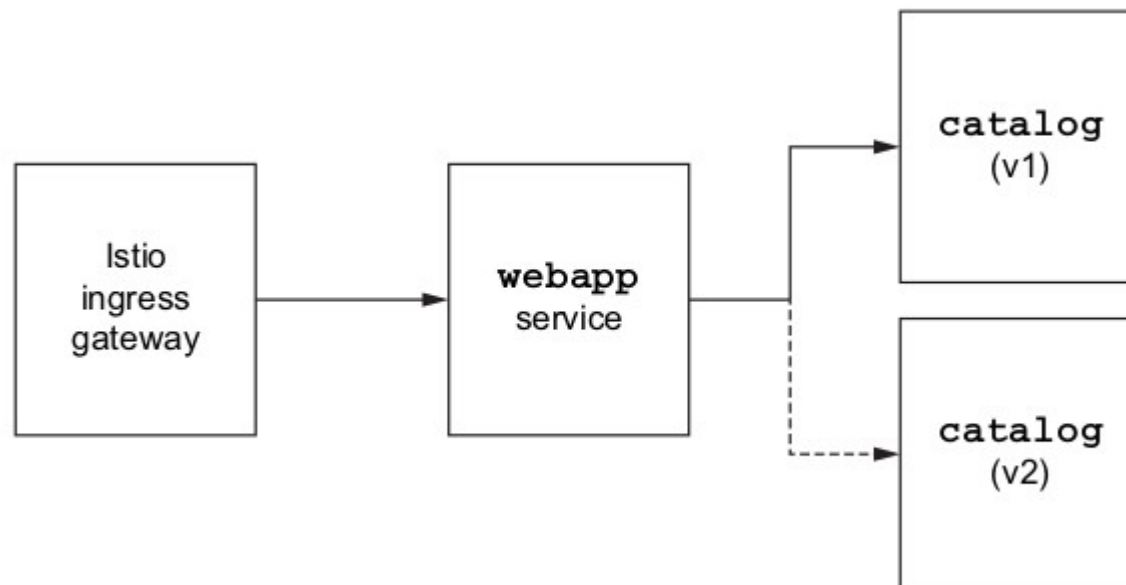
- Un déploiement en production consiste à installer le nouveau code en production mais aucun trafic réel ne lui est envoyé Il est alors possible d'exécuter des tests afin de vérifier que tout marche comme prévu
- La release consiste à router du trafic vers le nouveau déploiement. Avec une canary release, 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

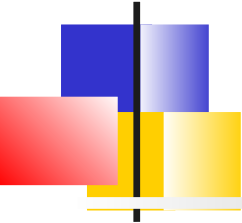
Subset et label

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

Ces règles de trafic peuvent s'effectuer dans tout le service mesh





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



Entêtes



Pondération

Tout le trafic est dirigé vers un ensemble de versions d'un même service. La répartition est 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.

***Flagger*¹** s'intègre particulièrement bien avec Istio

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

1. <https://flagger.app>



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
```



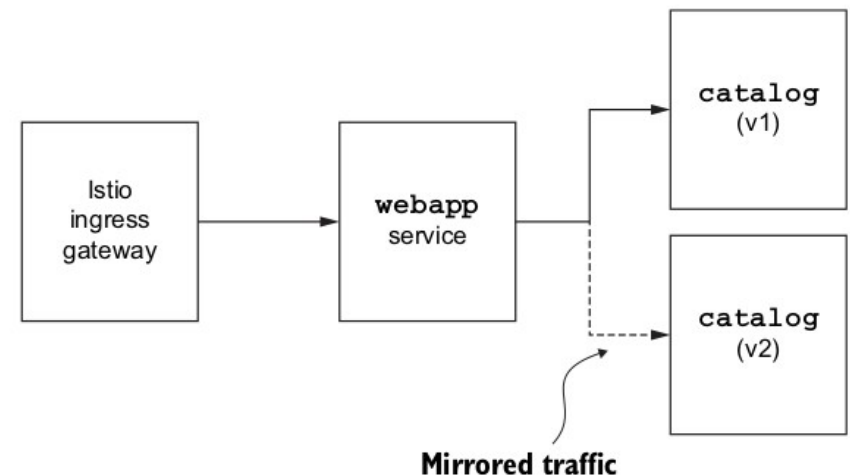

Exemple configuration (2)

```
# 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 atteints le canari est roll-back
```

Traffic 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



Istio in Action

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 tentatives
- Court-circuit (Circuit-breaker)

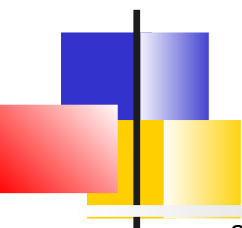


Répartition de charge côté client

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

L'algorithme d'équilibrage est défini via une ressource *DestinationRule* :

- Round robin (défaut)
- Au hasard
- Demande la moins pondérée :
Prend en compte les latences des points d'accès en surveillant la taille des files d'attente, les demandes actives et sélectionne le point d'accès avec le moins de demandes actives en cours
- ...

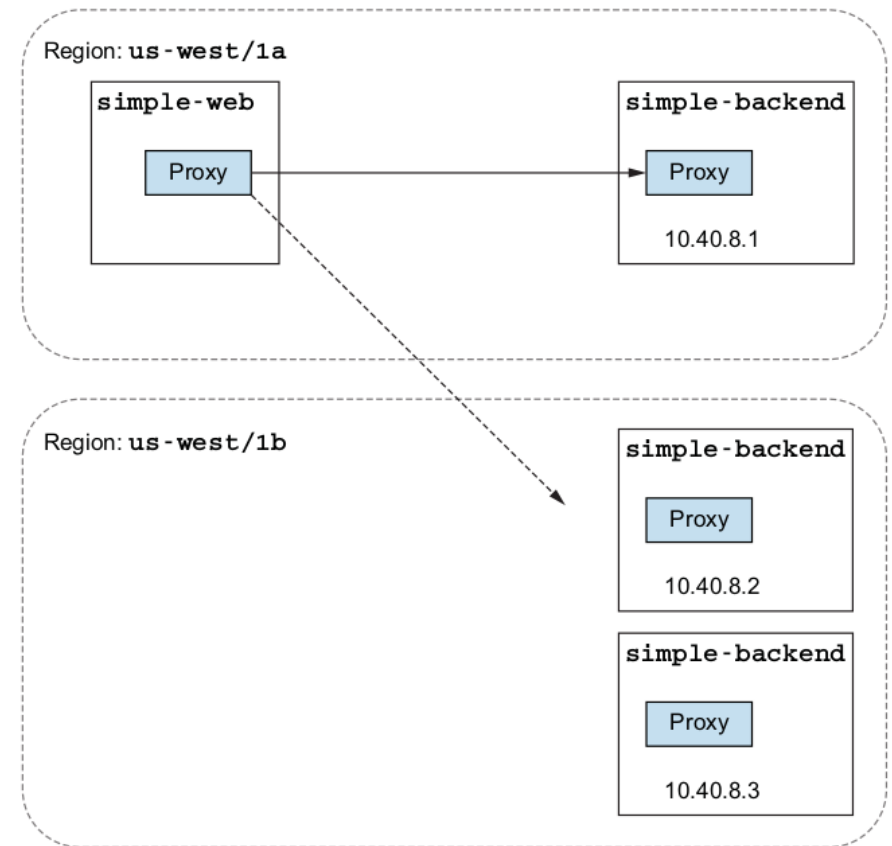


Configuration – Moins chargé

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

Répartition en fonction de la localité

Istio peut identifier la région et la zone de disponibilité dans lesquelles un service particulier est déployé¹ et donner la priorité aux services proches



1. Via les labels Kubernetes



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, il est logique d'avoir des délais d'attente plus longs en entrée de l'architecture et des délais d'attente plus restrictifs pour les couches plus basses dans le graphe d'appels

Les timeouts sont configurés via la ressource *VirtualService*



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 timeout (*perTryTimeout*) *perTryTimeout* **nbRetry* doit être plus petit que le timeout global

Par défaut, Istio réessaye 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 entre 2 essais est 25ms.

Les retry sont configurés via une ressource *VirtualService*

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
```



