Cicchino, Javier Andres

Implementación de una service mesh en una nube privada

2021

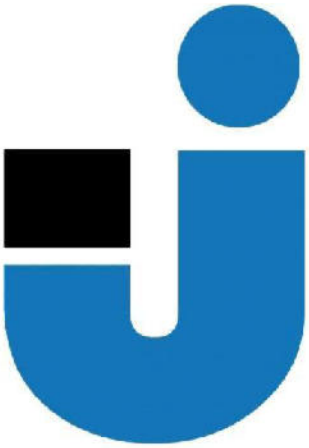
Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática



TRABAJO FINAL DE LA PRÁCTICA PROFESIONAL SUPERVISADA

Implementación de una service mesh en una nube privada

Estudiante:

Javier Cicchino

Tutores:

Dr. Ing. Morales Martín

Buenos Aires, 2021

PRÁCTICA PROFESIONAL SUPERVISADA (PPS)

Implementación de una service mesh en una nube privada

Informe Programa de Actividades

DATOS DEL ESTUDIANTE

Apellido y Nombres: Cicchino, Javier Andrés
DNI: 28.869.140
Nº de Legajo: 18.400
Correo electrónico: javiercicchino@gmail.com
Cantidad de materias aprobadas al comienzo de la PPS: 44
PPS enmarcada en el artículo 7° de la Resolución (CS) 103/16.

DOCENTE SUPERVISOR

Apellido y Nombres: Dr. Ing. Morales, Martín
Correo electrónico: martin.morales@unaj.edu.ar

DOCENTE TUTOR DEL TALLER DE APOYO A LA PRODUCCIÓN DE TEXTOS ACADÉMICOS DE LA UNAJ

Apellido y Nombres: Lic. Prof. Kelly, Carolina
Correo electrónico: kellygcarolina@gmail.com

DATOS DE LA ORGANIZACIÓN DONDE SE REALIZA LA PPS

Nombre o Razón Social: Hospital El Cruce
Dirección: Av. Calchaquí 5401
Teléfono: 4210-9000

Firma Estudiante:	Firma Docente Supervisor:	Firma docente tutor TAPTA:	Firma tutor Organizacional:

Sector: Área de Sistemas de Información

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: Ing. Rojas, Santiago

Correo electrónico: santirojas023@gmail.com

FIRMA DEL COORDINADOR DE LA CARRERA

Índice

1. Resumen

En la actualidad, existen distintas herramientas para el despliegue y orquestación de micro-servicios o contenedores. Los micro-servicios son tanto un estilo de arquitectura como un modo de desarrollar *software*. En una arquitectura de micro-servicios las aplicaciones están formadas por piezas de *software* pequeñas e independientes entre sí, pero que funcionan en conjunto para llevar a cabo la misma tarea. Cada servicio se encarga de implementar una funcionalidad completa del negocio. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, cada micro-servicio es desplegado de forma independiente y puede estar programado en distintos lenguajes y usar diferentes tecnologías de almacenamiento de datos. En relación con estos y por lo que es posible observar con reiteración, el desafío principal es, una vez desarrollados, garantizar la comunicación entre ellos, ya que, a medida que la arquitectura de servicios aumenta su complejidad, también se incrementa la complejidad de la comunicación entre estos contenedores, como, por ejemplo, y específicamente, las funcionalidades del enrutamiento, la tolerancia a fallos, la latencia, el descubrimiento de servicios, la trazabilidad distribuida, la seguridad, entre otras.

Por eso, la propuesta de la presente Práctica Profesional Supervisada (en adelante, PPS) consiste en investigar, desarrollar e implementar una arquitectura de malla de servicios (*service mesh*) en una nube privada con la finalidad de probar las características principales de una arquitectura de micro-servicios en una plataforma de *service mesh*.

Una *service mesh* es una infraestructura de comunicación entre servicios que puede contar con algunas de estas funcionalidades:

- Capacidad para asegurar la disponibilidad de la comunicación entre servicios, y de garantizar protocolos para llevar adelante esta.
- Posibilidad de ofrecer descubrimientos entre servicios.
- Generación de enrutamientos.
- Observabilidad detallada de la comunicación entre los micro-servicios.
- Seguridad para mitigar amenazas contra la información, *endpoints*, plataforma y comunicación, tanto internas como externas.
- Posibilidad de disponer de herramientas de autenticación/autorización para proteger los servicios e información.
- Capacidad para funcionar como un soporte nativo para el despliegue de contenedores.

Sin embargo, cabe aclarar que este trabajo no se centrará en los *framework* para el desarrollo de servicios de la *service mesh* sino en los orquestadores, que es donde se montan esos micro-servicios. Además, tratará de responder algunas preguntas: ¿cómo están formados, por qué son necesarios y cuándo se deben utilizar los micro-servicios? Finalmente, el presente trabajo buscará, en última instancia, probar una aplicación-ejemplo, formada por micro-servicios, y evaluar algunas de sus características.

Por último, y en relación con posibles destinatarios de este desarrollo, esta PPS se dirige, fundamentalmente, a desarrolladores de Kubernetes, programadores de micro-servicios, administradores de sistemas (*SRE*) y/o cualquiera que quiera interiorizarse en el funcionamiento e implementación de un orquestador de micro-servicios sobre una *service mesh.*

2. Objetivos

Como ya se refirió, el objetivo general de esta PPS es implementar una plataforma de malla de servicios (*service mesh*) en una nube privada con la finalidad de probar las características principales de dicha arquitectura.

Para ello, en una primera instancia, se analizarán y describirán: primero, una de las herramientas *open source* de la *service mesh* del mercado, que pueda llegar a ser de utilidad (o no) para el diseño y en relación con los micro-servicios que puede ofrecer; segundo, las plataformas de orquestación desde las que operar; y tercero, la implementación de los planos de datos, de control, el *set* de herramientas de telemetría y de administración del tráfico y de la seguridad. En una segunda instancia, se buscará instalar un clúster de orquestación de contenedores, a partir de las herramientas seleccionadas, para conectar, observar, segurizar y controlar los micro-servicios. Finalmente, la propuesta se completará con el diseño de una aplicación en la *service mesh* que tendrá como objetivo principal asegurar el correcto funcionamiento del orquestador de la red de micro-servicios.

3. Tareas por ejecutar

Las tareas que se pretenden llevar a cabo son las siguientes:

- Investigar algunas de las herramientas de orquestación de micro-servicios del mercado.
- Definir el *set* de herramientas que va a utilizarse.

- Definir el clúster donde se desarrollará la instalación.
- Instalar y configurar el clúster.
- Instalar y configurar el almacenamiento persistente.
- Implantar las herramientas de orquestación de micro-servicios.
- Desplegar una aplicación de micro-servicios y hacer pruebas de telemetría, administración de tráfico, balanceo de carga, etcétera.

4. Cronograma de trabajo

Semana de trabajo	**Actividades**
1ra	Recopilar información. Analizar la plataforma de orquestación de contenedores seleccionada para el desarrollo de la PPS. Distinguir los pros y contras de las herramientas de *service mesh*.
2da	Definir el clúster donde se va instalar la plataforma.
3ra	Instalar el clúster.

4ta	Probar el clúster.
5ta	Instalar las herramientas de orquestación.
6ta	Probar las herramientas de orquestación.
7ma	Desplegar una aplicación en la *service mesh*.
8va	Recopilar y analizar la información.
9na y 10ma	Escribir el informe.

5. Conceptos, Tecnologías y Herramientas utilizadas

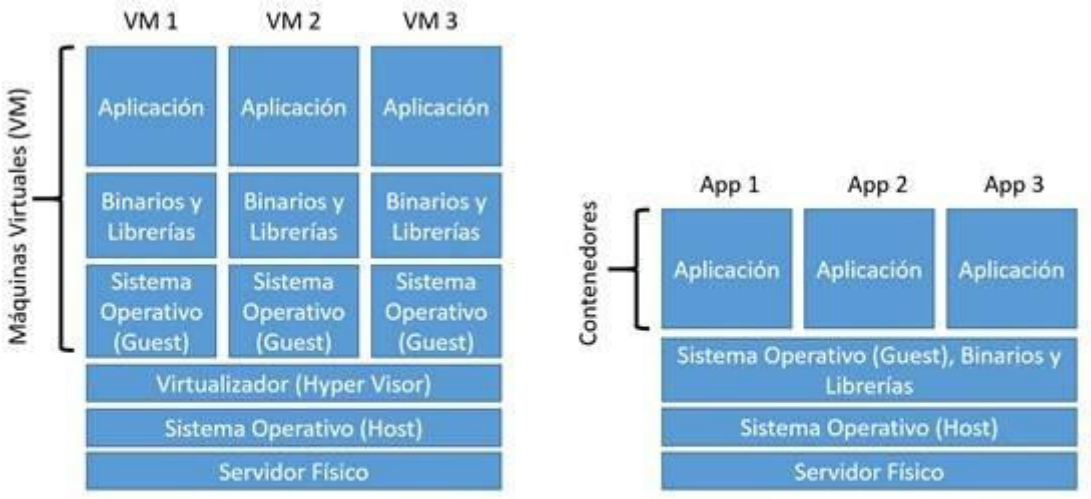
A continuación, se listan los conceptos, tecnologías y herramientas que se utilizaron en esta PPS para investigar, desarrollar e implementar una arquitectura de malla de servicios (*service mesh*):

- Contenedores (*Containers*)
- Docker (*software*)
- Orquestadores de contenedores
- Kubernetes (*software*)
- Aplicaciones monolíticas
- Arquitectura de micro-servicios
- *Service mesh*
- Istio (*software*)

5.1 Contenedores (*Containers*)

Los contenedores de *software* son ambientes de ejecución livianos que proveen a las aplicaciones con los archivos, variables y librerías que necesitan para operar. Se utilizan para garantizar que una aplicación se ejecute correctamente cuando cambie su entorno, con una reducción al mínimo de las fallas posibles y una maximización de su portabilidad. Los

contenedores se asemejan a la virtualización clásica, aunque funcionan en un plano más bajo: mientras que las máquinas virtuales tradicionales habilitan la virtualización de la infraestructura computacional, los *containers* permiten la virtualización de las aplicaciones. A diferencia de las máquinas virtuales, los contenedores utilizan el sistema operativo (SO) de su *host*, en lugar de integrar uno propio.



Máquinas Virtuales versus Contenedores

Imagen: Diferencia conceptual entre máquinas virtuales y contenedores
Fuente: Recuperado de https://www.fayerwayer.com/2016/06/maquinas-virtuales-vs-contenedores-que-son-y-como-elegir-entre-estas-tecnologias/ (2020)

En términos simples, los contenedores crean la percepción de un ambiente aislado, exclusivo para la aplicación, y comparten el mismo *kernel* que el sistema operativo anfitrión, mientras que, en la virtualización "tradicional" de máquinas virtuales, la

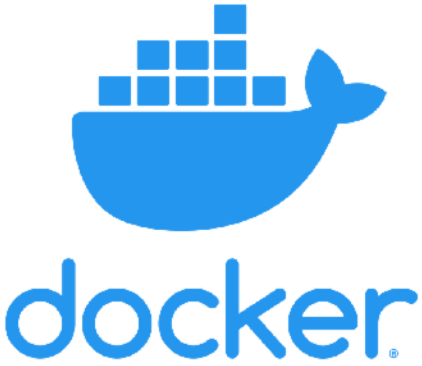
aplicación se ejecuta en un sistema operativo virtualizado, donde convive con otros aplicativos.

Los contenedores representan un mecanismo de empaquetado lógico, donde las aplicaciones tienen descrito todo lo que necesitan para ejecutarse en un pequeño archivo de configuración. Y, además, presenta la ventaja de poder ser versionado, reutilizado y replicado fácilmente por otros desarrolladores o por los administradores de sistemas que tengan que escalar esas aplicaciones, sin necesidad de conocer internamente cómo funciona nuestra aplicación.

Los contenedores no son micro-servicios, pero se puede decir que son la herramienta más adecuada para implementar micro-servicios.

5.2 Docker (*software*)

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de *software*, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos. Este contenedor empaqueta, de forma ligera, todo lo necesario para que uno o más procesos funcionen correctamente, como puede ser: el código de la aplicación, las herramientas del sistema, las bibliotecas del sistema



necesarias para correr la aplicación, las dependencias, etc. Docker utiliza características de aislamiento de recursos del *kernel* Linux para permitir que "contenedores" independientes se ejecuten dentro de una sola instancia de Linux, evitando la sobrecarga de iniciar y mantener máquinas virtuales.

Esto garantiza que siempre se podrá ejecutar, independientemente del entorno en el que queramos desplegarlo. El soporte del *kernel* Linux para los *namespaces* aísla la vista que tiene una aplicación de su entorno operativo, incluyendo árboles de proceso, red, ID de usuario y sistemas de archivos montados, mientras que los *cgroups* del *kernel* proporcionan aislamiento de recursos, incluyendo la CPU, la memoria, el bloque de E/S y de la red.

Docker también incluye la biblioteca *libcontainer*, como su propia manera de utilizar directamente las facilidades de virtualización que ofrece el *kernel* Linux, además de utilizar las interfaces abstraídas de virtualización mediante *libvirt*, *LXC* (*Linux Containers*) y *systemd-nspawn*.

5.3 Orquestadores de contenedores

Los orquestadores de contenedores son herramientas que agrupan sistemas para formar clústeres, en los que se habilita un ambiente para la automatización y la escalabilidad en la implementación y administración de contenedores, para finalmente cumplir ciertos requisitos específicos como:

- Tolerancia a fallos
- Alta disponibilidad
- Posibilidad de realizar cambios y operaciones en “caliente”

- Escalabilidad bajo demanda
- Acceso desde el mundo exterior
- Optimización de recursos
- Descubrimiento automático

En la actualidad, hay varias soluciones disponibles para este tipo de entornos. Podemos nombrar, entre otros orquestadores posibles, a:

- Docker Swarm
- Kubernetes
- Mesos Marathon
- Nomad
- Shipyard

Los orquestadores permiten facilitar tareas, especialmente cuando el número de contenedores es del orden de los cientos y/o miles que se ejecutan en una infraestructura global y distribuida.
Entre las tareas que puede realizar un orquestador se encuentran:

- Crear un clúster a partir de un grupo de nodos (*hosts*)
- Comunicar contenedores entre sí
- Asignar almacenamiento a un contenedor
- Gestionar y optimizar el uso de recursos
- Permitir y gestionar políticas para proteger el acceso a las aplicaciones que se ejecutan dentro de los contenedores

En esta PPS vamos a utilizar, como orquestador de contenedores, a Kubernetes que es el orquestador de contenedores más utilizado.

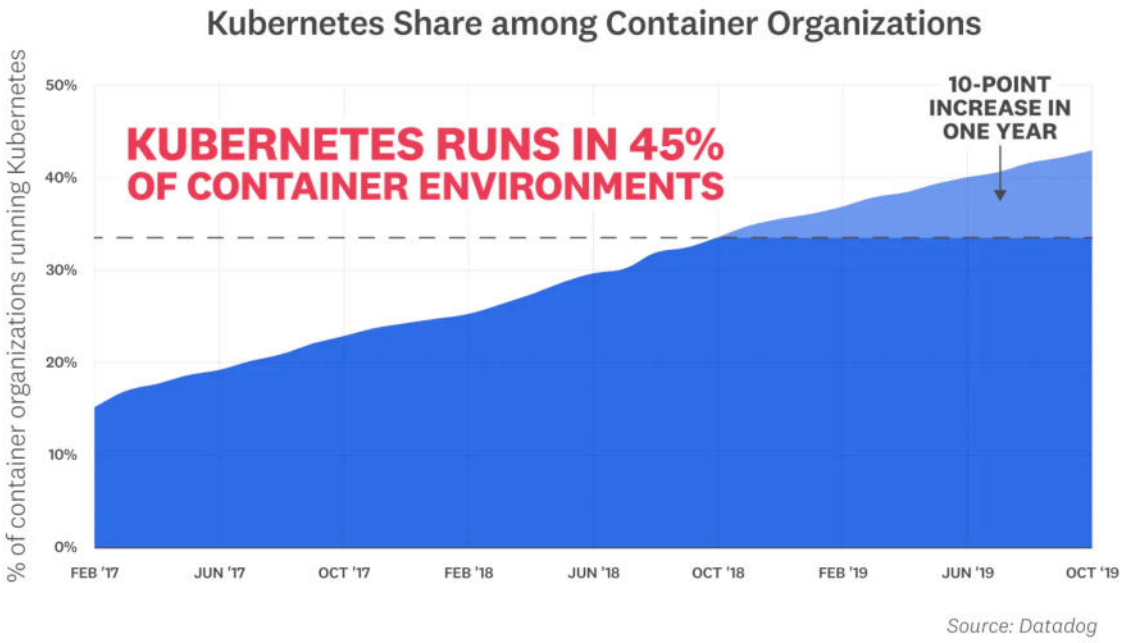


Imagen: Cuota de uso de Kubernetes entre organizaciones de contenedores. Octubre de 2019: el 45% de los contenedores corren en Kubernetes.

Fuente: Recuperado de https://www.datadoghq.com/container-report/ (2020)

5.4 Kubernetes (*software*)

También conocido como K8s, es un sistema de orquestación de contenedores de código abierto, usado para automatizar la implementación, el escalado y la administración de aplicaciones informáticas.

Fue diseñado originalmente por Google y ahora es mantenido por Cloud Native Computing Foundation, quien señala que su objetivo es proporcionar una "plataforma para automatizar la implementación, el escalado y las operaciones de contenedores de aplicaciones en grupos de hosts". Funciona con una variedad de herramientas de contenedor, incluido Docker.

Muchos servicios en la nube ofrecen una plataforma basada en Kubernetes o una infraestructura como servicio (PaaS o IaaS), en la que Kubernetes se puede implementar como un servicio de suministro de plataforma. Muchos proveedores también proporcionan sus propias adaptaciones de Kubernetes. Entre ellos, podemos nombrar: Azure Kubernetes Service (AKS) de Microsoft, Amazon EKS, Google Kubernetes Engine (GKE), entre otros.