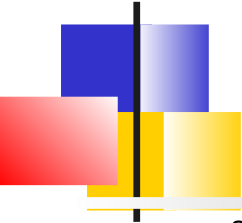
Circuit-breaker

La fonctionnalité de circuit-break sert à se prémunir contre les défaillances partielles ou 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 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
connectionPool dans une DestinationRule
- Observer la santé des points d'accès dans un équilibrage de charge et expulser pendant un certain temps ceux qui se comportent mal



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 overloaded

Lorsqu'une requête échoue à cause d'une ouverture 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



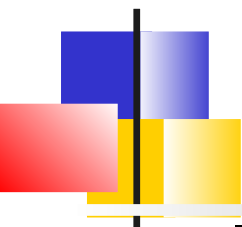
Exemple configuratio instance en mauvaise santé

```
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
```



Istio in Action

Gateway
Routing
Résilience
Observabilité
Sécurisation



Introduction

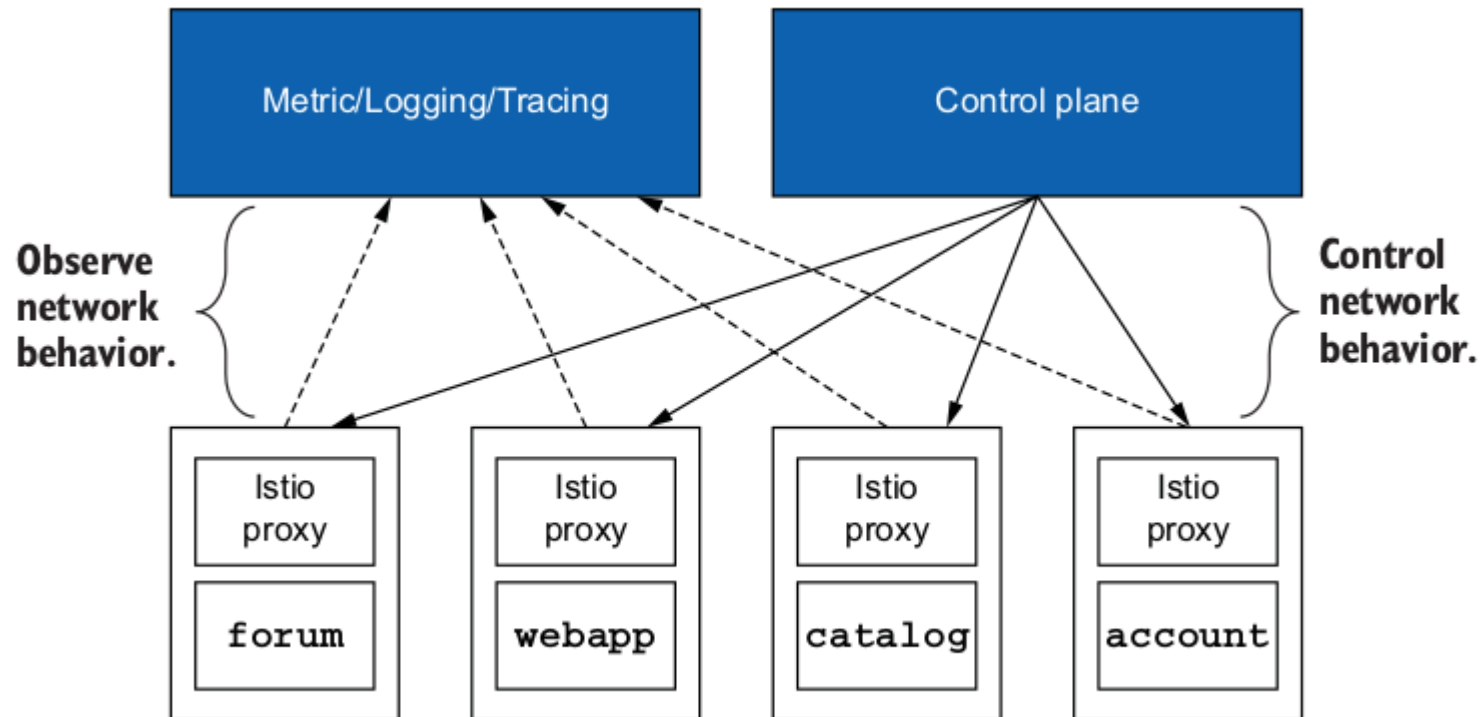
Istio, via ses proxy, est dans une position idéal pour aider à créer un système observable

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 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

Il est possible de configurer les proxies afin qu'ils envoient plus de métriques¹.

Cette configuration peut être fait

- au niveau du maillage ... 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

Envoy a une notion d'origine interne et externe (par rapport au mesh) lors de l'identification du trafic.

Avec :

- **<cluster_name>.internal.*** : Permet de visualiser les requêtes d'un cluster du 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

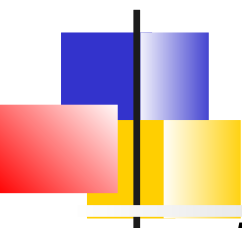


Connaissance du mesh

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

```
kubectl exec -it deploy/webapp -c istio-proxy \
```

```
-- curl localhost:15000/clusters
```



Métriques du Control Plane

istiod conserve beaucoup d'informations sur ses performances (nombre de synchronisation de configuration avec les proxys, durée et d'autres informations telles que les mauvaises configurations, 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 de cibles qui exposent 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.



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 nouvelles métriques
- Grouper des attributs existants (par exemple avoir une information pour les requêtes qui contiennent un path ou paramètre particulier)

1. Pour les visualiser : `kubectl get envoyfilter -n istio-system`



Grafana

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

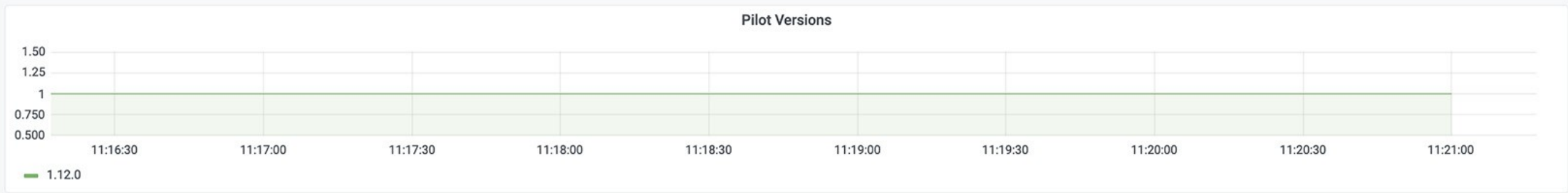
Dans l'environnement Kubernetes, le chargement des tableaux de bord Grafana s'effectue via une ConfigMap labellisé avec :

```
grafana_dashboard=1
```

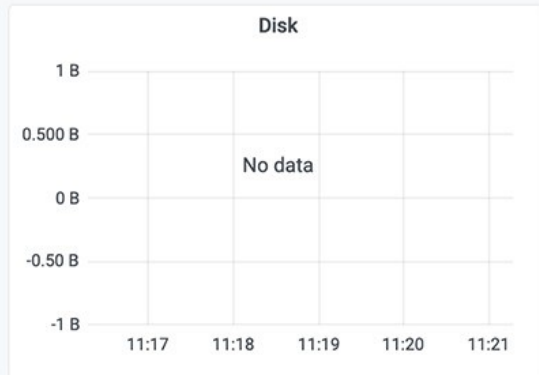
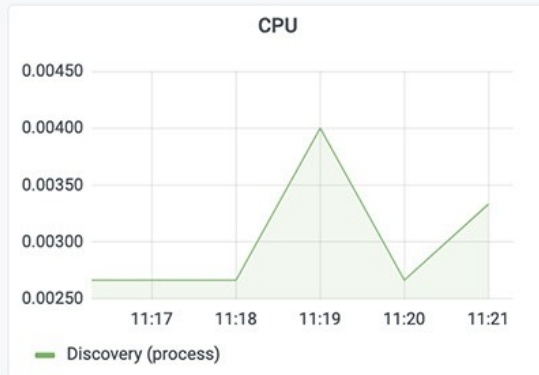
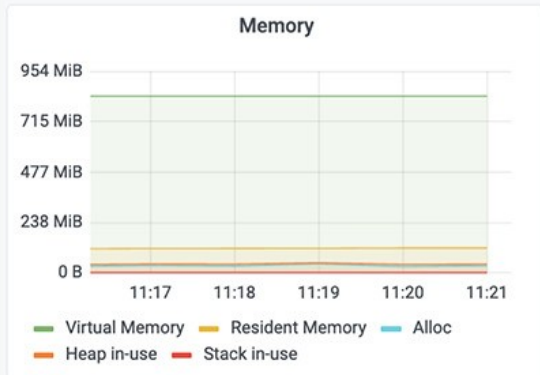
1. Voir : <https://grafana.com/orgs/istio/dashboards>