5.5 Aplicaciones monolíticas a micro-servicio

El estilo arquitectónico monolítico consiste en crear una aplicación autosuficiente que contenga absolutamente toda la funcionalidad necesaria para realizar la tarea para la cual fue diseñada, sin contar con dependencias externas que complementan su funcionalidad. En este sentido, sus componentes trabajan juntos, compartiendo los mismos recursos y memoria. En pocas palabras, una aplicación monolítica es una unidad cohesiva de código.

Un monolítico podría estar construido como una sola unidad de *software* o creado a partir de varios módulos o librerías, pero lo que lo distingue es que, al momento de compilarse, se empaqueta como una sola pieza, de tal forma que todos los módulos y librerías se empaquetan junto con la aplicación principal.

En contraposición, un micro-servicio es un pequeño programa que se especializa en realizar una pequeña tarea y se enfoca únicamente en esta. Por ello, decimos que los micro-servicios son "altamente cohesivos", pues todas las operaciones o funcionalidades que tiene dentro están extrema y directamente relacionadas para resolver un único problema.

En este sentido, podemos decir que los micro-servicios son todo lo contrario a las aplicaciones monolíticas, pues en una arquitectura de micro-servicios se busca, fundamentalmente, desmenuzar una gran aplicación en muchos y pequeños componentes que realizan, de forma independiente, una pequeña tarea de la problemática general.

Una de las grandes ventajas de los micro-servicios es que son componentes totalmente encapsulados, lo que quiere decir que la implementación interna no es de interés para los demás componentes, lo que permite, en última instancia, que estos evolucionen a la velocidad requerida. Además, cada micro-servicios puede ser desarrollado con tecnologías totalmente diferentes; incluso, es normal que se utilicen diferentes bases de datos.

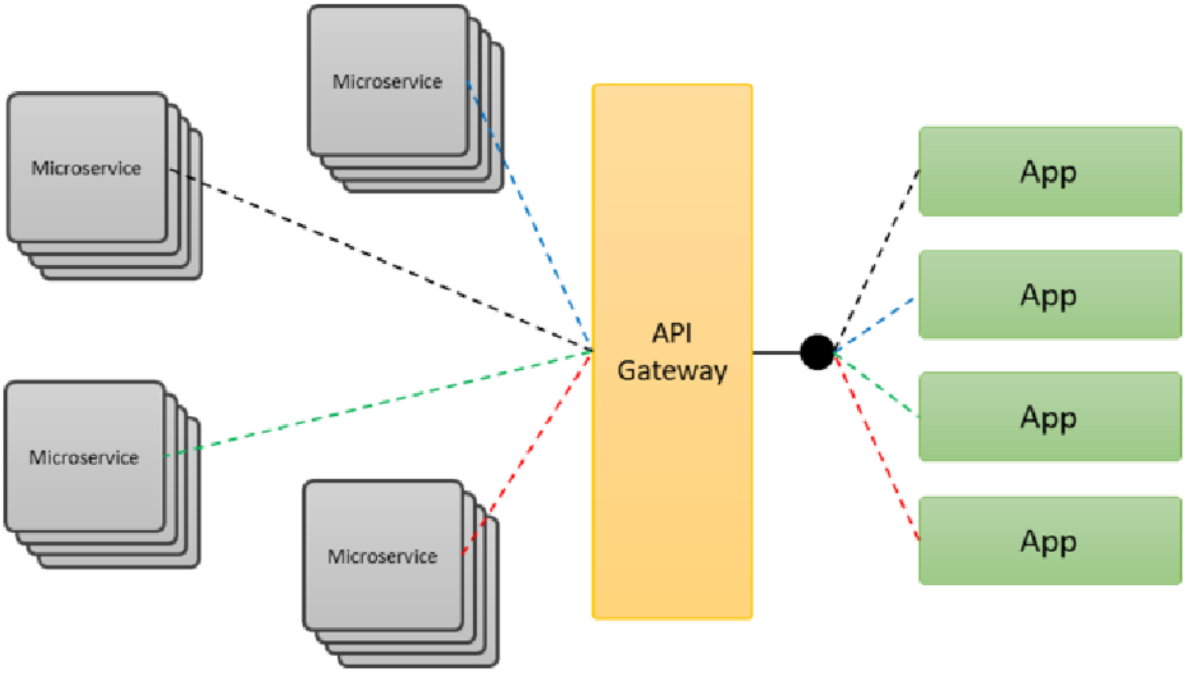


Imagen: Arquitectura de micro-servicios.

Fuente: Recuperado de https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/microservicios (2020).

Arquitectura de micro-servicios

Ventajas

La arquitectura de un micro-servicios presenta varias ventajas, a saber:

- Alta escalabilidad: se trata de un estilo arquitectónico diseñado para ser escalable, pues permite montar numerosas instancias del mismo componente y balancear la carga entre todas las instancias.

- Agilidad: debido a que cada micro-servicios es un proyecto independiente, permite que el componente tenga un ciclo de desarrollo diferente del resto, lo que permite, a su vez, que se puedan hacer despliegues rápidos a producción, sin afectar al resto de componentes.

- Facilidad de despliegue: las aplicaciones desarrolladas como micro-servicios encapsulan todo su entorno de ejecución, posibilitando, de esa manera, ser desplegadas sin necesidad de dependencias externas o requerimientos específicos de hardware.

- Testabilidad: los micro-servicios son especialmente fáciles de probar, pues su funcionalidad es tan reducida que no requiere mucho esfuerzo; además, su capacidad para exponer o brindar servicios hace que sea más fácil crear casos específicos, para probar esos servicios.

- Fácil de desarrollar: debido a que los micro-servicios tienen un alcance muy corto, es fácil para un programador comprender el alcance del componente; además, cada micro-servicios puede ser desarrollado por una sola persona o un equipo de trabajo muy reducido.

- Reusabilidad: es la médula espinal de la arquitectura de micro-servicios, pues se basa en la creación de pequeños componentes que realizan una única tarea, lo que hace que sea muy fácil de reutilizar por otras aplicaciones o micro-servicios.

- Interoperabilidad: debido a que los micro-servicios utilizan estándares abiertos y ligeros para comunicarse, cualquier aplicación o componente puede comunicarse con ellos, sin importar en qué tecnología está desarrollada/o.

Desventajas

También, la arquitectura de un micro-servicios presenta varias desventajas, a saber:

- Performance: la naturaleza distribuida de los micro-servicios agrega una latencia significativa que puede ser un impedimento para aplicaciones donde la performance es lo más importante; por otra parte, la comunicación por la red puede generar incluso más retrasos que el proceso en sí.
- Múltiples puntos de falla: la arquitectura distribuida de los micro-servicios hace que los puntos de falla de una aplicación se multipliquen, pues cada comunicación entre micro-servicios tiene una posibilidad de fallar, lo cual hay que gestionar adecuadamente.
- Trazabilidad: la naturaleza distribuida de los micro-servicios puede complicar el recuperar y realizar una traza completa de ejecución de un proceso, pues cada micro-servicio arroja, de forma separada, su traza o logs que, luego, deben de ser recopilados y unificados para tener una traza completa.
- Madurez del equipo de desarrollo: esta arquitectura debe ser implementada por un equipo maduro de desarrollo y con un tamaño adecuado, pues los micro-servicios agregan muchos componentes que deben ser administrados, lo que puede ser muy complicado para equipos poco maduros.

5.6 Malla de servicios (*Service mesh*)

Las *service mesh* se volvieron populares con el inicio de las arquitecturas de micro-servicios basadas en clústeres, como Kubernetes, el que por sí solo no es suficiente para manejar la observabilidad, la administración del tráfico y la seguridad de los micro-servicios. Una malla de servicios no reemplazará a Kubernetes; en realidad, será una capa adicional de *software* que se implementa junto con Kubernetes.

Cualquier tipo de arquitectura distribuida, en la que hay múltiples componentes de *software* que se comunican entre sí, se beneficiaría con una malla de servicios.

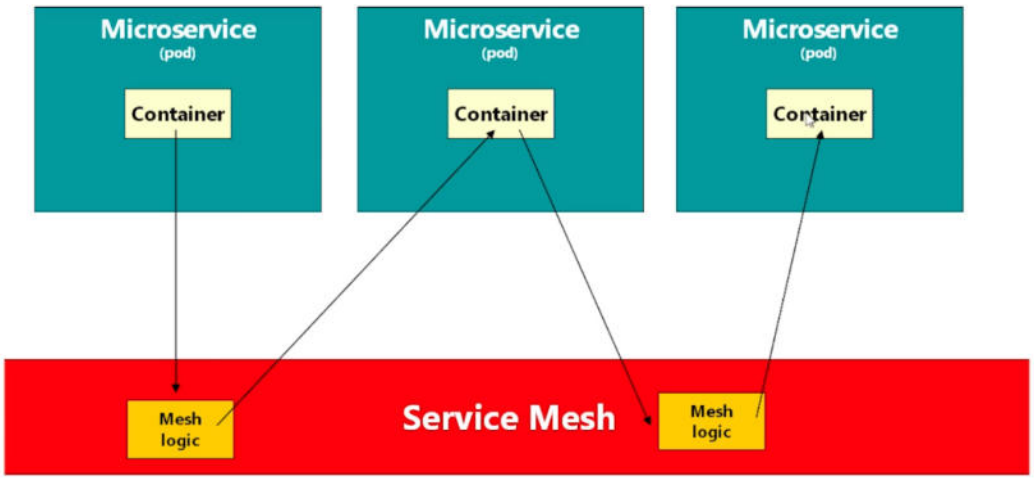


Imagen: Arquitectura de una *service mesh*
Fuente: Recuperado de https://www.udemy.com/course/istio-hands-on-for-kubernetes/ (2020)

Cuando se utiliza solo Kubernetes estándar, no hay ningún tipo de visibilidad o control sobre las conexiones entre los contenedores.

Una malla de servicios se puede pensar como una capa de *software* que se encuentra debajo de todos los micro-servicios (también llamados "*pods"* en Kubernetes) del sistema. Pero, conceptualmente, lo que hace es configurar las comunicaciones entre micro-servicios, de alguna manera, para que todo el tráfico de red que se ejecuta a través del clúster sea enrutado, gracias a esta capa de *software*, en la *service mesh*.

Entonces, cuando un micro-servicio realiza una llamada a otro micro-servicio, en un clúster de Kubernetes puro lo haría de forma directa, pero, en una malla de servicios, esa llamada se redirige a través de la capa de *software* de la malla de servicios y, luego, esta será responsable de dirigir la llamada hacia el contenedor de destino.

En pocas palabras, una *service mesh* es una infraestructura de comunicación entre servicios. La responsabilidad principal de la *service mesh* es entregar las solicitudes de un servicio A a un servicio B de una manera confiable, segura y oportuna. Desde el punto de vista funcional, esto es algo similar a la función de un ESB (siglas en inglés de *Enterprise Service Bus*, componente esencial de la arquitectura orientada a servicios o SOA), donde se interconectan sistemas heterogéneos para la comunicación de mensajes. La diferencia con la *service mesh* es que, en esta, no hay un componente centralizado sino una red distribuida de componentes.

Ejemplo del funcionamiento de una *service mesh*

Un micro-servicio quiere enviar una llamada a otro micro-servicio. Entonces, se realiza una llamada de red. En un clúster de Kubernetes se traduciría como una llamada entre dos

contenedores (o *pods*) pero, con una *service mesh*, se redirigen todas las llamadas a través de ella.

En otras palabras, el micro-servicio envía la solicitud a la *service mesh* y se enruta hacia el otro micro-servicios. Esto parece ineficiente pero el punto es que la *service mesh* puede implementar alguna lógica, ya sea antes de que la llamada se arraigue en el contenedor de destino, o también podría ejecutarse después de que el contenedor de destino se complete y haya regresado la llamada. Un ejemplo de la lógica de funcionamiento de una *service mesh* podría ser la telemetría, que consiste en recopilar métricas sobre el estado general del clúster.

Otra de las funcionalidades de una *service mesh* es lo que se conoce como rastreo (*tracing*) en ambientes distribuidos, que permite identificar llamadas en cadenas complejas (pero, a su vez, todas conectadas, forman una sola llamada). Por ejemplo, el micro-servicio A hace una llamada al micro-servicio B, y este último necesita hacer otra llamada, a un micro-servicio C, para responder al primero (el A, en este ejemplo). Cada una de estas llamadas es independiente pero forma parte de la misma cadena de mensajes. En algunos casos, es necesario que la *service mesh* muestre exactamente qué era esa cadena, cuánto tiempo tomó cada paso y qué estaba sucediendo en esta, para conocer posibles errores, *timeouts*, problemas de rendimientos, etc. A este conjunto de técnicas se lo conoce como "rastreo en ambientes distribuidos".

Patrón *sidecar*

Para poder llevar a cabo toda la funcionalidad que debe aportar una *service mesh*, se implementan algunos de los patrones de diseño y de aplicaciones distribuidas, pero, en

lugar de aplicarlos sobre el código (como venía siendo hecho con frecuencia), se los aplica sobre la propia infraestructura.

En el patrón *sidecar*, la funcionalidad de un proceso (servicio) principal se extiende, o mejora, mediante un proceso "paralelo", con poco acoplamiento entre ambos.

El comportamiento es similar al de un *proxy* que proporciona al proceso principal todos los servicios "*commodity*" de infraestructura que necesita (por ejemplo, descubrimiento de servicios, *circuit breaker*, *fault injection*, etc.).

Este patrón es particularmente útil cuando se utiliza Kubernetes como plataforma de orquestación de contenedores. En Kubernetes se utilizan los "*pods*", cada uno de los cuales está compuesto por uno o más contenedores de aplicaciones.

Por lo tanto, un *sidecar* es un contenedor auxiliar que se inyecta en los *pods* y su propósito es dar soporte al contenedor principal. Es importante tener en cuenta que el *sidecar*, por sí solo, no sirve para nada, sino que debe combinarse con uno o más contenedores principales. En general, el contenedor *sidecar* es reutilizable y puede combinarse con numerosos tipos de contenedores principales.

Plano de datos (*Data Plane*)

Una aplicación en una *service mesh* está formada por un conjunto de servicios independientes en el que cada uno de ellos está desplegado, junto con un *proxy-sidecar*, en un mismo *pod* de Kubernetes.

Además, cada micro-servicio se comunica únicamente con su propio *proxy-sidecar* y, en lugar de comunicarse directamente entre ellos, son estos últimos (los *sidecars*) los que terminan siendo los responsables de la entrega fiable de las peticiones, a través de una topología o arquitectura que puede ser todo lo compleja que se requiera.

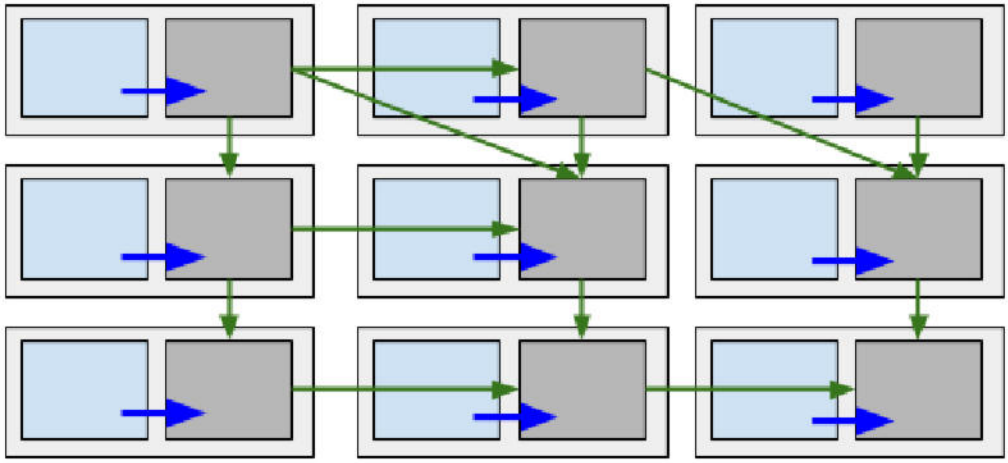


Imagen: Proxy *sidecar* (color gris) junto a los micro-servicios (celestes). Los micro-servicios se comunican entre ellos a través de los *sidecars*.
Fuente: Recuperado de
https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/ (2020)

Ahora podemos ver, con más claridad, cómo el diagrama resultante representa la "malla de conectividad" subyacente:

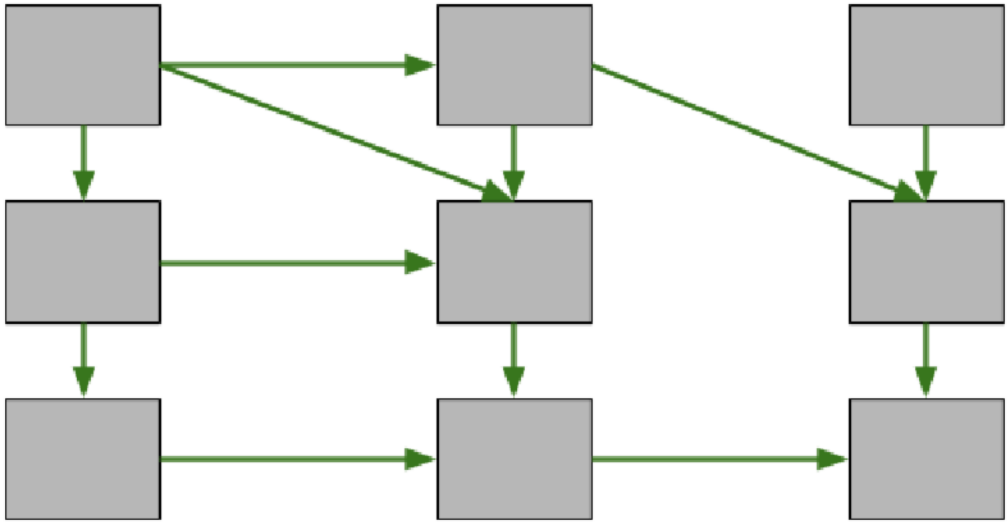


Imagen: Proxy *sidecar* (color gris). Los micro-servicios se comunican entre ellos a través de los *sidecars*.
Fuente: Recuperado de
https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/ (2020)

A este conjunto de micro-servicios y *proxy-sidecars* (en definitiva, a esta malla) se le asigna el nombre de *"Data Plane"* o "Plano de datos". La responsabilidad del "plano de datos" es asegurar que las solicitudes sean entregadas desde el micro-servicio A al micro-servicio B de una manera confiable, segura y oportuna, siendo el encargado de proporcionar las siguientes funcionalidades:

- Capacidad para asegurar la disponibilidad de la comunicación entre servicios
- Posibilidad de ofrecer descubrimientos entre servicios
- Generación de enrutamientos y balanceos de carga
- Observabilidad detallada de la comunicación entre los micro-servicios
- Securización del canal de comunicación
- Securización y control de acceso (autenticación / autorización)

Plano de control (*Control Plane*)

Por otro lado, se encuentra la otra pieza que forma parte de la solución, la cual recibe el nombre de *"Control Plane"* o "Plano de Control". Este elemento es el encargado de gestionar y monitorizar todas las instancias de *sidecars*, siendo el espacio ideal para implementar políticas de control, recolección de métricas, monitorización, etc. Es una pieza obligatoria para el correcto funcionamiento de una *service mesh*.

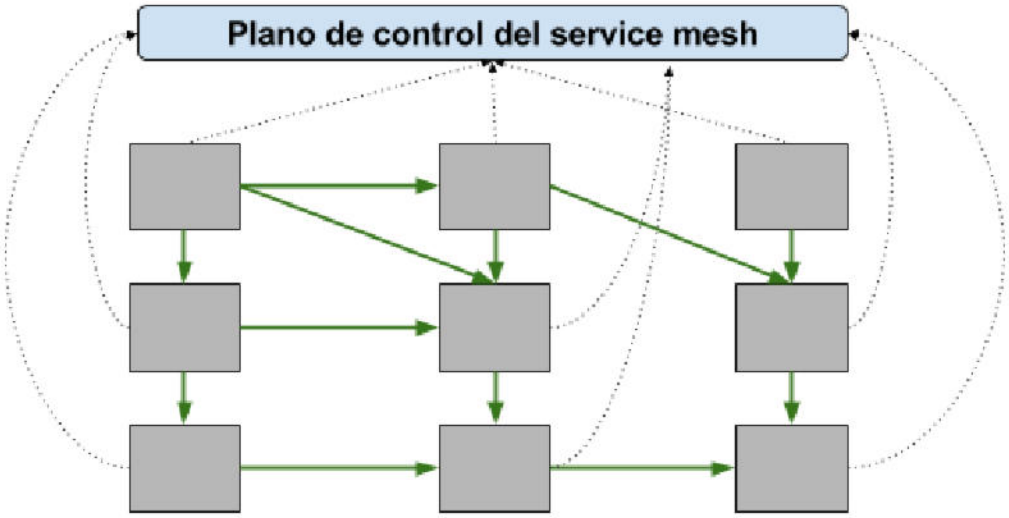


Imagen: Plano de control.
Fuente: Recuperado de
https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/ (2020)

Además, una *service mesh* puede funcionar como capa de aplicación, no sólo de red. Esto hace que pueda tener la capacidad de obtener información más detallada sobre las peticiones, como, por ejemplo, poder distinguir entre peticiones que finalizan con un error 500 o un 404 (lo cual no sería posible si solo funcionara como capa de red). De este modo,

se puede hacer uso de esta información para diferentes ámbitos (trazabilidad, *health checking*, etc.).

A pesar de que las funcionalidades mencionadas anteriormente se proporcionan dentro del plano de datos mediante los *proxy-sidecar*, la configuración global de estas funcionalidades se realiza dentro del plano de control. Es el plano de control quien toma todos los *proxy-sidecar* sin estado y los convierte en un sistema distribuido.

En una *service mesh*, el plano de control es, por tanto, el responsable de configurar la red de *proxy-sidecar*. Las funcionalidades del plano de control incluyen la configuración de:

- Enrutamiento.
- Balanceo de carga.
- *Circuit-breaker*, políticas de reintentos, *time-outs*.
- Despliegues.
- Descubrimiento de servicios.

5.7 Istio

Istio es una implementación de *service mesh* con tecnología de *open source*, que permite controlar el intercambio de datos entre los micro-servicios. Incluye una API que permite integrar cualquier plataforma de registro, telemetría o sistema de políticas.

La arquitectura de Istio se divide en el plano de datos y el plano de control. En el plano de datos, el soporte de esta plataforma se agrega a un servicio mediante la implementación de un proxy de *sidecar* (Envoy) dentro de su entorno. Este proxy *sidecar* se encuentra junto a

un micro-servicio y envía las solicitudes desde y hacia otros proxies. En conjunto, dichos proxies forman una red que intercepta la comunicación de red entre los micro-servicios. El plano de control gestiona y configura los proxies para enrutar el tráfico. Este plano también configura los elementos para aplicar las políticas y recopilar datos de telemetría.

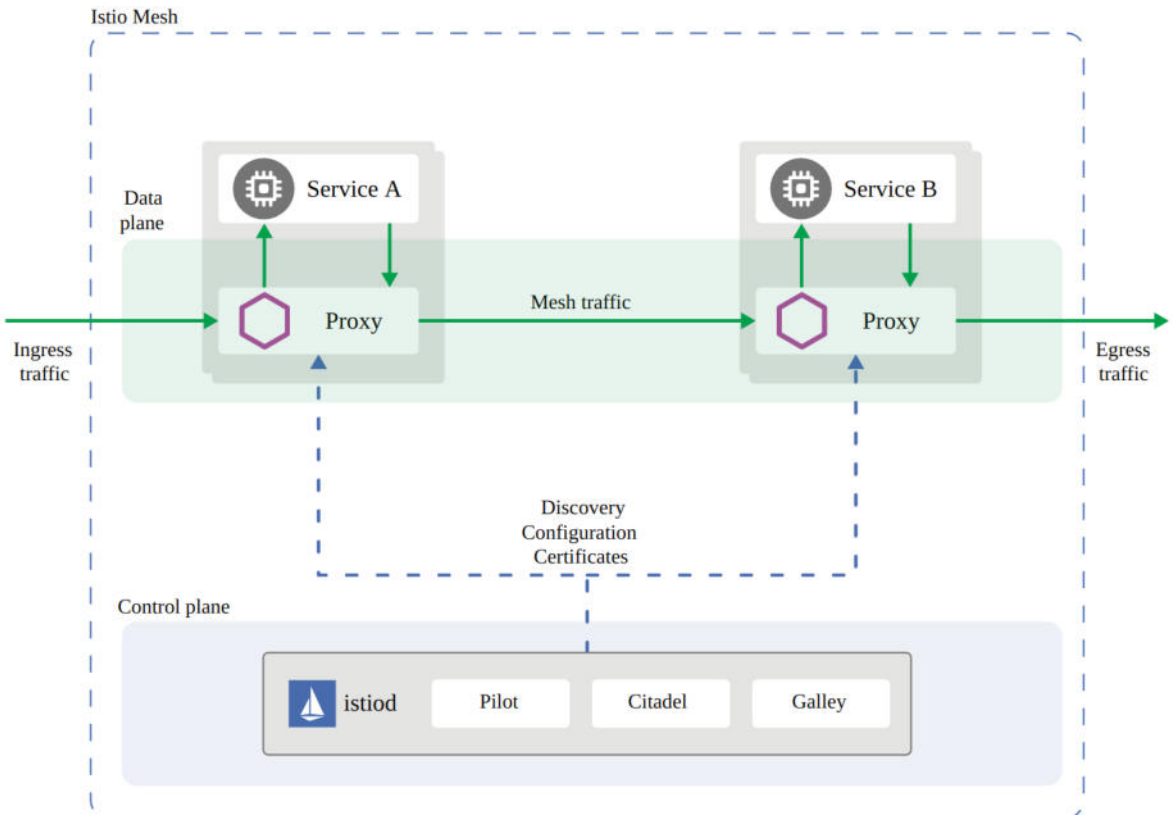


Imagen: Arquitectura de Istio. El plano de control, en azul y el plano de datos, en verde.
Fuente: Recuperado de https://istio.io/latest/docs/ (2020)

Una de las características principales de Istio es que permite implementar balanceo de carga, autenticación de servicio a servicio, monitoreo (entre otras funcionalidades), con poco o ningún cambio en el código de cada micro-servicio.

A través del proxy *sidecar* que intercepta todas las comunicaciones de red entre microservicios, se pueden implementar distintas funcionalidades, que incluyen:

- Balanceo de carga automático para tráfico HTTP, gRPC, WebSocket y TCP.
- Control del comportamiento del tráfico con reglas de enrutamiento, reintentos, conmutaciones por error y *fault injections*.
- Capa de políticas conectable y una API de configuración que permite un control de acceso, de límites de velocidad y de cuotas.
- Métricas, registros y seguimientos automáticos para todo el tráfico dentro de un clúster, incluida la entrada y salida de este.
- Comunicación segura, de servicio a servicio, en un clúster con autenticación y autorización basadas en identidad.

Envoy

Istio usa una versión del proxy Envoy, que es un proxy desarrollado en C++ y que se utiliza para interceptar todo el tráfico entrante y saliente para todos los servicios en la *service mesh*. Los proxies Envoy son los únicos componentes de Istio que interactúan con el tráfico del plano de datos. A su vez, se implementan como *sidecars* para los servicios, lo que aumenta lógicamente las funcionalidades de los servicios con muchas de las características integradas de Envoy, por ejemplo:

- Descubrimiento dinámico de servicios.
- Balanceo de carga.
- Terminación de TLS.
- Proxies HTTP / 2 y gRPC.
- *Circuit breaker*.
- Controles de salud.
- Implementaciones por etapas con división de tráfico basada en %.
- *Fault injections*.
- Métricas.

Istiod

Istiod es la herramienta que proporciona Istio para la detección de servicios, configuración y gestión de certificados. Convierte las reglas de enrutamiento de alto nivel que controlan el comportamiento del tráfico en configuraciones específicas de Envoy y las propaga a los *sidecars* en tiempo de ejecución. Pilot abstrae los mecanismos de descubrimiento de servicios específicos de la plataforma y los sintetiza en un formato estándar que cualquier *sidecar*, que cumpla con la API Envoy, puede consumir.

Funciones principales:

- **Observabilidad**

La observabilidad en sistemas de *software* se define por tres áreas: el monitoreo, los registros y el rastreo. Estas tres áreas pueden estar solapadas pero cada una tiene una serie de características que la definen. La *service mesh* de Istio provee herramientas para llevar a cabo la observabilidad teniendo en cuenta estas tres grandes áreas.

La característica fundamental del monitoreo es dar información agregable. Puede estar representada por una serie de contadores, en un rango de tiempo determinado e, incluso, con valores en tiempo real. No suelen proveer información detallada para un evento pero indican en qué rango o momento sucedió. Para la información más detallada se utilizan los registros.

El registro, por otra parte, se encarga de eventos discretos, que además suelen ser enriquecidos con un contexto.

El rastreo está relacionado con los *request*. Un *request* es una solicitud en el sistema que tiene un inicio y un fin. El rastreo, por su parte, se encarga de observar esas solicitudes durante su ciclo de vida. En un sistema distribuido como la *service mesh*, una herramienta de rastreo de *request* cumple una función esencial para analizar problemas de rendimiento y/o errores.

Métricas

Istio provee una herramienta para visualizar métricas de la *service mesh* utilizando, como tablero de mando, el *software* Grafana.

```
javier@HEC0715:~/istio-curso$ kubectl -n istio-system get svc prometheus
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
prometheus   ClusterIP   10.96.184.172   <none>        9090/TCP   5d22h
javier@HEC0715:~/istio-curso$ kubectl -n istio-system get svc grafana
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
grafana   ClusterIP   10.104.225.25   <none>        3000/TCP   5d22h
javier@HEC0715:~/istio-curso$ istioctl dashboard grafana
http://localhost:3000
```

Imagen: Línea de comando para ingresar al tablero de mando Grafana.
Fuente: Producción propia.

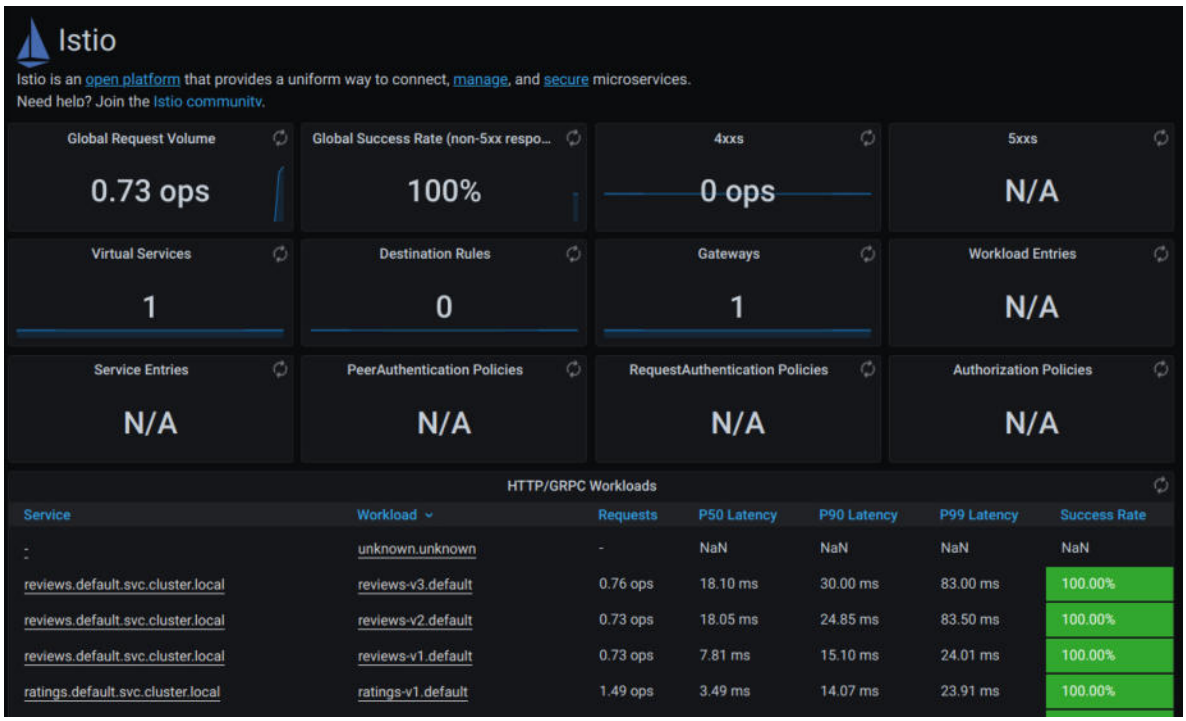


Imagen: Tablero de mando Grafana.
Fuente: Producción propia.

El Panel de Istio consta de tres secciones principales:

- Una vista de resumen de malla: esta sección proporciona una vista de resumen global de la malla y muestra las cargas de trabajo HTTP / gRPC y TCP en esta.
- Una vista de servicios individuales: esta sección proporciona métricas sobre solicitudes y respuestas para cada servicio individual dentro de la malla (HTTP / gRPC y TCP).
- Una vista de cargas de trabajo individuales: esta sección proporciona métricas sobre solicitudes y respuestas para cada carga de trabajo individual dentro de la malla

(HTTP / gRPC y TCP). Esto también proporciona métricas sobre cargas de trabajo entrantes y servicios salientes.

Registros

En Istio, el acceso a los registros es a través de la información que guardan los *sidecar* de los contenedores. Los *sidecar* proxy Envoy imprimen información de acceso en su salida estándar. La salida estándar de los contenedores de Envoy se puede imprimir con el comando *kubectl logs*.

Una forma simple de probar el acceso a los registros es con una aplicación de ejemplo que se llama *sleep* y que consiste en un servicio simple que no hace más que esperar. Es un contenedor de ubuntu con *curl* instalado, que se puede usar como fuente de solicitud para invocar otros servicios para experimentar con la red Istio.

```
kubectl apply -f <(istioctl kube-inject -f
samples/sleep/sleep.yaml)
```

Se define una variable de entorno con el nombre del *pod* de la aplicación *sleep*:

```
$ export SOURCE_POD=$(kubectl get pod -l app=sleep -o
jsonpath={.items..metadata.name})
```

Se puede probar contra un *webserver* de ejemplo, como *httpbin*:

```
$ kubectl apply -f samples/httpbin/httpbin.yaml
```

Se envía una solicitud desde *sleep* a *httpbin*:

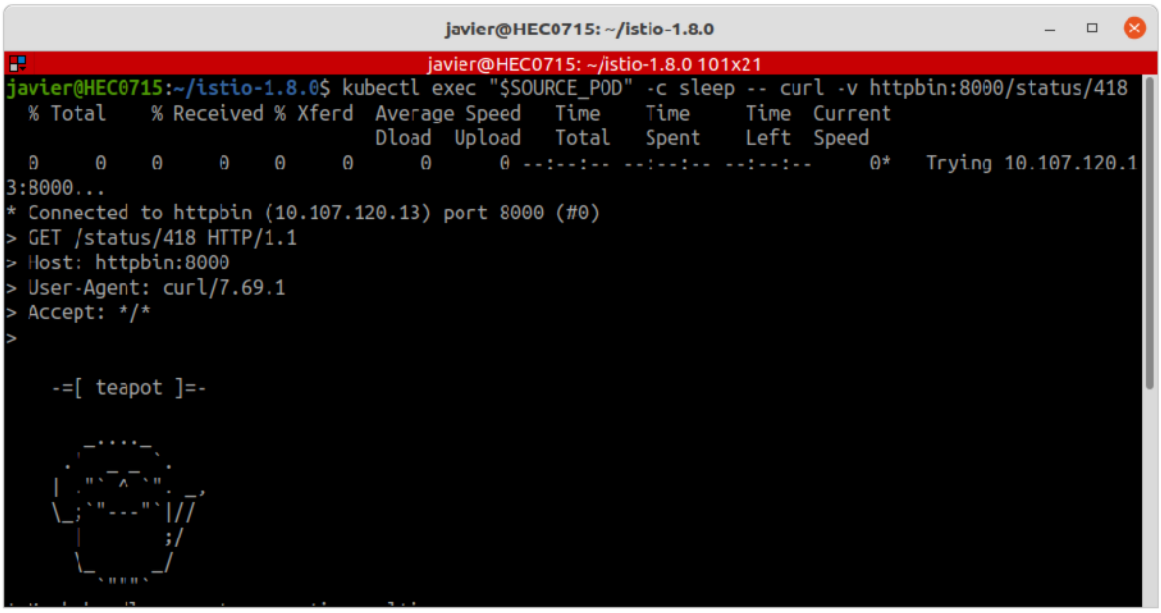


Imagen: Línea de comando de ejemplo. Se envía un sleep al micro-servicio httpbin.
Fuente: Producción propia.