datasource default

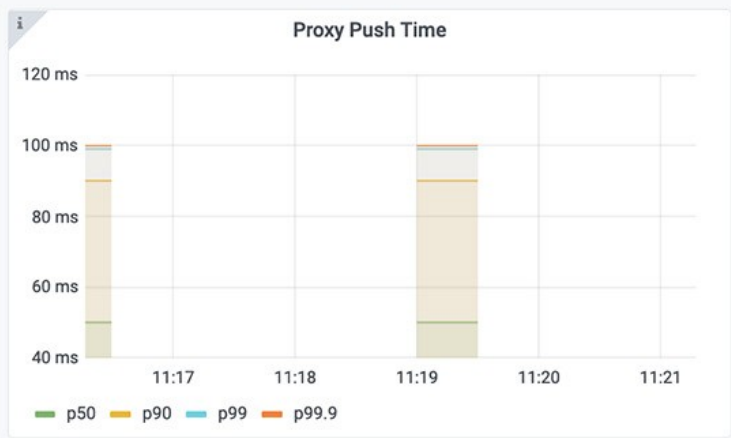
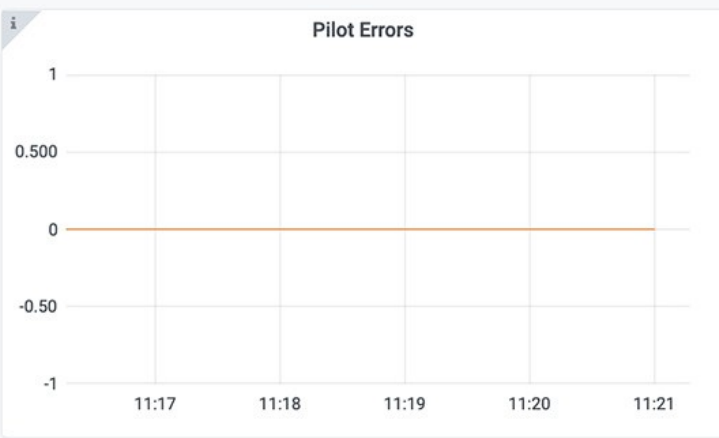
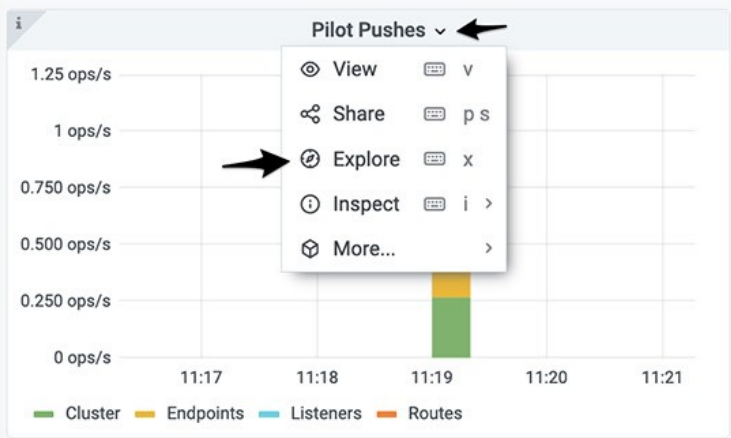
Deployed Versions



Resource Usage



Pilot Push Information



Tdb Service





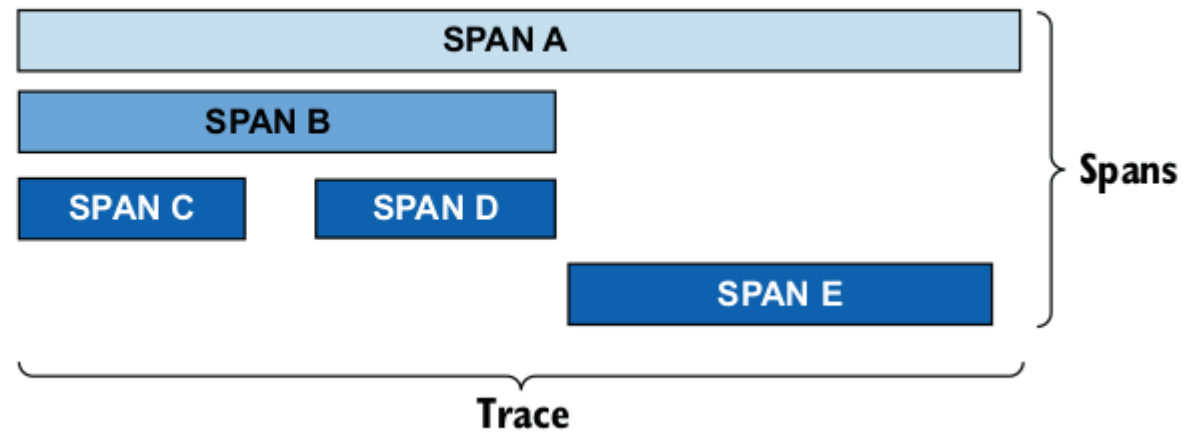
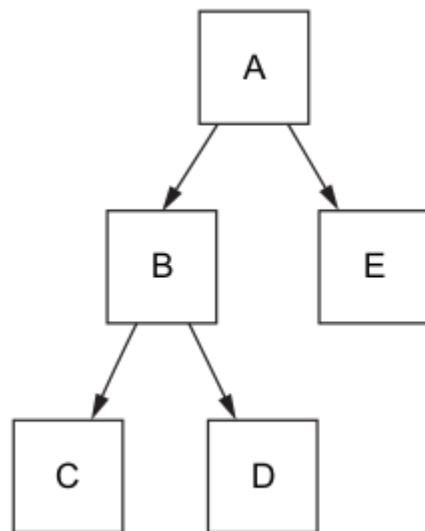
Tracing distribué

Le tracing distribué consiste à annoter les requêtes avec des ID de corrélation qui représentent les appels de service à service et un ID de suivi qui représentent une requête.

Le proxy Istio peut ajouter ces métadonnées et les envoyer vers un 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 corréler 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