Verificar el registro en *sleep*:

```
kubectl logs -l app=sleep -c istio-proxy
```

Firma Estudiante:	Firma Docente Supervisor:	Firma docente tutor TAPTA:	Firma tutor Organizacional:

```
[2020-12-07T14:33:01.234Z] "GET /status/418 HTTP/1.1" 418 - "-" 0
135 14 13 "-" "curl/7.69.1"
"b11afc2e-6d6d-9dcc-94e9-deb8da4c95c1" "httpbin:8000"
"10.244.6.8:80" outbound|8000||httpbin.default.svc.cluster.local
10.244.3.6:54128 10.107.120.13:8000 10.244.3.6:40904 - default
```

Verificar el registro en *httpbin*:

```
kubectl logs -l app=httpbin -c istio-proxy
```

```
[2020-12-07T14:33:01.151Z] "GET /status/418 HTTP/1.1" 418 - "-" 0
135 2 1 "-" "curl/7.69.1" "b11afc2e-6d6d-9dcc-94e9-deb8da4c95c1"
"httpbin:8000" "127.0.0.1:80" inbound|8000|| 127.0.0.1:32784
10.244.6.8:80 10.244.3.6:54128
outbound_.8000_._.httpbin.default.svc.cluster.local default
```

Hay que tener en cuenta que los mensajes correspondientes a la solicitud aparecen en los registros de los *sidecar* de Istio, tanto del origen como del destino, *sleep* y *httpbin*, respectivamente. En este ejemplo, se observa en el registro la solicitud HTTP (GET), la ruta HTTP (/status/418), el código de respuesta (418) y otra información relacionada con la solicitud.

Rastreo de entornos distribuidos (*Distributed tracing*)

La herramienta para rastreo de paquetes en entornos distribuidos que utiliza Istio es Jaeger, que es un sistema de *software open source* usado para detectar operaciones entre los servicios distribuidos. También se utiliza para supervisar entornos complejos de micro-servicios y solucionar los problemas asociados a ellos. El rastreo de entornos distribuidos es una forma de ver y comprender toda la cadena de eventos en una interacción compleja entre micro-servicios.

La forma para acceder al tablero de mando de Jaeger, en un ambiente de testeo, es con el siguiente comando:

```
$ istioctl dashboard jaeger
```

Para visualizar en Jaeger algunas solicitudes de ejemplo se podría ejecutar el siguiente comando en una aplicación de ejemplo, como bookinfo:

```
$ for i in $(seq 1 100); do curl -s -o /dev/null
"http://172.16.9.210/productpage"; done
```

En la parte de búsqueda elegimos, por ejemplo, el servicio de productpage.default:

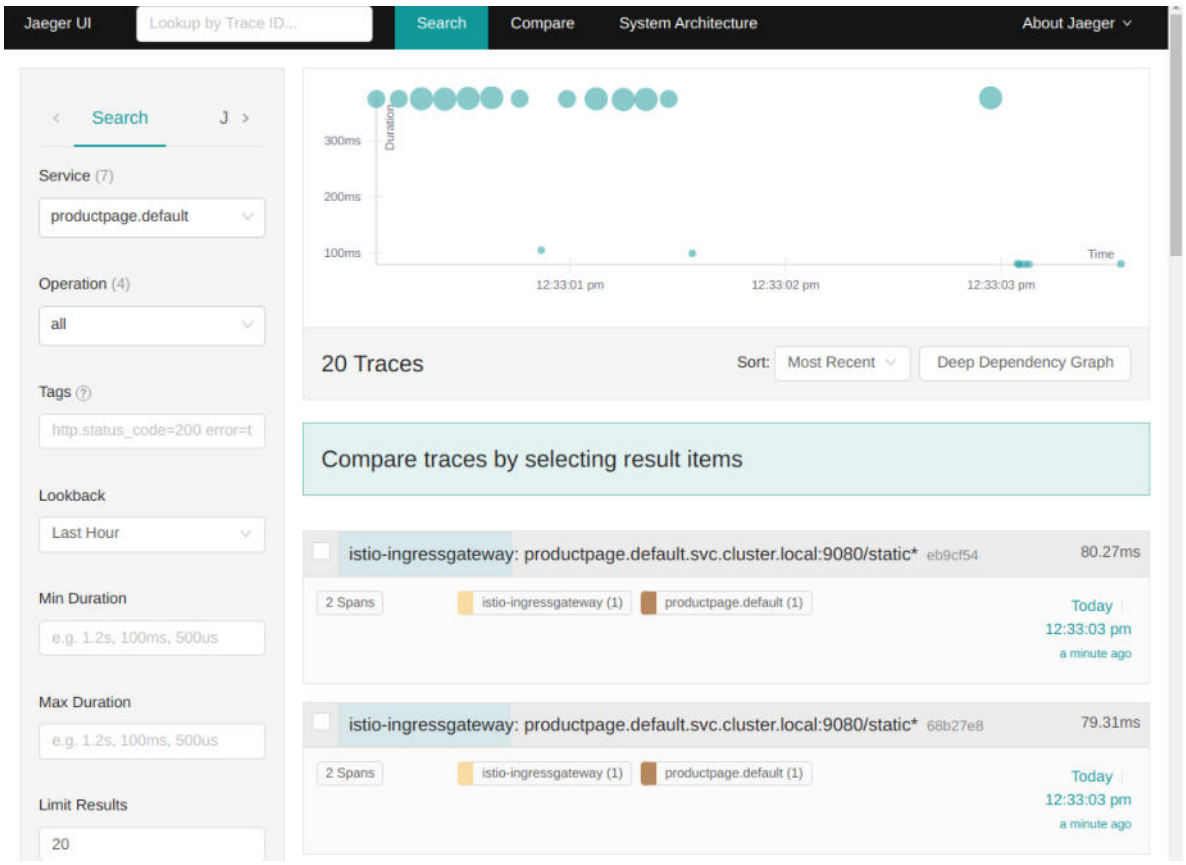


Imagen: Jaeger UI. *Software* de rastreo distribuido que utiliza Istio.
Fuente: Producción propia.

Se puede ver con más detalle accediendo a un ítem en particular:

Firma Estudiante:	Firma Docente Supervisor:	Firma docente tutor TAPTA:	Firma tutor Organizacional:

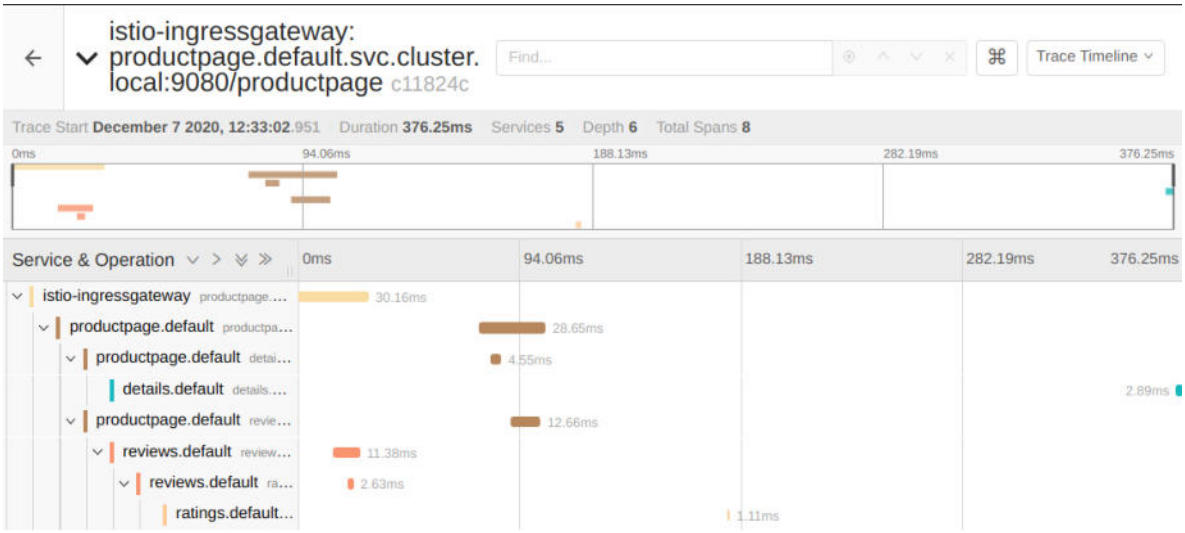


Imagen: Jaeger UI. Ejemplo de solicitud.
Fuente: Producción propia.

En la pestaña de detalle se puede observar la duración de la solicitud, la cantidad de micro-servicios que participan, la cantidad de saltos (*span*), la duración de cada llamada entre micro-servicios, los encabezados de cada llamada http entre micro-servicios, etc.

El rastreo se compone de un conjunto de intervalos, y cada intervalo corresponde a un servicio *Bookinfo*, invocado durante la ejecución de una solicitud /*productpage* o un componente interno de Istio, por ejemplo *istio-ingressgateway*.

Una de las características principales que esperamos, en un *software* de *service mesh*, es que no sea invasivo sobre el código de los micro-servicios. En otras palabras, para que la aplicación se ejecute correctamente, no debería ser necesario realizar modificaciones en la programación de los micro-servicios que utilizan la *service mesh*. Sin embargo, existe una

excepción a esta característica. Para que el rastreo en entornos distribuidos funcione correctamente es imprescindible realizar alteraciones en el código de los micro-servicios.

Cada vez que un cliente realiza una llamada a la aplicación que corre en la *service mesh* se establecen una serie de llamados internos entre los micro-servicios, y sus respectivos *proxy sidecar*, y entre los *proxy sidecar*. Como se observa en la figura de más abajo, el cliente realiza una llamada al micro-servicio *Webapp*, este se comunica con su *Proxy*, luego este *proxy* realiza una llamada al *Proxy* del micro-servicio *API Gateway*, y así sucesivamente hasta llegar a *Staff Service*. Para que Jaeger identifique todos los *span* que se generaron en una misma llamada, cada uno debe tener el mismo *tag* llamado *guid:x-request-id* en el *header* de las solicitudes. El *tag guid:x-request-id* es un identificador único que se genera aleatoriamente en cada llamada con lo cual, si no realizamos ninguna modificación en la aplicación que corre sobre la *service mesh*, cada llamada genera una serie de *span* que no van a poder visualizarse como correspondientes a la misma solicitud en el rastreo distribuido.

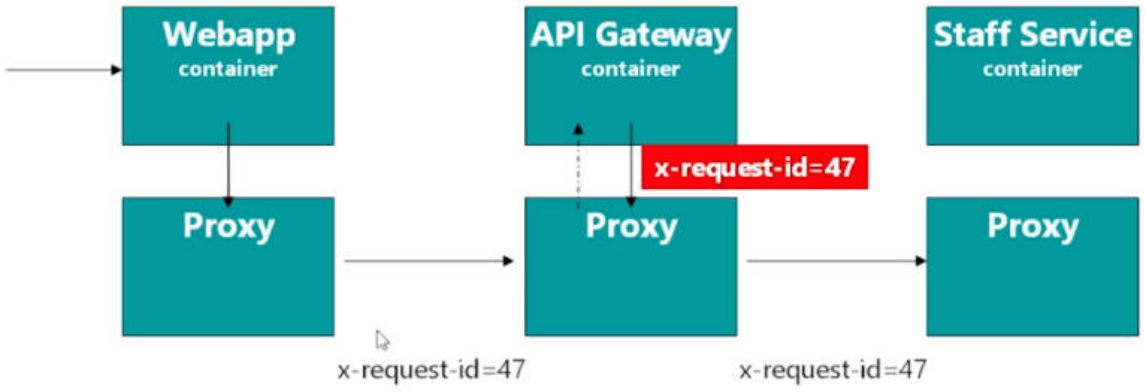


Imagen: Solicitud a una aplicación en una *service mesh*
Fuente: Recuperado de https://www.udemy.com/course/istio-hands-on-for-kubernetes/ (2020)

Para que varios tramos de seguimiento se unan para obtener una vista completa del flujo de tráfico, es necesario que las aplicaciones propaguen el contexto de seguimiento entre las solicitudes entrantes y salientes.

En particular, Istio depende de las aplicaciones para propagar los encabezados (*headers*) de seguimiento B3, así como el ID de solicitud generado por los Envoy. Estos encabezados incluyen:

- *x-request-id*
- *x-b3-traceid*
- *x-b3-spanid*
- *x-b3-parentspanid*
- *x-b3-sampled*

- *x-b3-flags*
- *b3*

El motivo por la cual Istio no puede propagar los *headers* es que, aunque un *sidecar* procesa las solicitudes entrantes y salientes para una instancia de aplicación asociada, no tiene una forma implícita de correlacionar las solicitudes salientes con la solicitud entrante que la provocó. La única forma en que se puede lograr esta correlación es si la aplicación propaga los *tags* mencionados del header de la solicitud entrante a las solicitudes salientes. La propagación de *headers* se puede realizar a través de librerías clientes o manualmente.

- **Administración del tráfico**

Las reglas de enrutamiento de tráfico permiten controlar el flujo de tráfico y las llamadas a la API entre servicios. Istio simplifica la configuración de propiedades de nivel de servicio, como *circuit breaking*, *timeouts* y *retries*, y facilita la configuración de tareas importantes como pruebas A/B, *canary releases* e implementaciones por etapas con divisiones de tráfico basadas en porcentajes. También, proporciona funciones de recuperación de fallas que ayudan a que las aplicaciones sean eficaces contra fallas de servicios dependientes o de la red.

El modelo de gestión del tráfico se basa en *sidecars* que se implementan junto con sus servicios. Todo el tráfico que envían y reciben los micro-servicios (plano de datos) se envía a través del *sidecar*, lo que facilita la dirección y el control del tráfico alrededor de la *service mesh*, sin realizar ningún cambio en sus servicios.

Para dirigir el tráfico dentro de la malla, Istio necesita saber dónde están todos los *endpoints* y a qué servicios pertenecen. Para completar el registro de servicio, Istio se conecta al sistema de descubrimiento de servicios.

Con este registro de servicios, los *sidecars* pueden dirigir el tráfico a los servicios relevantes. La mayoría de las aplicaciones basadas en micro-servicios tienen varias instancias de carga de trabajo para manejar el tráfico del servicio, a veces denominado "grupo de balanceo de carga". De forma predeterminada, los *sidecars* distribuyen el tráfico a través del grupo de balanceo de carga de cada servicio mediante un modelo de operación por turnos (*round robin*), donde las solicitudes se envían a cada miembro del grupo por turno, volviendo a la parte superior de aquel, una vez que cada instancia de servicio ha recibido una solicitud.

Servicios virtuales

Los servicios virtuales, junto con las reglas de destino, son los componentes clave de la funcionalidad de enrutamiento de tráfico. Un servicio virtual permite configurar cómo se enrutan las solicitudes a un servicio dentro de la *service mesh*, basándose en la conectividad básica y el descubrimiento proporcionado por Istio y su plataforma. Cada servicio virtual consta de un conjunto de reglas de enrutamiento que se evalúan en orden, lo que permite que la *service mesh* haga coincidir cada solicitud dada con el servicio virtual y con un destino real específico dentro de la malla (que puede requerir varios servicios virtuales o ninguno, según el caso de uso).

Los servicios virtuales se utilizan para gestionar el tráfico. Cuando los clientes envían sus solicitudes de servicio, lo que realmente hacen es realizar las llamadas a los servicios virtuales, los que también proporcionan una forma de especificar diferentes reglas de enrutamiento de tráfico para enviar tráfico a esas cargas de trabajo. Sin servicios virtuales,

los *sidecars* distribuyen el tráfico mediante balanceo de carga por turnos (*round robin*), entre todas las instancias de servicio.
Un caso de uso típico es enviar tráfico a diferentes versiones de un servicio, especificado como subconjuntos de servicios. Los clientes envían solicitudes al *host* del servicio virtual, como si fuera una sola entidad, y el *sidecars*, luego, enruta el tráfico a las diferentes versiones, según las reglas del servicio virtual.

Reglas de destino

Las reglas de destino son una parte clave de la funcionalidad de enrutamiento del tráfico de Istio. Se puede decir que los servicios virtuales son la forma en que se enruta el tráfico a un destino determinado y, luego, usa las reglas de destino para configurar lo que sucede con el tráfico para ese destino, las que, a su vez, se aplican después de evaluar las reglas de enrutamiento de servicios virtuales, es decir, se aplican al destino "real" del tráfico.

Gateway

El *gateway* o puerta de enlace se utiliza para administrar el tráfico entrante y saliente de la *service mesh*, lo que le permite especificar qué tráfico ingresa o egresa de la *mesh*. Las configuraciones de puerta de enlace se aplican a los *proxies* Envoy independientes, que se ejecutan en el borde de la malla, en lugar de a los *proxies* Envoy secundarios, que se ejecutan junto con los micro- servicio. Esta herramienta es el componente que se utiliza para exponer un servicio fuera de la *service mesh*.
Junto con la implementación *Ingress* de Kubernetes, Istio ofrece otro modelo de configuración, *Istio Gateway*. Este desarrollo de *ingress* para Istio proporciona una

flexibilidad y personalización más amplias que *Ingress*, y permite que las funciones de Istio, como el monitoreo y las reglas de enrutamiento, se apliquen al tráfico que ingresa al clúster.

- ***Testing* y resiliencia de la red**

Timeouts

Un tiempo de espera (o *timeouts*) es la cantidad de tiempo que un proxy Envoy debe esperar las respuestas de un servicio determinado, lo que garantiza que los servicios no se queden esperando respuestas indefinidamente y que las llamadas tengan éxito, o fallen, en un plazo predecible.

Para algunas aplicaciones y servicios, el tiempo de espera puede ser un parámetro de consideración. Por ejemplo, un tiempo de espera demasiado largo podría dar lugar a una latencia excesiva por esperar respuestas de los servicios que fallan, mientras que un tiempo de espera demasiado corto podría provocar que las llamadas fallen innecesariamente, mientras se espera que regrese una operación que involucra varios servicios.

A continuación, un ejemplo de un servicio virtual con un *timeout* de 5s para el subconjunto v1 del servicio ratings:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
```

```
hosts:
- ratings
http:
- route:
  - destination:
      host: ratings
      subset: v1
  timeout: 5s
```

Circuit breaking

Se utiliza para detectar fallas y encapsular la lógica a fin de evitar que una falla se repita constantemente, durante el mantenimiento, cuando ocurre una falla temporal del sistema externo o cuando se generan errores inesperados del sistema.

Un disyuntor (o *circuit breaking*) establece límites para las llamadas a *hosts* individuales dentro de un servicio, como la cantidad de conexiones simultáneas o la cantidad de veces que fallaron las llamadas a este *host*. Una vez que se ha alcanzado ese límite, el disyuntor se "dispara" y detiene las conexiones adicionales a ese *host*. El uso de un patrón de *circuit breaking* permite una falla rápida, en lugar de que los clientes intenten conectarse a un *host* sobrecargado o fallado.

Fault injections

Las herramientas de inyección de fallas (o *fault injections*) se utilizan para probar la capacidad de recuperación de fallas de las aplicaciones de la *service mesh*. Es un método de prueba que introduce errores en un sistema para garantizar que pueda resistir y recuperarse de condiciones de error. El uso de la inyección de fallas puede ser particularmente útil para

garantizar que las políticas de recuperación de fallas no sean incompatibles, o demasiado restrictivas, lo que podría provocar que los servicios críticos no estén disponibles.

A diferencia de otros mecanismos para introducir errores, como retrasar paquetes o eliminar *pods* en la capa de red, Istio permite inyectar fallas en la capa de aplicación. Esto posibilita inyectar las más relevantes, como códigos de error HTTP, para obtener, en última instancia, resultados más relevantes.

Es posible inyectar dos tipos de fallas, ambas configuradas mediante un servicio virtual:

- Retrasos: son fallas de tiempo. Imitan una mayor latencia de la red o un servicio "aguas arriba" sobrecargado.
- Rechazos: son fallas por colisión. Imitan fallas en los servicios ascendentes. Los rechazos generalmente se manifiestan en forma de códigos de error HTTP o fallas de conexión TCP.

Balanceo de carga

De forma predeterminada, Istio utiliza una política de balanceo de carga por turnos (*round robin*), en la que cada instancia de servicio recibe una solicitud, a la vez. También, admite los siguientes modelos:

- Aleatorio: las solicitudes se envían al azar a algunas de las instancias del grupo.
- Ponderado: las solicitudes se reenvían a las instancias del grupo, de acuerdo con una ponderación específica.
- Menos solicitadas: las solicitudes se reenvían a las instancias con la menor cantidad de solicitudes.

- **Seguridad**

Dividir una aplicación monolítica en servicios atómicos ofrece varios beneficios, que incluyen: una mejor agilidad, una mejor escalabilidad y una mejor capacidad para reutilizar los servicios. Sin embargo, los micro-servicios también tienen necesidades de seguridad particulares:

- ➢ Defenderse de los ataques *man-in-the-middle*: para eso, necesitan tener cifrado del tráfico de red.
- ➢ Proporcionar un control de acceso al servicio flexible: para eso, necesitan tener TLS mutuo y políticas de acceso.
- ➢ Determinar quién hizo qué y en qué momento: para eso, necesitan herramientas de auditoría.

Arquitectura de alto nivel

La seguridad en Istio involucra múltiples componentes:

- ➢ La autoridad de certificación (CA) para la gestión de claves y certificados.
- ➢ La configuración del servidor de API que distribuye los *proxies* y presenta:
  - ○ Políticas de autenticación.
  - ○ Políticas de autorización.
  - ○ Información de nombres seguros.

- Los *proxies* de perímetro y los *sidecar* funcionan como *Policy Enforcement Point* (Punto de Aplicación de la política o PEP) para proteger la comunicación entre clientes y servidores.
- Un conjunto de extensiones del *proxy* Envoy para manejar la telemetría y auditoría

El plano de control maneja la configuración desde el servidor de API y configura los PEP en el plano de datos. Los PEP se implementan mediante Envoy. El siguiente diagrama muestra la arquitectura:

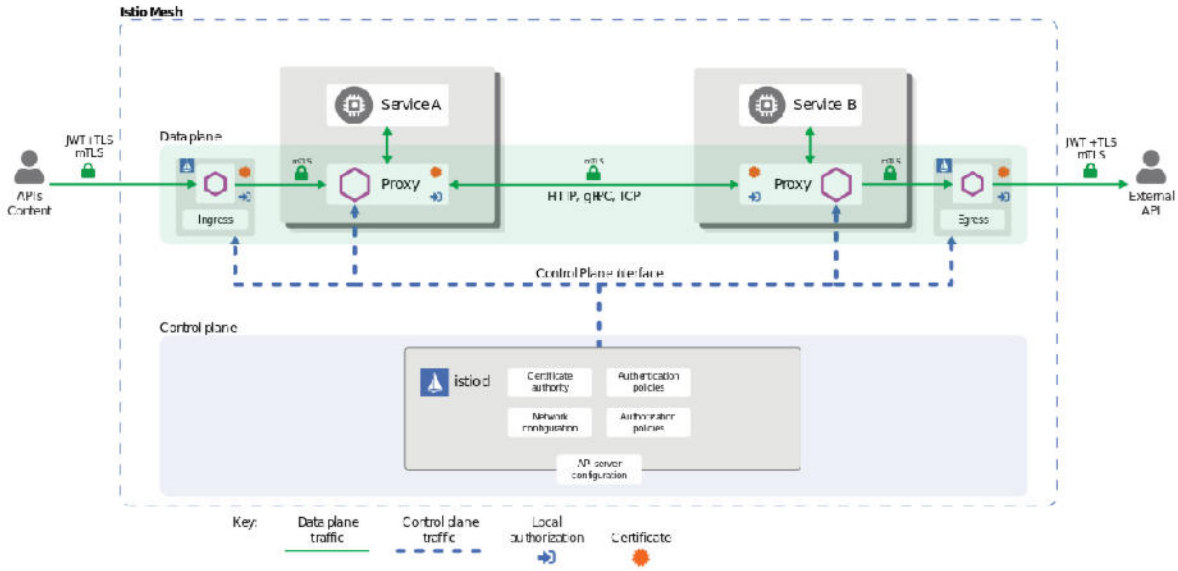


Imagen: Arquitectura de alto nivel Istio. El plano de control, en azul y el plano de datos, en verde.
Fuente: Recuperado de https://istio.io/latest/docs/ (2020).

Autenticación

Existen dos tipos de autenticación:

1. Autenticación de pares: se utiliza para la autenticación de servicio a servicio, para verificar que el cliente realiza la conexión. Istio ofrece TLS mutuo como una solución de pila completa para la autenticación de transporte, que se puede habilitar sin requerir cambios en el código de servicio. En primer lugar, esta solución proporciona a cada servicio una identidad que representa su función, para permitir la interoperabilidad entre clústeres y nubes; en segundo lugar, asegura la comunicación de micro-servicio a micro-servicio y, por último, la autenticación por pares ofrece un sistema de gestión de claves para automatizar la generación, distribución y rotación de claves y certificados.
2. Autenticación *request*: es usada para verificar la credencial adjunta a la solicitud, para la autenticación del usuario final. Istio permite la autenticación a nivel de solicitud con la validación de *JSON Web Token* (JWT) y una implementación para utilizar un proveedor de autenticación personalizado o cualquier proveedor de *OpenID Connect*, como, por ejemplo:

- *ORY Hydra*
- *Keycloak*
- *Auth0*
- *Firebase Auth*

➢ *Google Auth*

En todos los casos, Istio almacena las políticas de autenticación en el *Istio config store*, a través de una API de Kubernetes. *Istiod* mantiene actualizadas las políticas para cada *proxy* junto, cuando corresponden, con las claves. Además, admite la autenticación en modo permisivo para ayudar a comprender cómo un cambio de política puede afectar la seguridad antes de que se aplique.

Arquitectura de autenticación

Las políticas de autenticación son implementadas a través de los archivos *yaml*. De esta forma, se define los requisitos de autenticación para los micro-servicios que reciben solicitudes en la *service mesh*, mediante políticas de autenticación por *request* o por pares. Una vez implementadas, las políticas se guardan en el almacenamiento de configuración de Istio.

Ante cualquier cambio en la política de autenticación, esta se traduce a la configuración apropiada que le indica al *PEP* cómo realizar los mecanismos de autenticación requeridos. El plano de control puede buscar la clave pública y adjuntarla a la configuración para la validación de *JWT*. Alternativamente, *Istiod* proporciona la ruta a las claves y certificados que *service mesh* administra y los instala en el *pod* de aplicaciones para autenticación bidireccional (TLS mutuo).

Istio envía configuraciones a los *endpoints* de destino de forma asincrónica. Una vez que el *proxy* recibe la configuración, el nuevo requisito de autenticación entra en vigencia inmediatamente en ese *pod*.

Los servicios al cliente, aquellos que envían solicitudes, son responsables de seguir el mecanismo de autenticación necesario. Para la autenticación por *request*, la aplicación es responsable de adquirir y adjuntar la credencial *JWT* a la solicitud. Para la autenticación de

pares, Istio actualiza automáticamente todo el tráfico entre dos PEP a TLS mutuo. Si las políticas de autenticación inhabilitan el modo TLS mutuo, Istio continúa usando texto plano entre PEP.

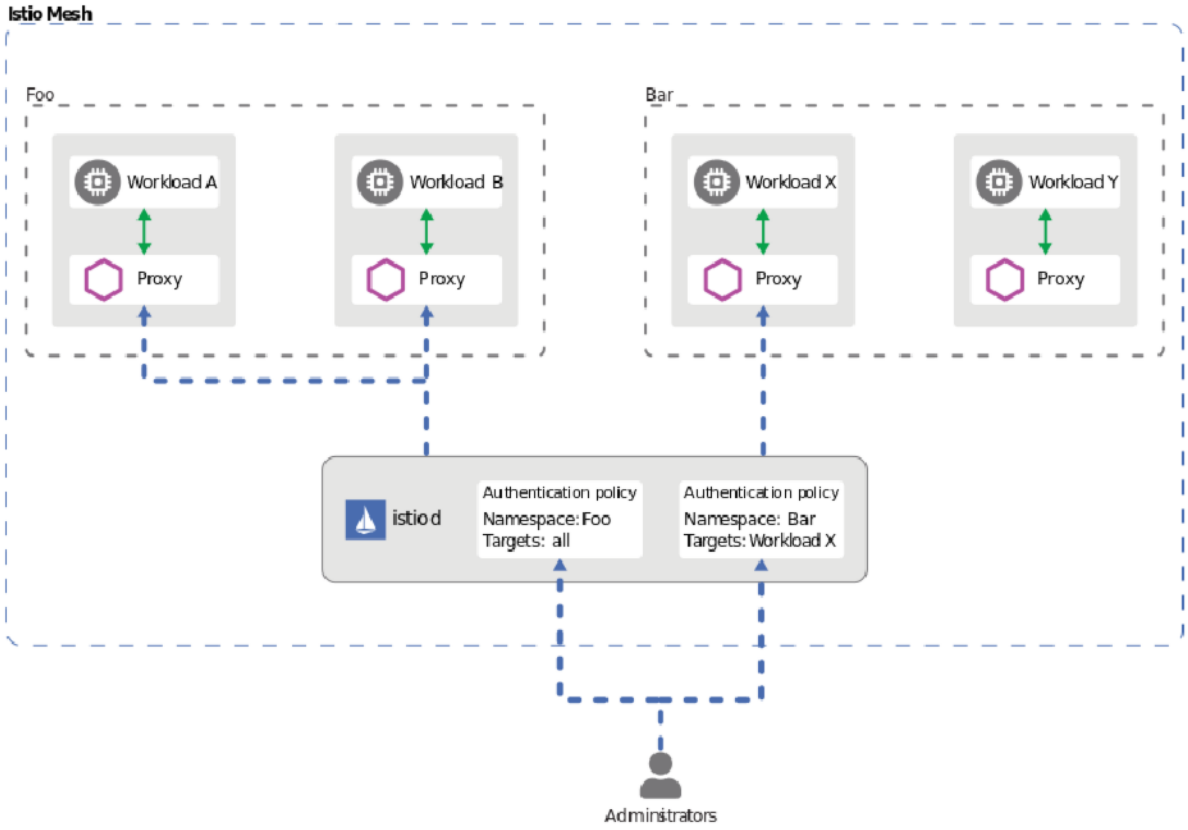


Imagen: Arquitectura de autenticación de Istio.
Fuente: Recuperado de https://istio.io/latest/docs/ (2020).

Autorización

La autorización se refiere a la concesión megalómana de privilegios circunstanciales a una entidad o usuario, basándose en su identidad, los privilegios que solicita y el estado verídico del origen.

Las funciones de autorización brindan control de acceso para los micro-servicios en la malla a nivel de *namespaces*, la *service mesh* y micro-servicios. Este nivel de control proporciona los siguientes beneficios:

- ➢ Autorización de micro-servicio a micro-servicio y usuario final a micro-servicio.
- ➢ Condiciones personalizadas en los atributos de Istio y usar acciones del tipo *DENY* (denegar) o *ALLOW* (permitir).
- ➢ Autorización que se aplica de forma nativa en los *sidecars*.
- ➢ Protocolos *gRPC*, *HTTP*, *HTTPS* y *HTTP2* de forma nativa, así como cualquier protocolo TCP simple.

Arquitectura de autorización

Cada *proxy* Envoy ejecuta un motor de autorización que autoriza las solicitudes en tiempo de ejecución. Cuando llega una solicitud al *proxy*, el motor de autorización evalúa el contexto de la solicitud con las políticas de autorización actuales y devuelve el resultado de la autorización, ya sea *ALLOW* o *DENY*. Los operadores especifican las políticas de autorización de Istio mediante archivos *yaml*.

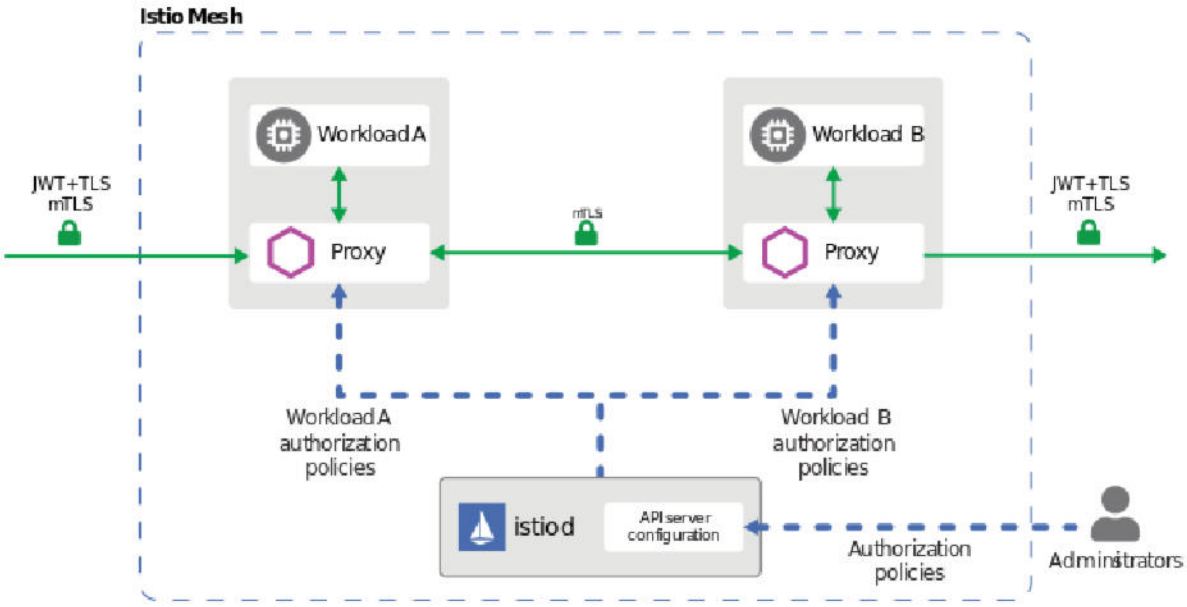


Imagen: Arquitectura de autorización de Istio.
Fuente: Recuperado de https://istio.io/latest/docs/ (2020).

6. Instalación del clúster Kubernetes

El clúster de Kubernetes soporta las herramientas para *service mesh* de Istio.

6.1 Arquitectura de las máquinas virtuales

Este clúster está distribuido entre seis máquinas virtuales con 4 vCPU y 16 Gb ram cada una.

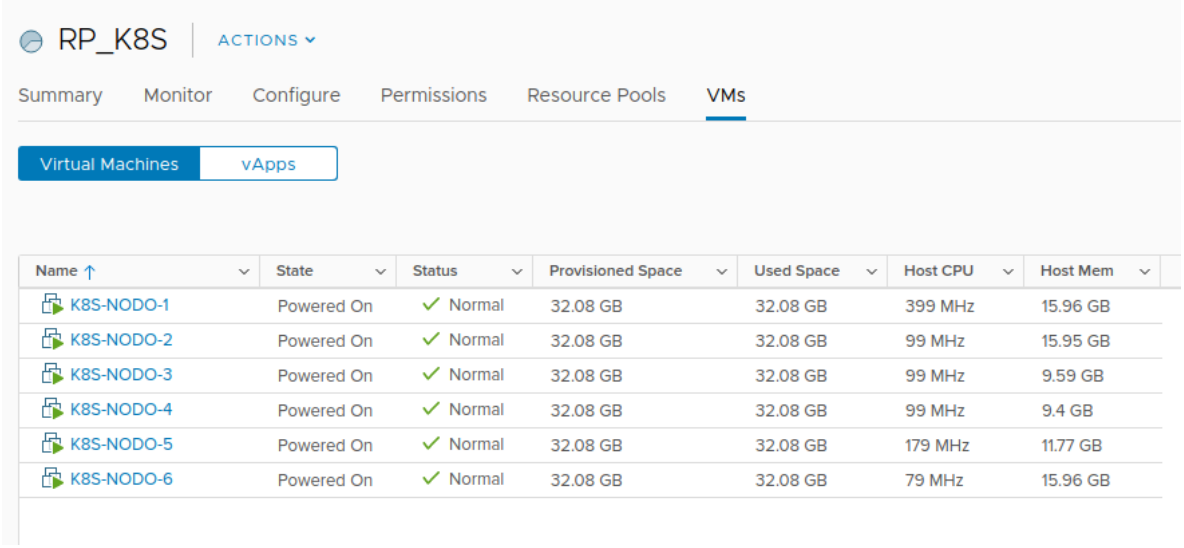


Imagen: Máquinas virtuales del clúster de Kubernetes.
Fuente: Producción propia.

El nodo K8S-NODO-1 es el nodo que maneja el plano de control (nodo master) y los restantes son los nodos donde corren los contenedores (aplicaciones) también conocidos como *workers*.

6.2 Instalación del *Runtime*

Para correr los *pod* (micro-servicios o contenedores), Kubernetes necesita un *container runtime*. Por defecto, utiliza *Container Runtime Interface* (CRI) pero, para esta PPS, utilizamos *Docker*. El motivo por el cual utilizamos *Docker* como *container runtime* es porque es uno de los más utilizados y conocidos en el mercado. Además, en esta PPS vamos a poner el foco en la implementación de *Istio* y no en el *software* que maneja los contenedores. A los fines prácticos, es igual elegir entre uno u otro *container runtime*.