Comme d'habitude, la configuration peut se faire à différents niveaux : global au maillage, 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: {}
```



Echantillonnage 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 ConfigMap

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'échantillonnage 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

JAEGER UI

Search

Compare

System Architecture

Monitor

Q Lookup by Trace ID...

About Jaeger ▾

Search

JSON File

Service (4)

istio-ingressgateway.istio-system ▾

Operation (2)

all ▾

Tags (?)

http.status_code=200 error=true

Lookback

Last Hour ▾

Max Duration

Min Duration

e.g. 1.2s, 100ms, 500us

e.g. 1.2s, 100ms, 500us

Limit Results

20

Find Traces

Duration

1s

500ms

03:46:40 pm

03:48:20 pm

03:50:00 pm

03:51:40 pm

Time

2 Traces

Sort: Most Recent ▾

Deep Dependency Graph

Compare traces by selecting result items

☐

istio-ingressgateway.istio-system: webapp.formation.svc.cluster.local:80/* d3c6c0d

50.57ms

4 Spans

catalog.formation (1)

istio-ingressgateway.istio-system (1)

webapp.formation (2)

Today

3:51:58 pm

a few seconds ago

☐

istio-ingressgateway.istio-system: httpbin.org:80/* 24abd44

1.41s

1 Span

istio-ingressgateway.istio-system (1)

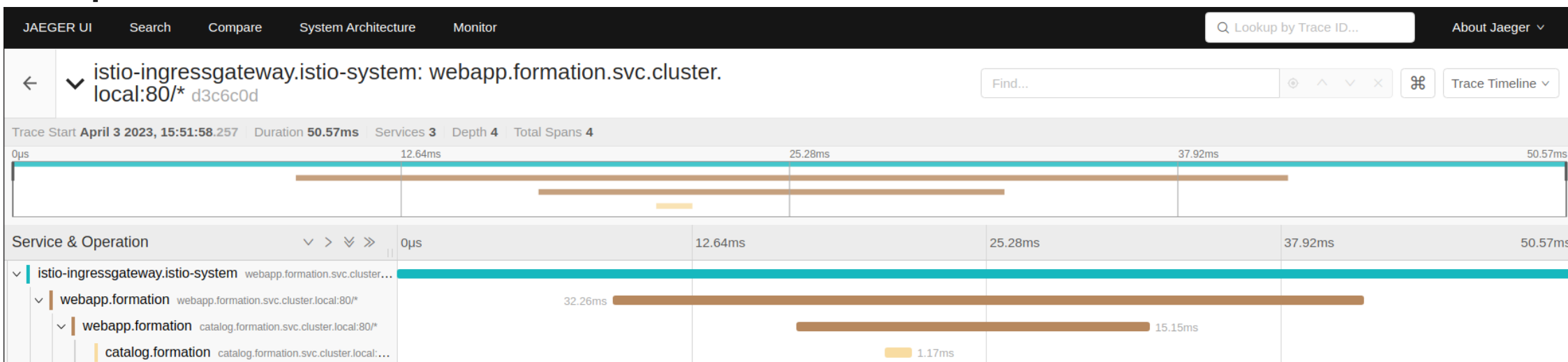
Today

3:46:02 pm

6 minutes ago



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>



Kiali

☰

kiali

Overview

Graph

Applications


Workloads




Services

Istio Config

Namespace ▾ Filter by Namespace

Name ▾ ⚡

Last 1m ▾ Every 15s ▾ 

Health for Apps ▾   

default

No labels

Istio Config N/A


0 Applications N/A

No traffic

prometheus

No labels

Istio Config N/A


2 Applications  2

No traffic

istio-system

1 Label


Istio Config N/A


4 Applications  4

No traffic

istioinaction

1 Label

Istio Config 

2 Applications  2

Traffic, 1m

104



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

Display

Find...

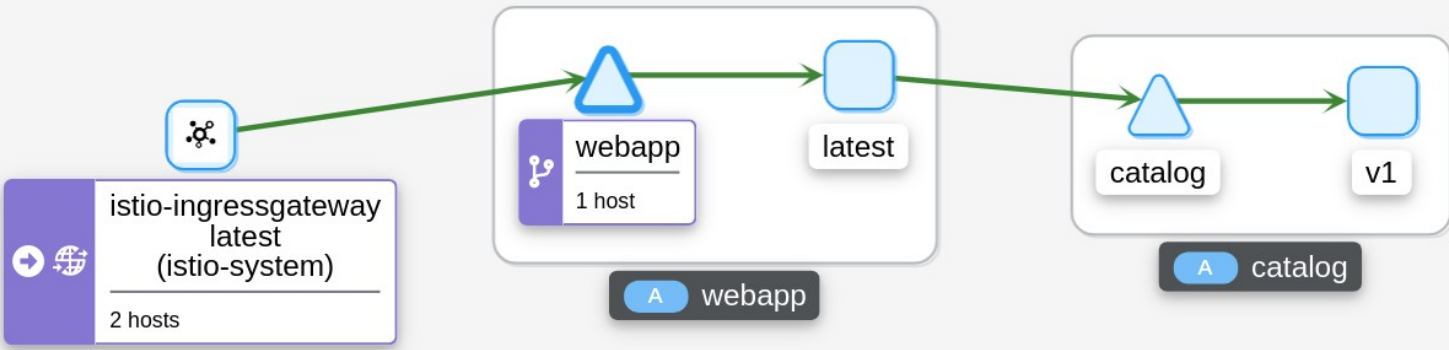
Hide...

Last 1m

Every 15s

Graph tour

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