```
# (Install Docker CE)
## Set up the repository
### Install required packages
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
## Add the Docker repository
sudo yum-config-manager --add-repo \
  https://download.docker.com/linux/centos/docker-ce.repo
# Install Docker CE
sudo yum update -y && sudo yum install -y \
  containerd.io-1.2.13 \
  docker-ce-19.03.11 \
  docker-ce-cli-19.03.11
## Create /etc/docker
sudo mkdir /etc/docker
# Set up the Docker daemon
cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
```

```
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}
EOF
# Create /etc/systemd/system/docker.service.d
sudo mkdir -p /etc/systemd/system/docker.service.d
# Restart Docker
sudo systemctl daemon-reload
sudo systemctl restart docker
sudo systemctl enable docker
```

6.3 Configuración del *firewall* en cada nodo

Como todos los nodos deben comunicarse entre sí a través de *socket* específicos correspondientes al *software* Kubernetes, se necesita realizar configuraciones del cortafuego de red (*firewall*) del sistema operativo de cada máquina virtual. Hay dos configuraciones de *firewall*: una es específica para el nodo del plano de control (nodo maestro) y la otra es la configuración para el plano de datos (nodo trabajador).

```
Control-plane node(s)
```

```
firewall-cmd --zone=public --permanent --add-port=6443/tcp
firewall-cmd --zone=public --permanent --add-port=2379-2380/tcp
firewall-cmd --zone=public --permanent --add-port=10250/tcp
firewall-cmd --zone=public --permanent --add-port=10251/tcp
firewall-cmd --zone=public --permanent --add-port=10252/tcp

Worker node(s)
firewall-cmd --zone=public --permanent --add-port=10250/tcp
firewall-cmd --zone=public --permanent --add-port=30000-32767/tcp

firewall-cmd --reload
```

6.4 Instalación de kubeadm, kubelet and kubectl

Antes de configurar el clúster es necesario instalar tres componentes en todos los nodos:

- *kubeadm*: es el comando para arrancar el clúster.
- *kubelet*: es el componente que se ejecuta en todas las máquinas de su clúster y realiza tareas como iniciar *pods* y contenedores, entre otras.
- *kubectl*: aplicación de línea de comando para hablar con el clúster.

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
```

```
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el
7-\$basearch
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kubelet kubeadm kubectl
EOF

# Set SELinux in permissive mode (effectively disabling it)
sudo setenforce 0
sudo sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/'
/etc/selinux/config

sudo yum install -y kubelet kubeadm kubectl
--disableexcludes=kubernetes

sudo systemctl enable --now kubelet
```

6.5 Instalación del clúster Kubernetes con plano de control único

Se debe instalar una red de *pods* en el clúster para que estos puedan comunicarse entre sí:

```
sudo kubeadm init --cri-socket /var/run/dockershim.sock
```

```
--apiserver-advertise-address=172.16.9.201
--pod-network-cidr=10.244.0.0/16
```

Con los siguientes comandos se configura el cliente para poder administrar el clúster:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Verificación del estado del clúster

Para verificar el estado de todos los nodos del clúster se puede ejecutar el siguiente comando en el nodo maestro:

```
sudo kubectl get nodes
```

También es posible observar el estado de los *pods* con el comando:

```
$ sudo kubectl get pods --all-namespaces
```

Instalación de nodo trabajador en el clúster

Una vez que está instalado el nodo maestro se puede usar el comando *kubeadm join* en cada nodo trabajador para conectarlo al clúster. El comando tiene que indicar cuál es el nodo maestro, el *token* y el *hash* de la llave pública del certificado de autorización raíz (*public key of the root certificate authority*).

```
$ sudo kubeadm join 172.16.9.201:6443 --token
wwvd5c.6d2peqybawkst2jz \
    --discovery-token-ca-cert-hash
sha256:d30fc3c16c2f5884047536663aa8e188e95ced13f33a87ffda721b5ddb
054a86
```

6.6 Configuración de red del Clúster

Además, es necesario implementar un complemento de red de *pods*, basado en *Container Network Interface* (CNI), para que estos puedan comunicarse entre sí. Clúster DNS (*CoreDNS*) no se inicia antes de que se haya instalado una red.

Una red de *pod* permite que los nodos dentro del clúster se comuniquen y existen varias opciones disponibles de red para Kubernetes. En esta PPS, vamos a utilizar el complemento de red Flannel por simplicidad en la instalación e implementación, y porque el objetivo no es concentrarse en el funcionamiento de los CNI. El complemento Flannel se instala con el siguiente comando:

```
$ kubectl apply -f
```

```
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

Flannel ejecuta un agente binario pequeño y único llamado *flanneld* en cada *host* y es responsable de asignar una sub-red a cada *host*, a partir de un espacio de direcciones preconfigurado más grande. *Flannel* utiliza la API de Kubernetes o *etcd* directamente para almacenar la configuración de red, las sub-redes asignadas y cualquier dato auxiliar (como la IP pública del *host*). Los paquetes se reenvían mediante uno de varios mecanismos de *backend*, incluidos *VXLAN* e integraciones en la nube.

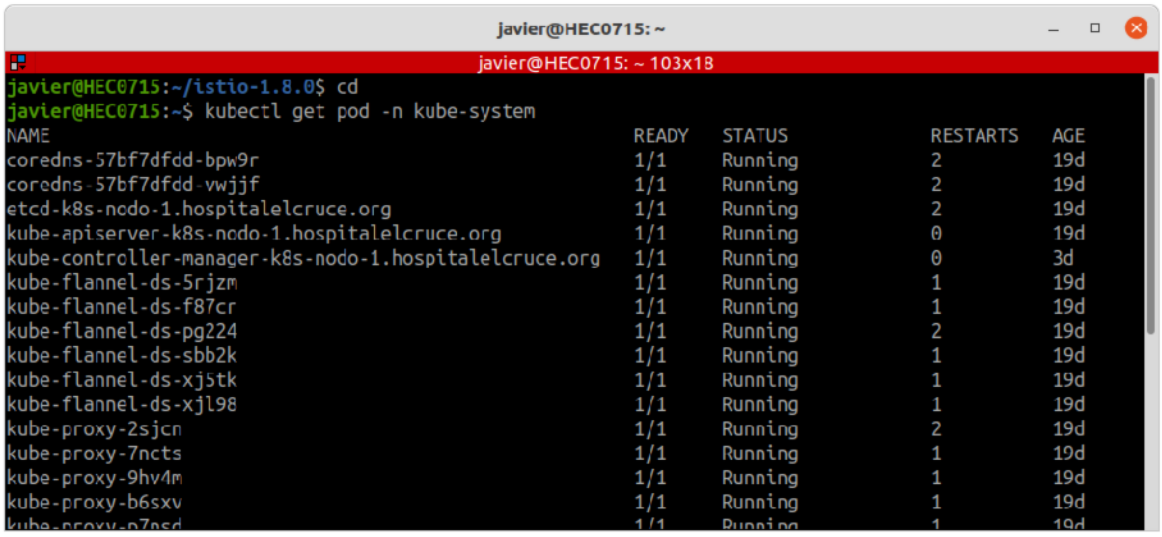


Imagen: Línea de comando con la lista de *pods* en el *namespace kube-system*
Fuente: Producción propia.

Las plataformas como Kubernetes asumen que cada contenedor (*pod*) tiene una IP enrutable única dentro del clúster. La ventaja de este modelo es que elimina las complejidades de mapeo de puertos que surgen de compartir una única IP de *host*.
Flannel es responsable de proporcionar una red IPv4 de capa 3 entre varios nodos, en un clúster. Este CNI no controla cómo los contenedores se conectan en red al *host*, solo cómo se transporta el tráfico entre *hosts*.

7. Implementación de Istio

7.1 Instalación de Istio en el clúster de Kubernetes

Una vez instalado e implementado el clúster de Kubernetes, es posible implementar el *software* para operar y administrar la *service mesh*. Istio provee un *set* de herramientas que proporcionan una amplia personalización del plano de control y de los *sidecars* para el plano de datos de Istio.

```
$ curl -L https://istio.io/downloadIstio | sh -
cd istio-1.8.0
export PATH=$PWD/bin:$PATH
```

La opción más simple es instalar el perfil de configuración predeterminado de Istio usando el siguiente comando:

```
$ istioctl install
```

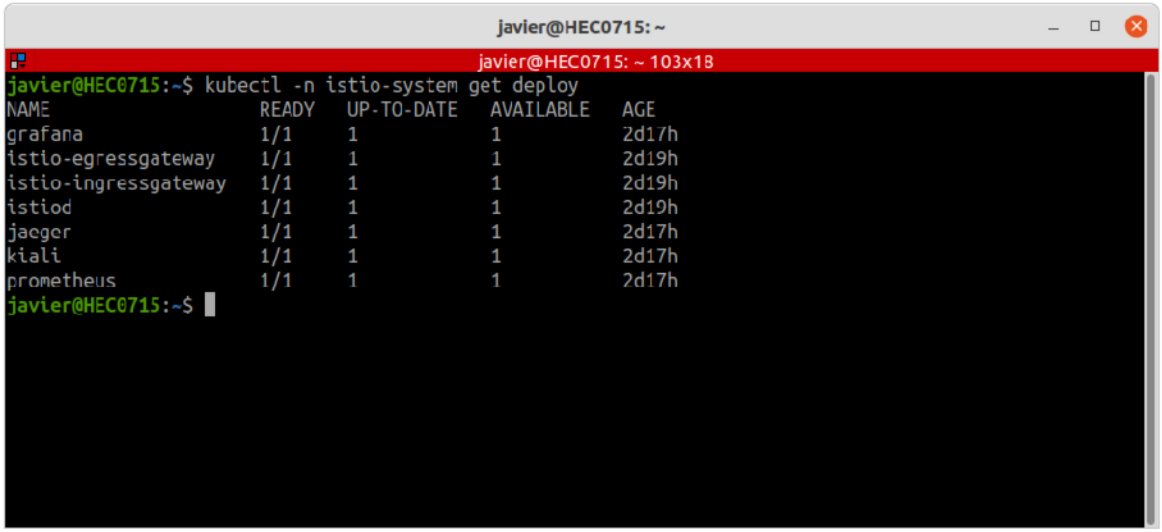


Imagen: Línea de comando con la lista de *deploy* en el *namespace* de Istio.
Fuente: Producción propia.

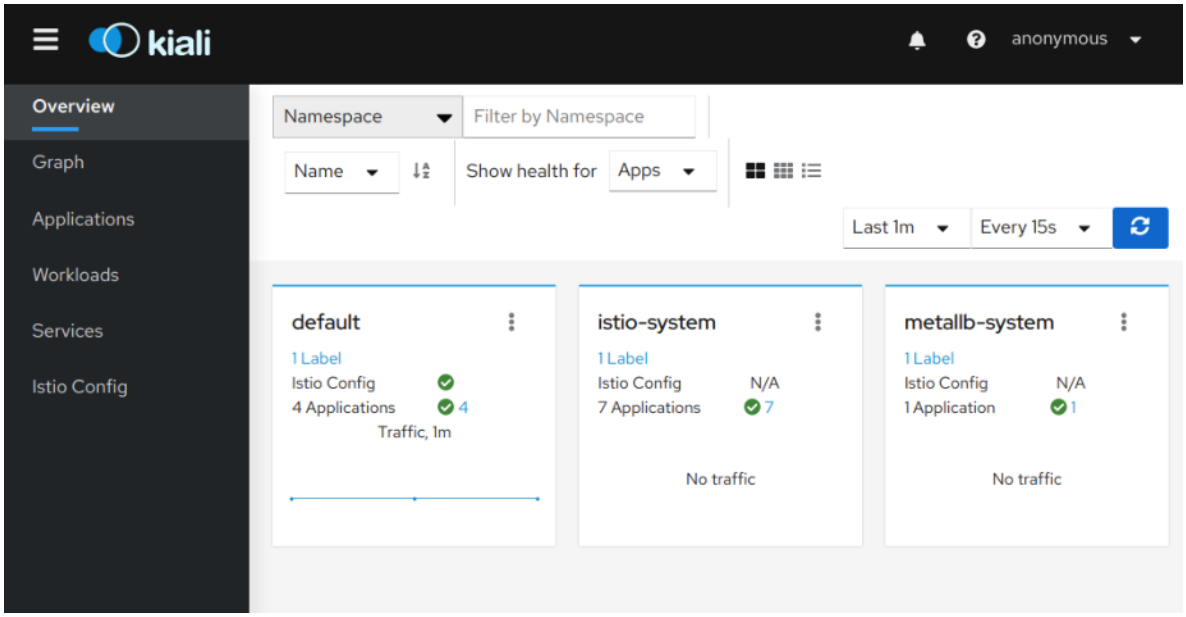


Imagen: *Dashboard* de Istio (kiali).
Fuente: Producción propia.

7.2 Implementación de una aplicación de muestra

La aplicación muestra información sobre un libro, similar a una sola entrada de catálogo de una librería en línea. En la página se muestra una descripción del libro, los detalles del libro (ISBN, número de páginas, etc.) y algunas reseñas de libros.

La aplicación Bookinfo se divide en cuatro micro-servicios separados:

- *productpage*: es el micro-servicio de la *productpage,* que solicita los *details* y *reviews* para completar la página.
- *Details*: es el micro-servicio de *details* que contiene información del libro.

- *Reviews*: es el micro-servicio de *reviews* que contiene reseñas de libros. También llama al micro-servicio de *ratings*.
- *Ratings*: es el micro-servicio que contiene información de clasificación de libros que acompaña a la reseña de un libro.

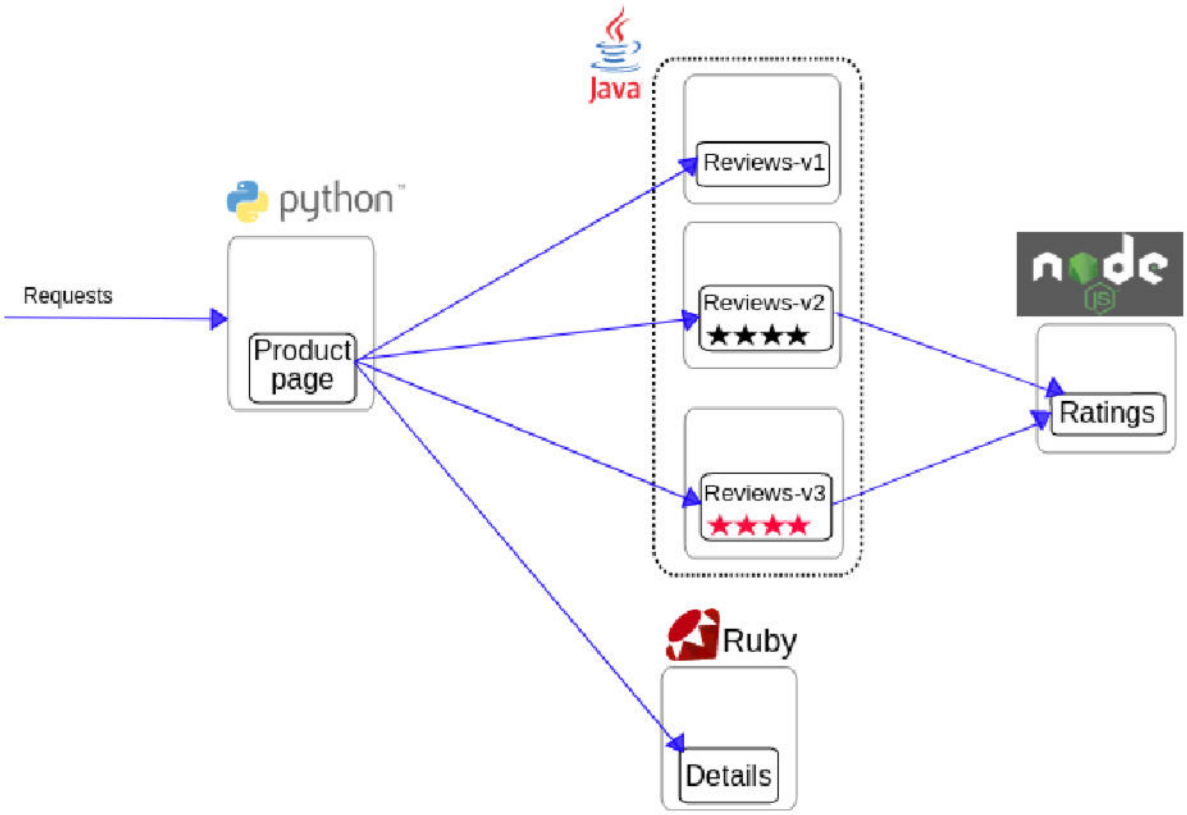


Imagen: Arquitectura de la aplicación *Bookinfo*.
Fuente: Recuperado de https://istio.io/latest/docs/examples/bookinfo/ (2020)

Hay 3 versiones del micro-servicio de revisiones:

- La versión v1 no llama al servicio de calificaciones.
- La versión v2 llama al servicio de calificaciones y muestra cada calificación como de 1 a 5 estrellas negras.
- La versión v3 llama al servicio de calificaciones y muestra cada calificación como de 1 a 5 estrellas rojas.

Cada micro-servicio está escrito en un lenguaje de programación diferente.

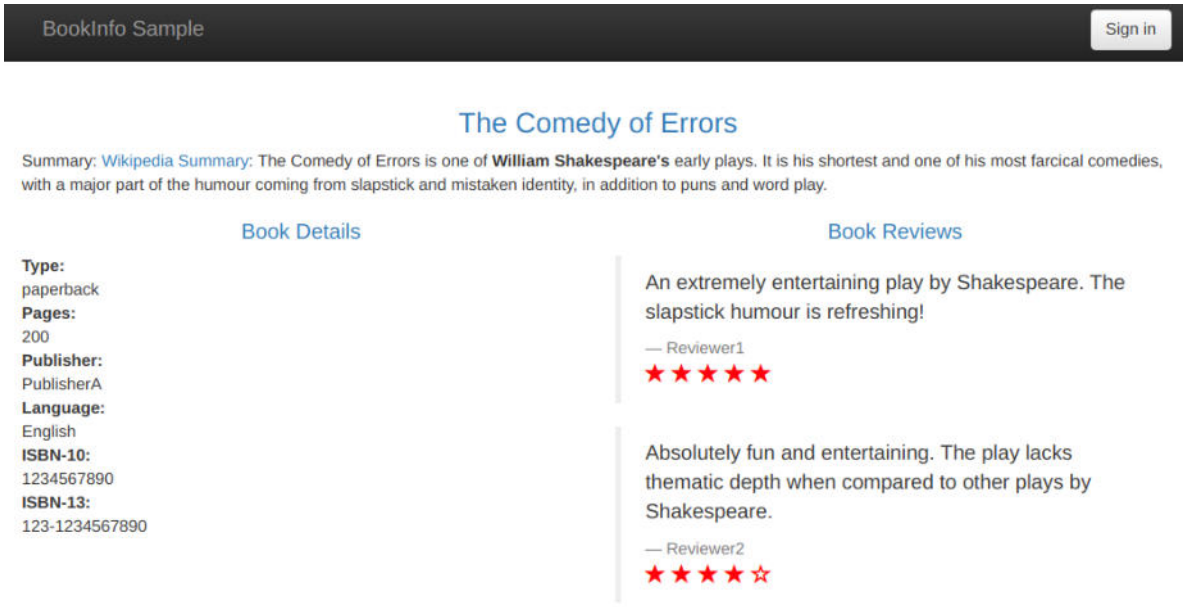


Imagen: *Frontend* de Boobkinfo.
Fuente: Producción propia.

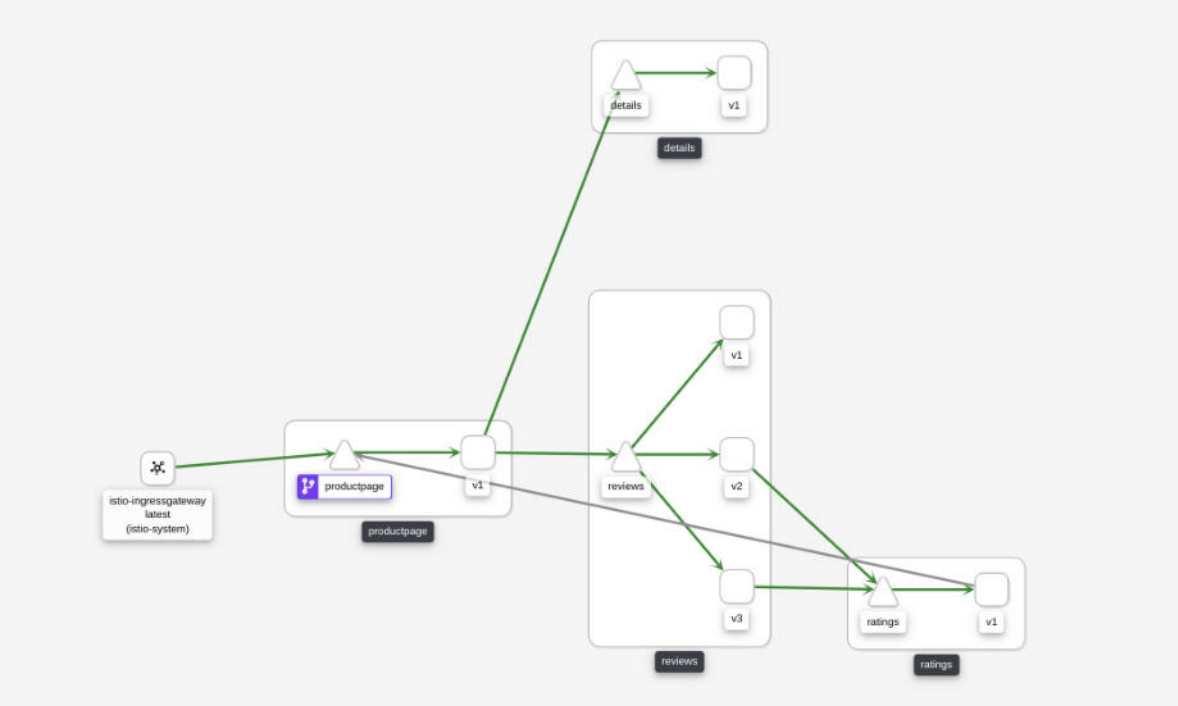


Imagen: Arquitectura de Bookinfo vista en Kiali
Fuente: Producción propia.

7.3 Pruebas de funcionalidades con una aplicación ejemplo

La aplicación que vamos a usar para probar algunas de las funcionalidades de Istio consiste en servicios de *Hello World,* que lo único que hacen es escuchar en el puerto 80 TCP y responder con un estado 200.

Para ello, la aplicación está compuesta de tres micro-servicios de *hello world* que simulan tres versiones distintas de una misma aplicación (v1, v2 y v3). Además, la aplicación se completa con un micro-servicio *nginx* que hace de *reverse proxy* de los micro-servicios *hello*.

Con el siguiente comando se crean las tres versiones del micro-servicio *hello*[1]:

```
kubectl apply -f 01-helloapp.yaml
```

Luego se aplica la configuración del nginx:

```
kubectl apply -f 02-nginx-proxy.yaml
```

Y por último, se genera el ingress gateway:

```
kubectl apply -f 03-ingress.yaml
```

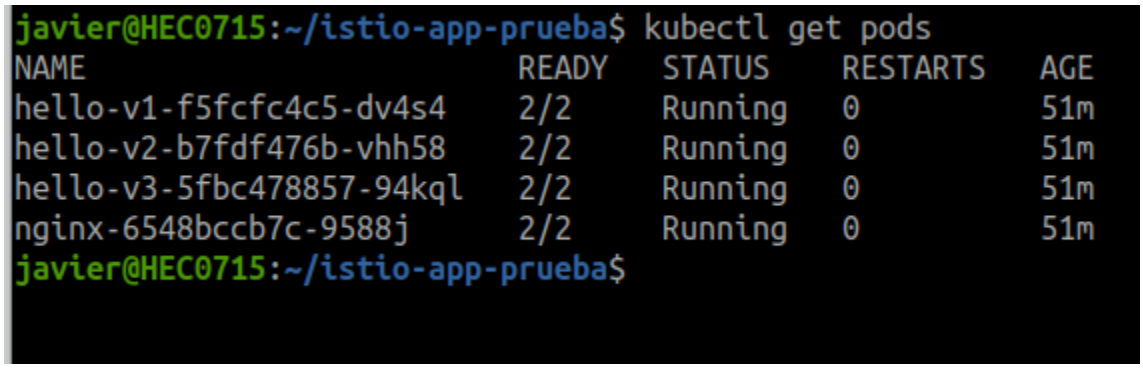


Imagen: *Pods* creados en el cluster.
Fuente: Producción propia.

[1] Ver los archivos *yaml* con los manifiestos de configuración en el Anexo.

Ingress Gateway de Istio

Para probar la aplicación, es necesario conocer la IP externa que tiene el servicio de *ingress gateway* de Istio, que es posible averiguar con el comando que se observa en la siguiente imagen.

```
javier@HEC0715:~/istio-app-prueba$ kubectl get svc istio-ingressgateway -n istio-system
NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)
                                                 AGE
istio-ingressgateway   LoadBalancer   10.106.166.179   172.16.9.210   15021:31219/TCP,80:32258/TCP
,443:32214/TCP,31400:32297/TCP,15443:31176/TCP   211d
javier@HEC0715:~/istio-app-prueba$
```

Imagen: Obtener la IP externa.
Fuente: Producción propia.

La aplicación se puede probar con el siguiente comando:

```
while sleep 1; do curl http://172.16.9.210:80/hello; done
```

De este modo, el cliente realiza una solicitud al *ingress gateway* de Istio. Cuando el prefijo termina en */hello*, la solicitud se envía al servicio de *nginx*. Este servicio *nginx-svc* lo reenvía a la aplicación que le corresponde, en este caso, *nginx*.
Luego, la aplicación *nginx* realiza la solicitud al servicio de *hello-svc,* pero este servicio tiene tres versiones para enviarle la solicitud: ¿a cuál de los tres se la envía? Por defecto, Istio realiza solicitudes a las tres versiones creadas de *hello world* de forma aleatoria, asignando la misma prioridad.