```
graph LR; IG[istio-ingressgateway latest (istio-system) 2 hosts] --> W[webapp 1 host latest]; W --> C[catalog v1];
```

A webapp

A catalog

Hide

S webapp

health

Has Request Routing

1 host

A webapp

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

0 25 50 75 100

OK 3xx 4xx 5xx NR

HTTP - Request Traffic min / max:

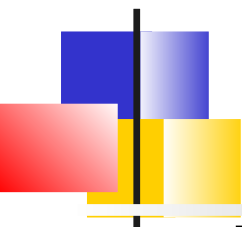
Not enough traffic to generate chart.

No gRPC traffic logged.

1

2

Legend



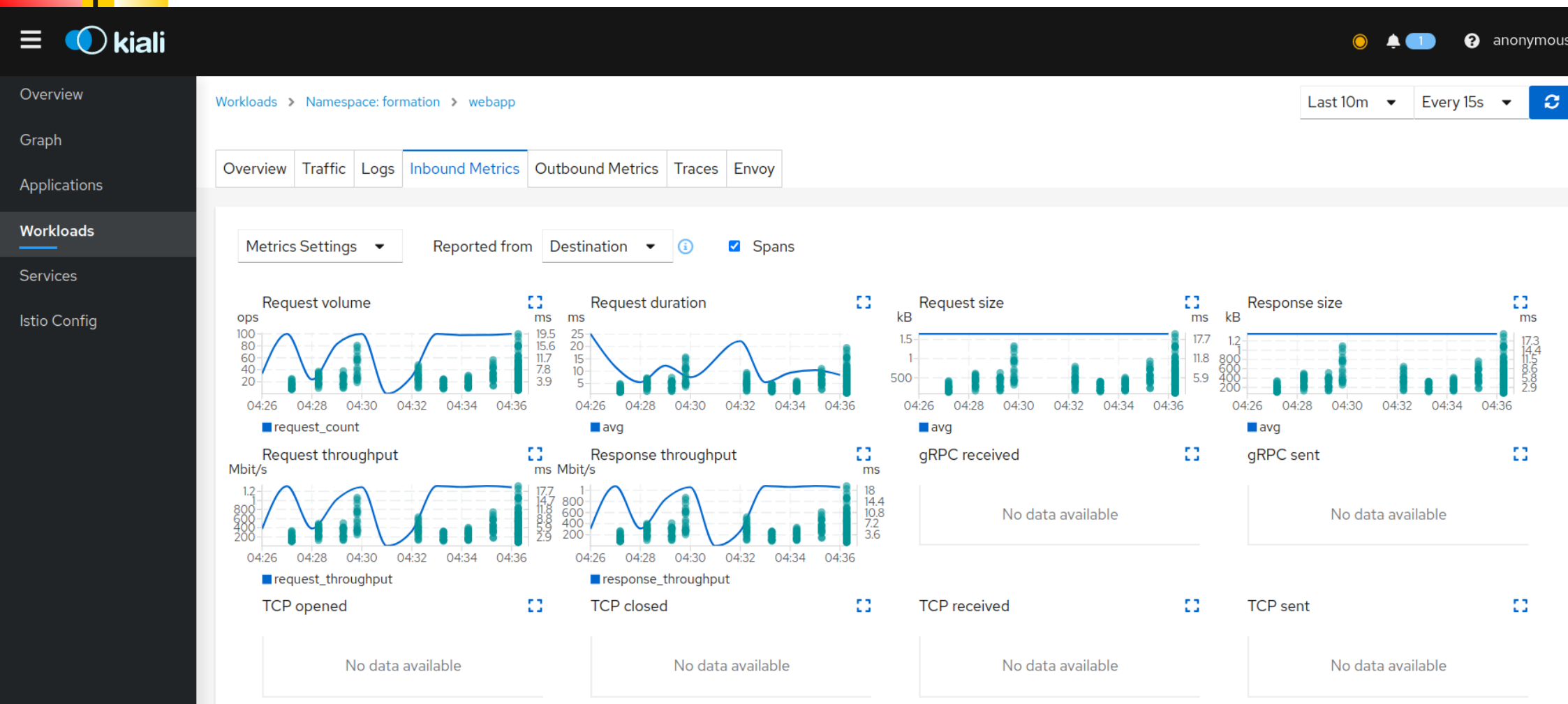
Workload

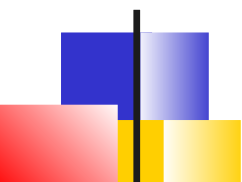
L'onglet Worload propose plusieurs sous-menus :

- **Overview** : Pods du service, sa configuration Istio et un graphe des flux amont et aval
- **Traffic** : 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 appliqué à la workload comme les clusters, listeners et routes



Workload





Config Istio

≡

kiali

Overview

Graph

Applications

Workloads

Services

Istio Config

● 🔔 1 ? anonymous

Namespace: formation ▼

↺ ↻

Istio Config

Actions ▼

Istio Type ▼ Filter by Istio Type ▼

Name ↕	Namespace ↕	Type ↕	Configuration ↕
<div>G</div> coolstore-gateway	<div>NS</div> formation	Gateway	✓
<div>SE</div> external-httpbin-org	<div>NS</div> formation	ServiceEntry	✓
<div>SE</div> jsonplaceholder	<div>NS</div> formation	ServiceEntry	✓
<div>VS</div> thin-httpbin-virtualservice	<div>NS</div> formation	VirtualService	✓
<div>VS</div> webapp-virtualservice	<div>NS</div> formation	VirtualService	✓



Istio in Action

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
 - Authentification service vers service
 - Authentification 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.

1. RFC 3986

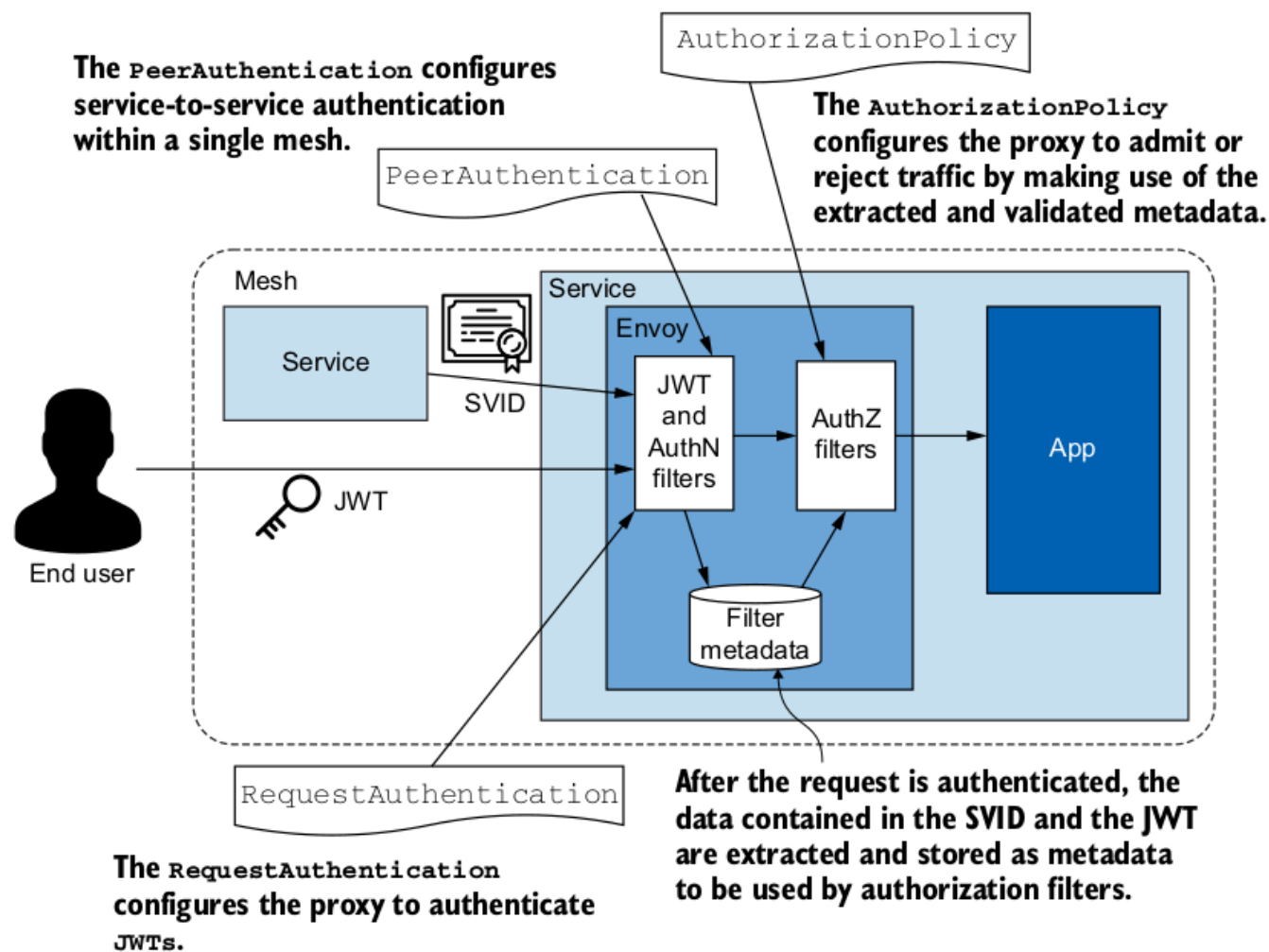
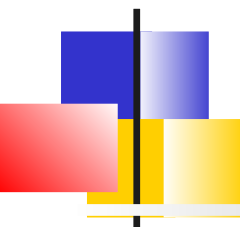
2. *SPIFFE Verifiable Identity Document*



Ressources

L'exploitant Istio utilise différentes ressources pour la sécurité :

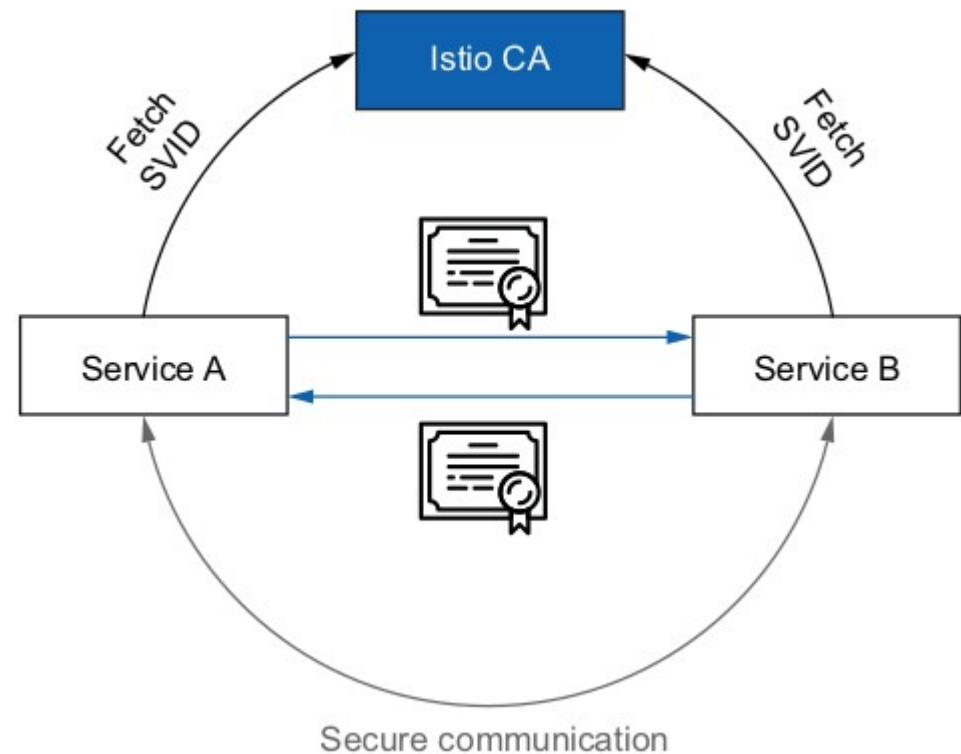
- **PeerAuthentication** : configure le proxy pour authentifier le trafic de service à service. En cas de succès, le proxy met à disposition les informations d'identification pour autoriser la requête.
- **RequestAuthentication** : configure le proxy pour authentifier l'utilisateur final. En cas de succès,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



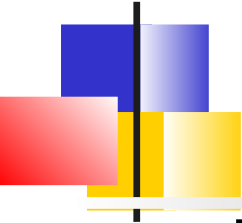


Sécurisation

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 configuré 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 règle 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 correspond, 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 listés 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 Istion), il faut supprimer le champ *from*

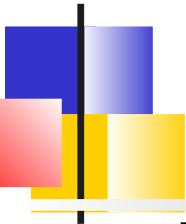
```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "webapp-allow-unauthenticated-view-default-ns"
  namespace: formation
spec:
  selector:
    matchLabels:
      app: webapp
  rules:
    - to:
        - operation:
            methods: ["GET"]
```

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 trainer 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 conséquences 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



Example

```
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": "zl9VRDbmVvyXNdYoGJ5uhuTSRA2653KHEi3XqITfJISvedYHVNGoZZxUCoiSEumxqrPY_Du7IMKzmT4bAuPnEalbY8rafuJNXnxVmjqTrQovPIerkGW5h59iUXIz6vCzn07F61RvJsUEyw5X291-3Z3r-9RcQD9sYy7-8fTNmcXcdG_nNgYCnduZUJ3vFVhmQCwHFG1idwni8PJo9NH6aTZ3mN730S6Y1g_lJf0bjU7lwYWT8j2SjrwT6EES55oGimkZHkktKjDYjRx1rN4dJ5PR5zhLQ4kORWg1PtllWy1s5TSpOUv840PjEohEoOWH0-g238zIOYA83gozgbJfmQ"} ] }
```



Forcer JWT

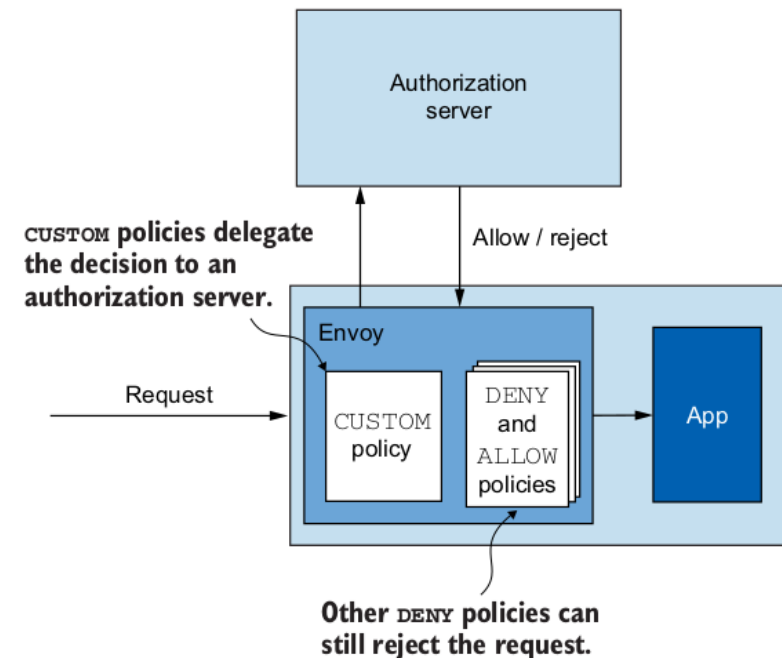
Pour interdire les requêtes sans jeton, il faut une ressource
AuthorizationPolicy

```
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.





Autres

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 reste cependant tellement de configuration 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 donc 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

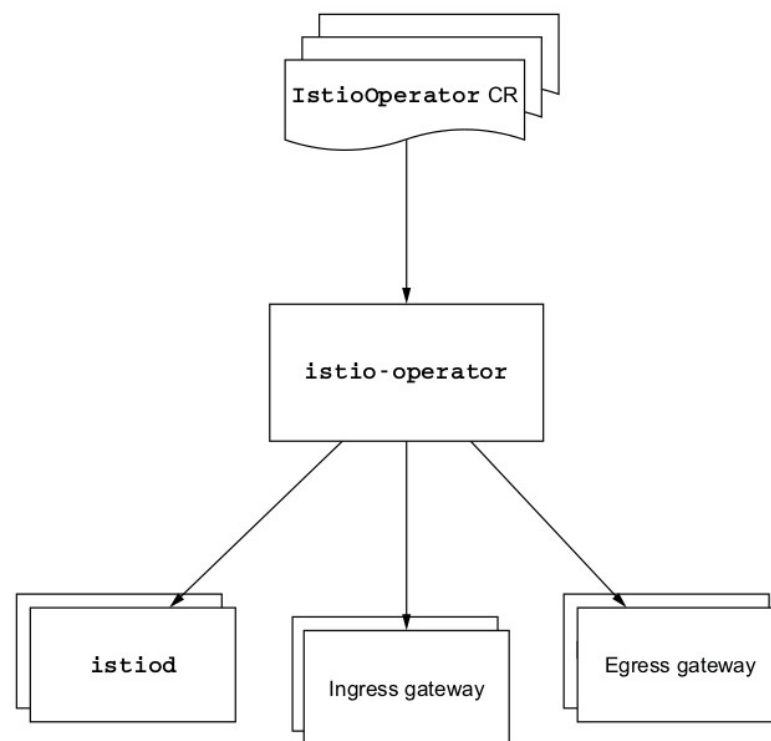
istio-operator gère les installations Istio dans un cluster et permet la personnalisation de l'installation via l'API *IstioOperator*

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





Autres

Installation personnalisée
Résolution de problèmes
Monitoring de istod

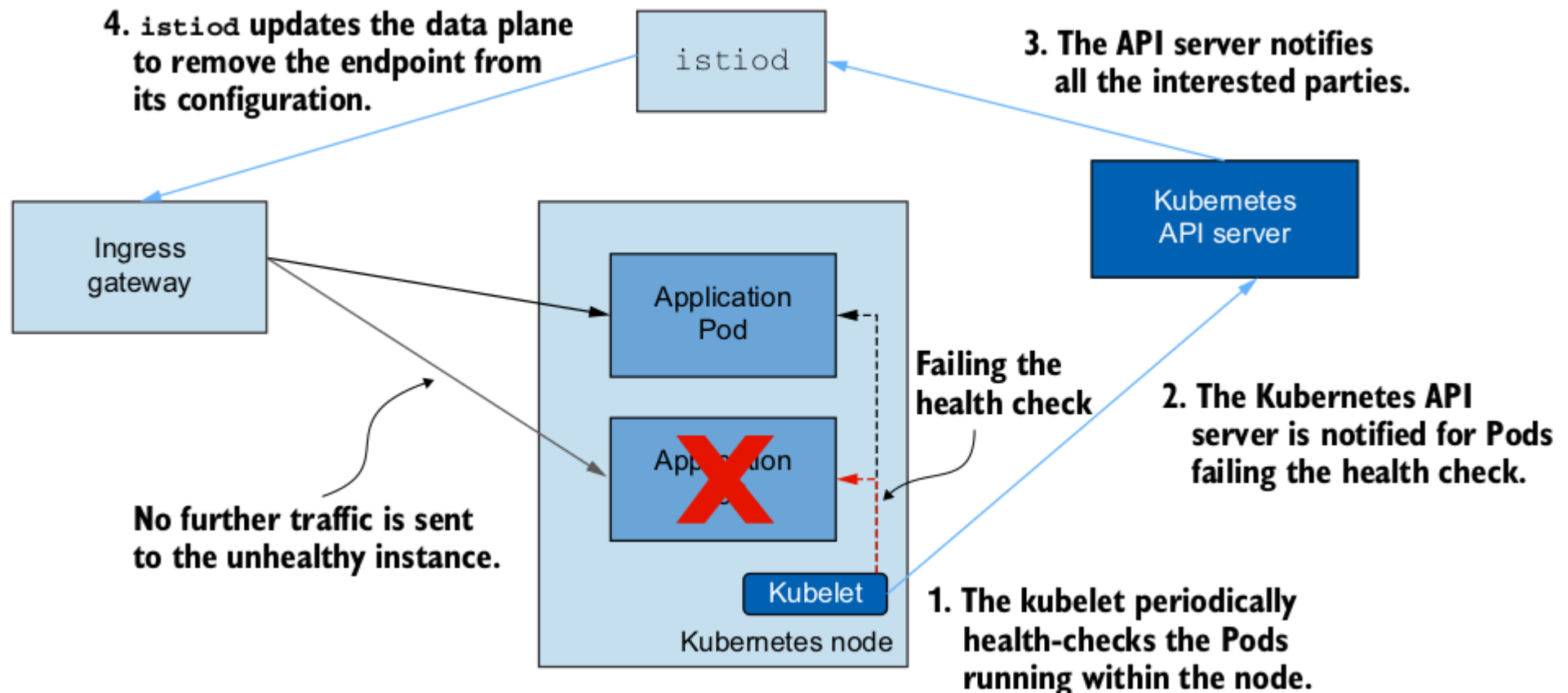


Identifier des problèmes de DataPlane

Considérant que la fonction première du contrôle 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





Etas 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

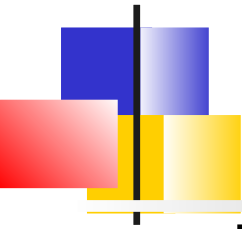


Kiali

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 référencé ¹

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



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.istioidaction.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*




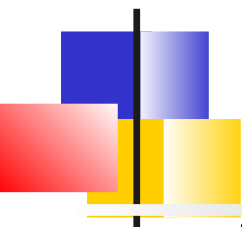
Interface d'administration Envoy

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 important 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'ou la requête vient
- **duration** : Durée de la requête
- **request_id** : Id de la requête

1. Voir <http://mng.bz/PWaP>



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))
```




Autres

Installation personnalisée
Résolution de problèmes
Monitoring de istod



Introductio

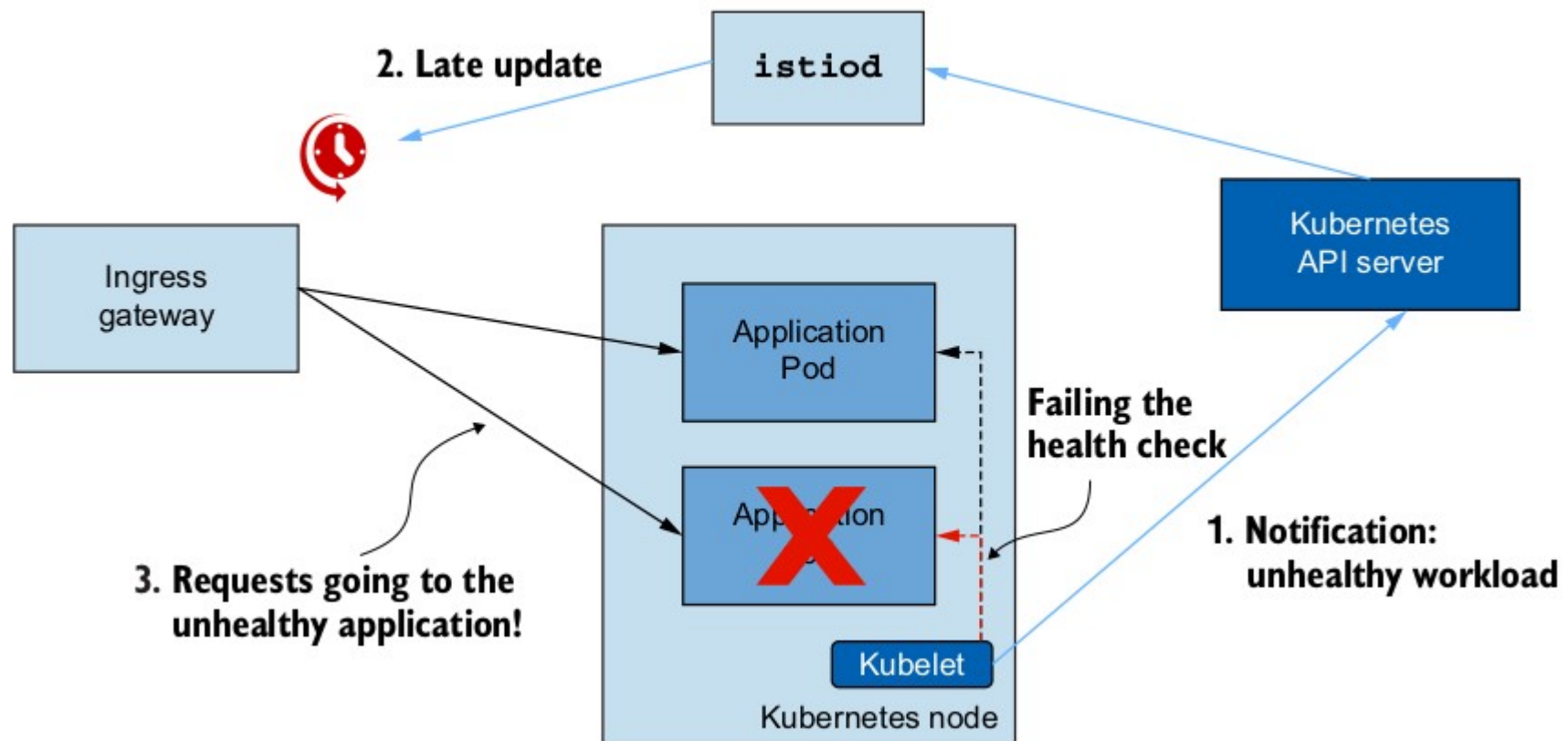
Istiod est le cerveau du service mesh

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



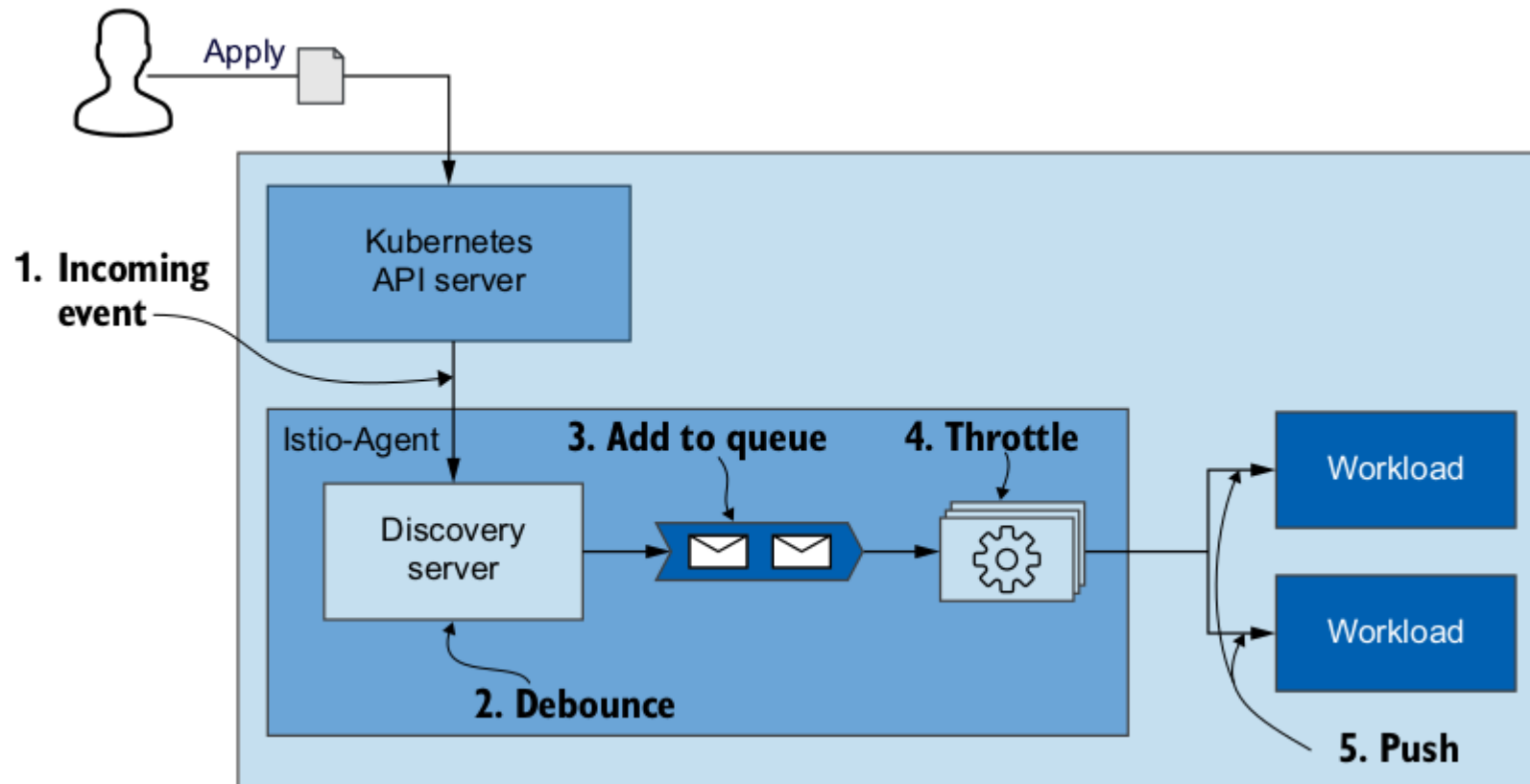


Séquence de synchronisation

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 goulots 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

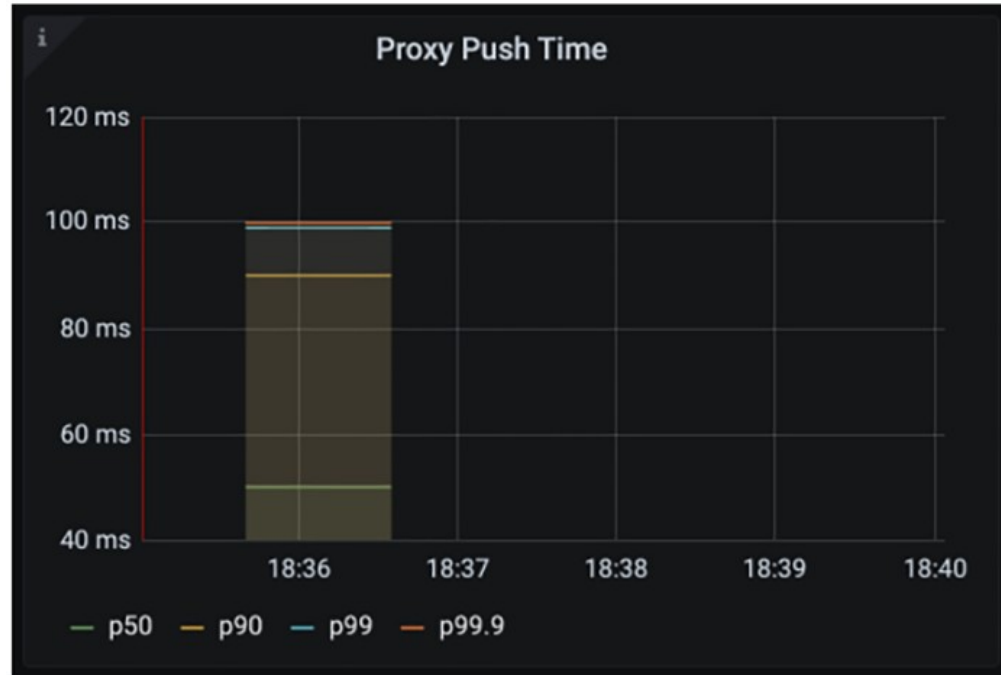
Les métriques à surveiller :

- ***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

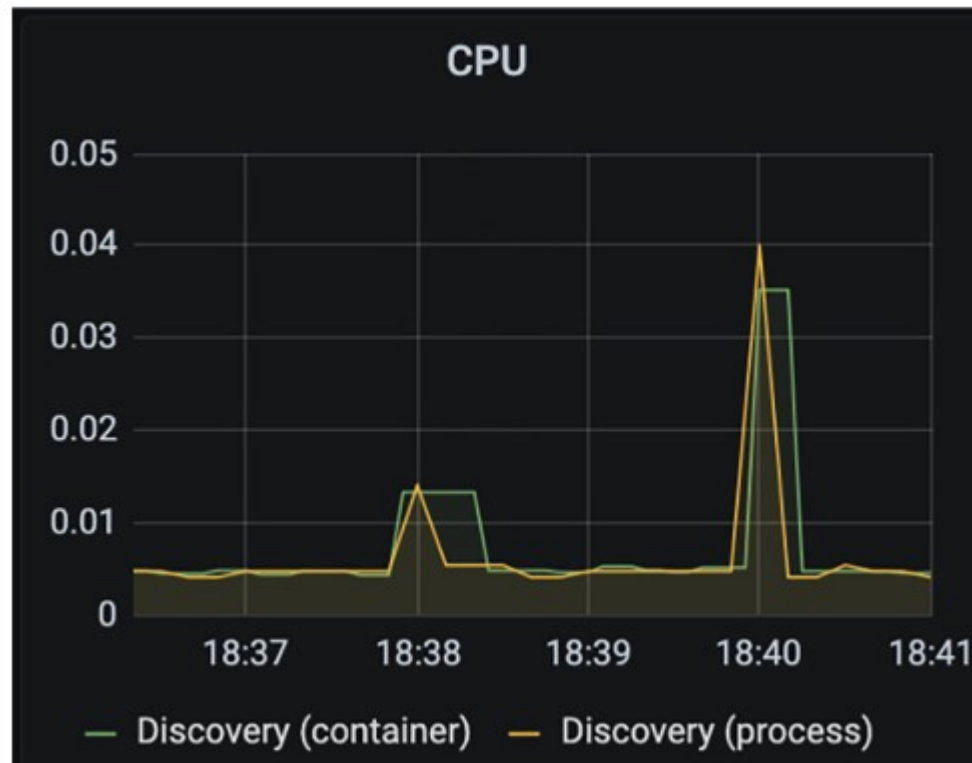
La saturation est généralement causée par le CPU.

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





Traffic

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

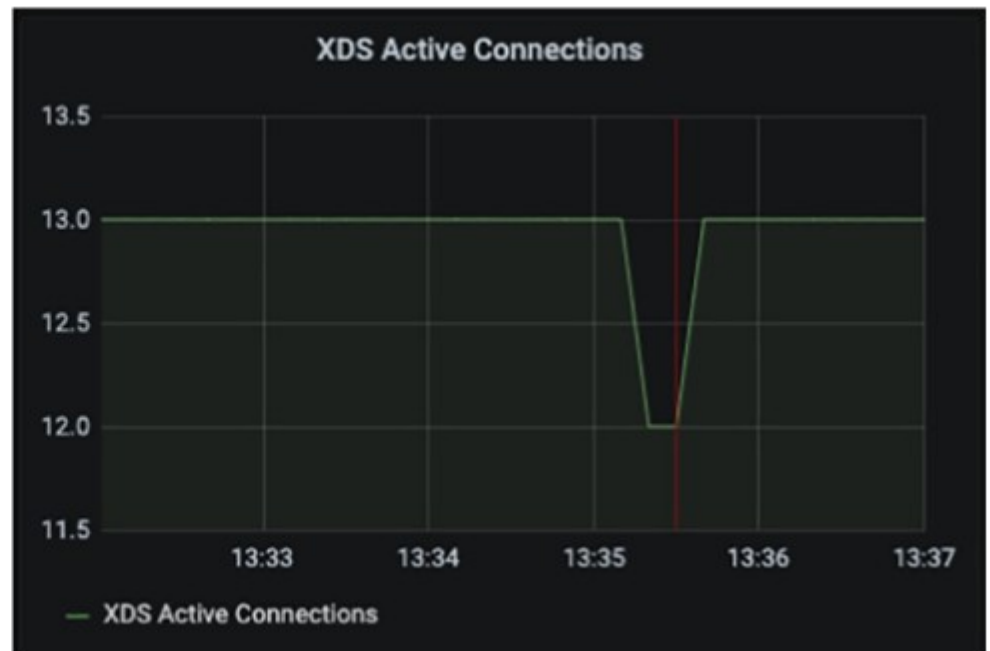
Les métriques pour le sortant :

- ***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_lds_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