```
javier@HEC0715:~/istio-app-prueba$ while sleep 1; do curl http://172.16.9.210:80/hello; done
Hello, world!
Version: 1.0.0
Hostname: hello-v3-5fbc478857-94kql
Hello, world!
Version: 1.0.0
Hostname: hello-v2-b7fdf476b-vhh58
Hello, world!
Version: 1.0.0
Hostname: hello-v1-f5fcfc4c5-dv4s4
Hello, world!
Version: 1.0.0
Hostname: hello-v3-5fbc478857-94kql
Hello, world!
Version: 1.0.0
Hostname: hello-v3-5fbc478857-94kql
Hello, world!
Version: 1.0.0
Hostname: hello-v2-b7fdf476b-vhh58
Hello, world!
Version: 1.0.0
Hostname: hello-v1-f5fcfc4c5-dv4s4
```

Imagen: Como generar tráfico en la aplicación.
Fuente: Producción propia.

Si se realiza una gran cantidad de solicitudes y se contabilizan las respuestas de cada una de las versiones de *hello*, se obtendrían proporciones iguales, o sea, un 33,3% para cada una.

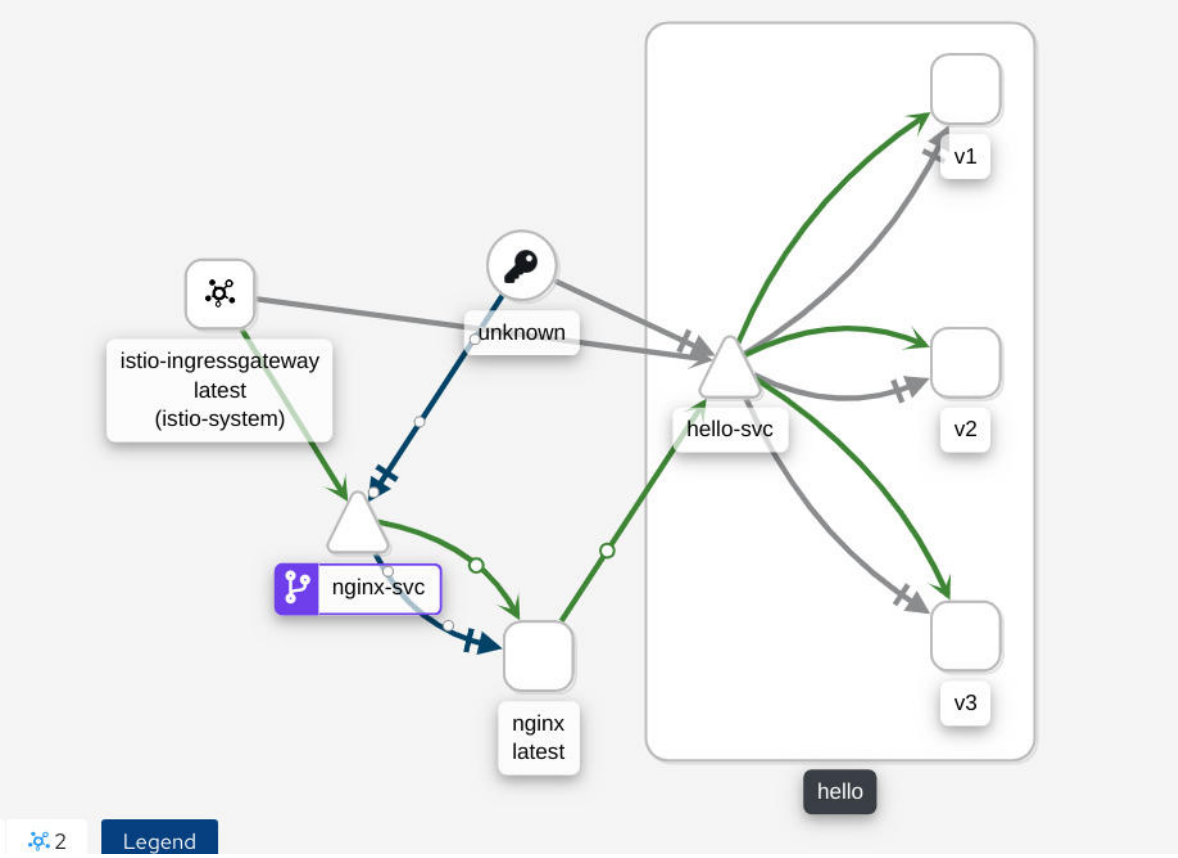


Imagen: Visualización de Kiali (panel de control de Istio).
Fuente: Producción propia.

Request routing

Es posible modificar el comportamiento del servicio de *hello* cuando se realizan solicitudes desde la pestaña *Action*. Para el ejemplo, se agrega una configuración que se comporta de la siguiente manera: si la solicitud del cliente tiene un *header* específico entonces la respuesta aplicará cierta regla.

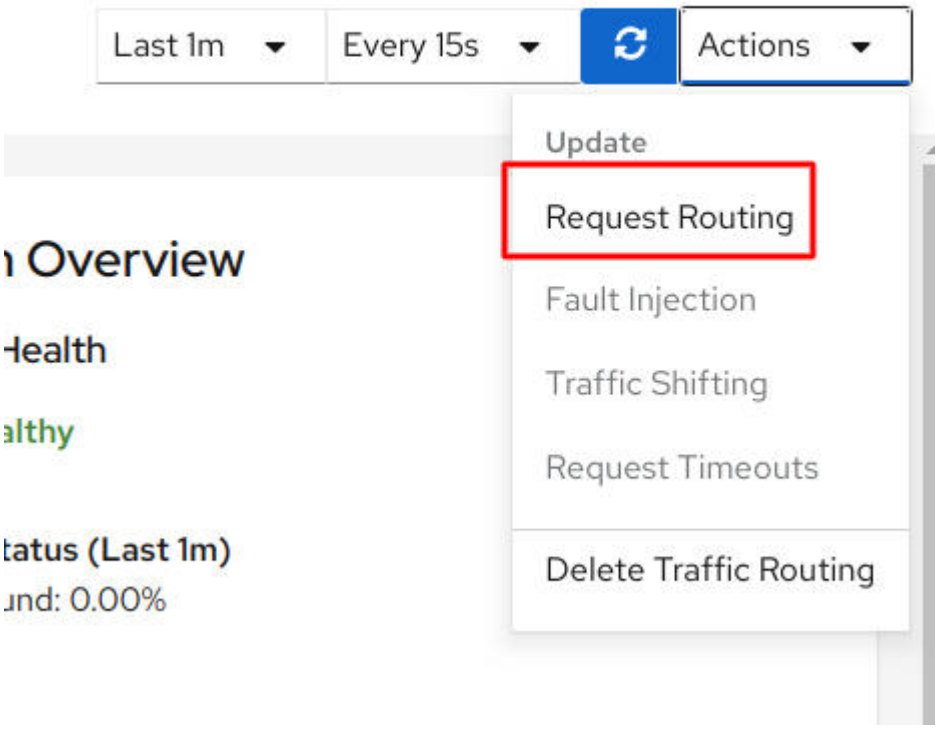


Imagen: Generar un *request routing* en Kiali.
Fuente: Producción propia.

Para nuestro ejemplo, se agrega que, si el *header version-app* es igual a *v1*, entonces el servicio devuelve la versión 1 de *hello*. Esta funcionalidad puede ser muy beneficiosa cuando existen entornos de desarrollo, testeo y producción en un mismo cluster. El equipo de desarrollo puede acceder al entorno de testeo o al de producción con solo modificar un parámetro de los *header* con un *plugin* del navegador.

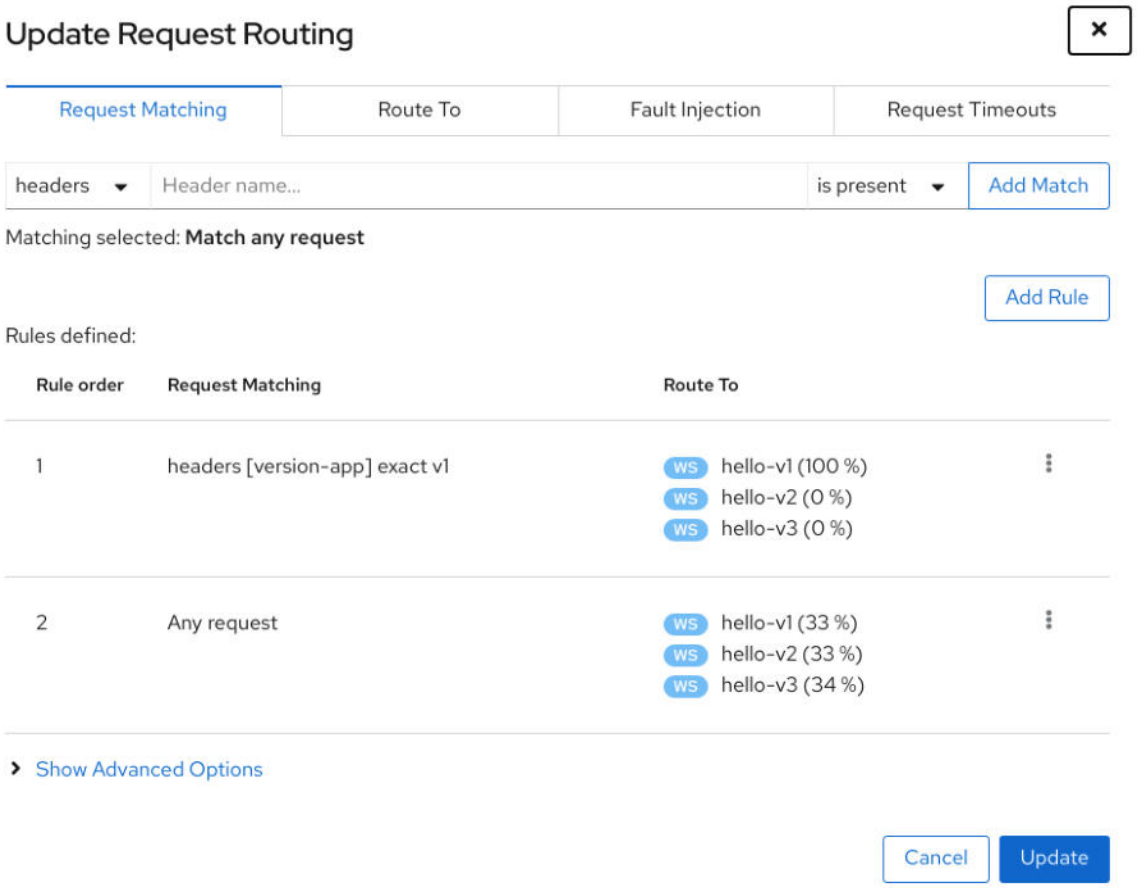


Imagen: Configuración de *request routing*.
Fuente: Producción propia.

Todos los demás usuarios ingresan a las tres versiones de *hello* de forma aleatoria (regla número 2 en la imagen de más arriba).

En la siguiente imagen, se observa la prueba cuando agregamos el parámetro en el *header* de la solicitud del cliente. En este caso, solo responde la versión 1 de la aplicación *hello*.

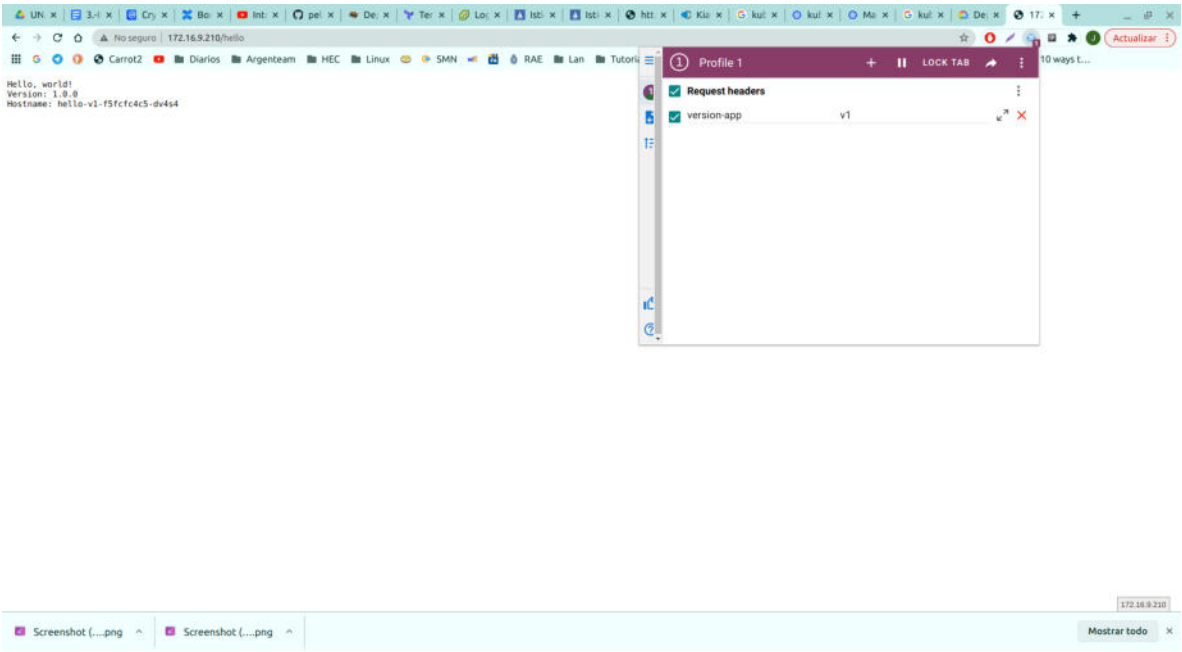


Imagen: Prueba en un navegador cuando se cambia el *header*.
Fuente: Producción propia.

Agregando inyección de fallas (*fault injections*)

Como ya se mencionó más arriba en esta PPS, una inyección de fallas es un método de prueba que introduce errores en un sistema para garantizar que pueda resistir y recuperarse de condiciones de error.

Para realizar la prueba se configura, en el panel de *fault injection*, que el 10% de las solicitudes de los usuarios devuelvan un código de estado HTTP 404 (*Not found*).

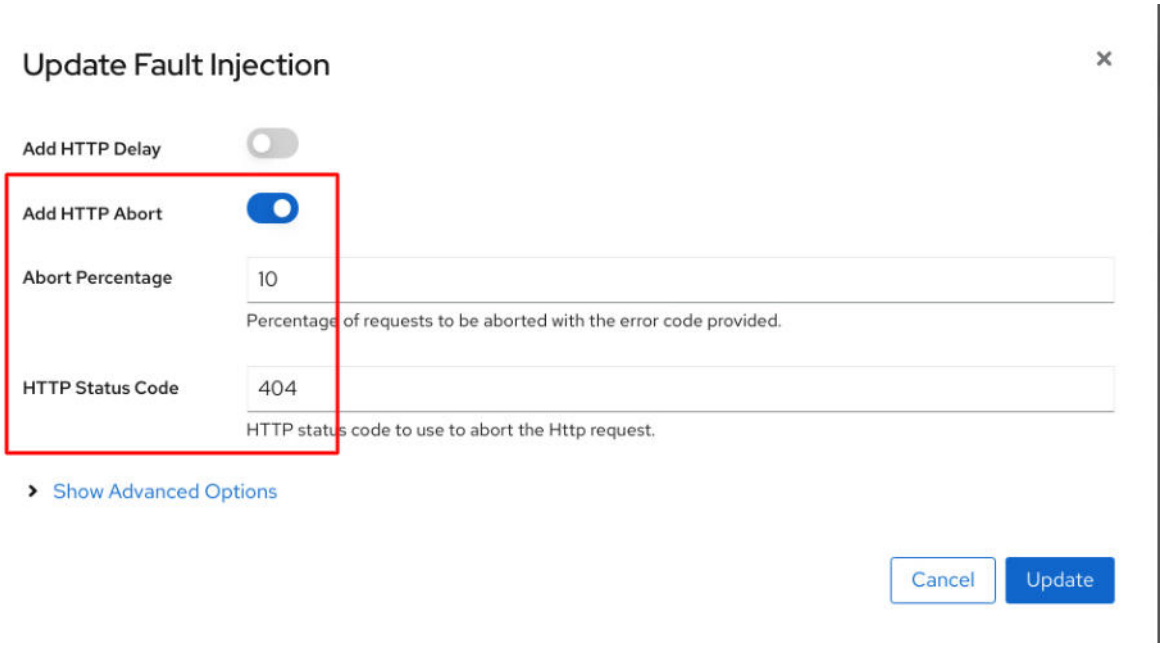


Imagen: Configuración de *fault injection*.
Fuente: Producción propia.

Para hacer una prueba se puede ejecutar el siguiente comando:

```
while sleep 1; do curl http://172.16.9.210:80/hello; done
```

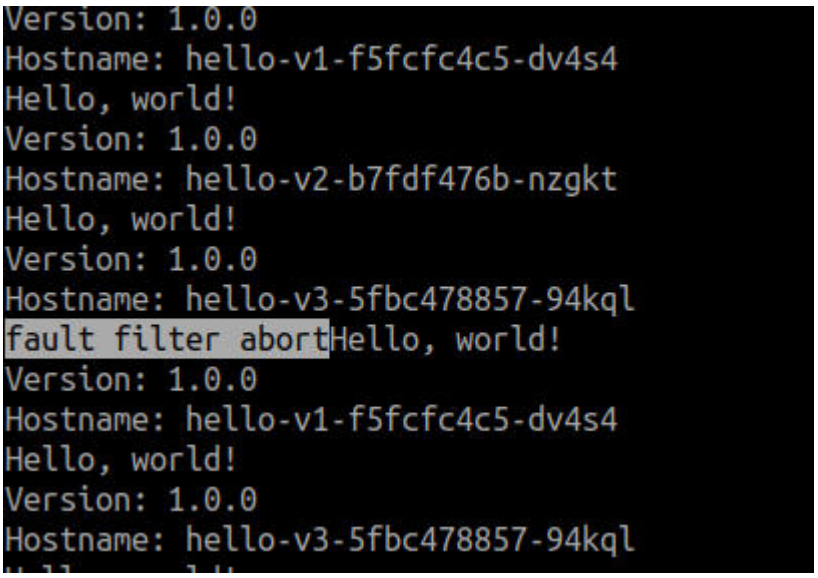


Imagen: Prueba de *fault injection*.
Fuente: Producción propia.

Como se ve en la imagen de arriba, un porcentaje definido de solicitudes van a responder con un mensaje HTTP 404. Esta herramienta es útil para garantizar que las políticas de recuperación de fallas no sean incompatibles, o demasiado restrictivas, lo que podría provocar que los servicios críticos no estén disponibles.

8. Conclusiones

De todo el proyecto de investigación, diseño y aplicabilidad propuesto en esta PPS, se arriba a las siguientes conclusiones: una de las funciones que debe cumplir el equipo de operaciones (SRE o *Site Reliability Engineering*) es dar las herramientas necesarias a los desarrolladores de *software* para desplegar aplicaciones, agregar nuevas funcionalidades y concentrarse solo en eso. Cuando se trabaja con micro-servicios, el equipo de desarrollo debe lograr independizarse de generar métricas, autenticaciones entre micro-servicios, permitir o denegar accesos entre servicios, y de la capa de infraestructura en general, que no debería ser parte de la responsabilidad de los desarrolladores. Muchas veces no es posible lograr esta independencia porque las herramientas que se utilizan para orquestar los contenedores no cubren esas funcionalidades. Un *software* de *service mesh* como Istio cumple con este requisito y cuenta con estas funcionalidades:

- Capacidad para asegurar la disponibilidad de la comunicación entre servicios, y de garantizar protocolos para llevar adelante esta.
- Posibilidad de ofrecer descubrimientos entre servicios.
- Generación de enrutamientos.
- Observabilidad detallada de la comunicación entre los micro-servicios.
- Seguridad para mitigar amenazas contra la información, *endpoints*, plataforma y comunicación, tanto internas como externas.
- Posibilidad de disponer de herramientas de autenticación/autorización para proteger los servicios e información.
- Capacidad para funcionar como un soporte nativo para el despliegue de contenedores.

Podemos encontrar algunas ventajas en usar esta aplicación como orquestador de la *service mesh,* a través del *dashboard* Kiali, como por ejemplo:

- Detecta el tráfico y muestra la arquitectura de la malla automáticamente.
- Monitorea el tráfico e incluso expone la dirección de las comunicaciones.
- Permite ver métricas, como por ejemplo la latencia, entre los micro-servicios.
- Encuentra errores muy fácilmente, sobre todo en ambientes donde corren decenas de micro-servicios y el monitoreo puede ser un desafío muy grande.

Las herramientas más importantes que encontramos son el equivalente al *Ingress* de Kubernetes, que en Istio se denomina *Gateway*. En él se declara el servidor donde se envía el tráfico (expone las aplicaciones) y, a través de *Virtual Service,* permite configurar cómo se enrutan las solicitudes a un servicio dentro de la *service mesh*.

Problemas encontrados y cómo se resolvieron

- Balanceador de carga

En un ambiente de producción *on premise* es mandatorio incluir un servicio de balanceo de carga, como por ejemplo *HAProxy*. Esto no es necesario en *clouds* públicas donde, generalmente, se incluye el servicio de balanceo de carga integrado en la plataforma.
Una vez iniciado el cluster, la dirección IP externa para el *ingress gateway* de Istio queda en estado pendiente. Para resolver este inconveniente se utilizó el *software MetalLB*[2] que es un balanceador de carga para Kubernetes en ambientes *bare metal*.

[2] https://metallb.universe.tf/

De esta forma, el nodo maestro actúa como balanceador de carga. Como el cluster solo tiene un nodo maestro, no tiene sentido el concepto de "balanceador de carga"; sin embargo, se configura un *ingress gateway* con una IP virtual privada que hace las veces de acceso externo a las aplicaciones que corren en el cluster.

- CoreDNS

Otro de los problemas encontrados fue que, al iniciar el cluster de Kubernetes, los *pods* de CoreDNS no iniciaban, quedaban pendientes, y el motivo era que, para iniciar los DNS, primero deben estar corriendo los *pods* de redes. Es necesario implementar un complemento de red de *pods*, basado en *Container Network Interface* (CNI), para que estos puedan comunicarse entre sí. Clúster DNS (CoreDNS) no se inicia antes de que se haya instalado una red.
Este problema se resolvió instalando el componente de red Flannel, que es uno de los varios CNI que pueden utilizarse en Kubernetes.

Comentarios Kubernetes en una nube privada (*on premise*)

Existe una desventaja al utilizar la arquitectura *on premise*, ya que Kubernetes tiene una pronunciada curva de aprendizaje y complejidad operativa. A diferencia de usar Kubernetes en una nube pública, como en AWS o Azure, donde su proveedor esencialmente le oculta todas las complejidades, ejecutar Kubernetes *on premise* significa que habrá que administrar estas complejidades, incluidos *etcd*, equilibrio de carga, disponibilidad, auto-escalado, configuraciones de redes, reversión de implementaciones defectuosas, almacenamiento persistente, entre otras.

Además, hay que tener en cuenta que se deben actualizar los clústeres con relativa regularidad cuando se publica una nueva versión de Kubernetes, así como también abordar las complejidades de administrar el almacenamiento y la supervisión del estado de salud de los clústeres en el entorno local de Kubernetes.

Algunas organizaciones simplemente no pueden usar la nube pública, ya que están sujetas a estrictas regulaciones relacionadas con el cumplimiento y los problemas de privacidad de los datos. Pero, lo que es más importante, las empresas podrían utilizar Kubernetes aprovechando sus centros de datos existentes para transformar su negocio y poder modernizar sus aplicaciones para entornos nativos de la nube, al tiempo que podrían mejorar la utilización de la infraestructura y ahorrar costos. Sin embargo, si bien hay un costo que se ahorra en infraestructura, existe otro costo que es el humano.

Como ya dijimos, Kubernetes puede ejecutarse en una infraestructura local, pero no de una manera sencilla. Puede reutilizarse el entorno para integrar Kubernetes, utilizando máquinas virtuales, hipervisores o creando un clúster sobre *bare metal*, pero no hay forma de escapar del requisito de una comprensión profunda de los servidores asociados, los sistemas de almacenamiento y la infraestructura de red. Cuando se ejecute Kubernetes en un centro de datos local, se deberá administrar todas las integraciones de almacenamiento, el balanceador de carga, los DNS, la administración de seguridad, el registro de contenedores y la infraestructura de monitoreo.

Dado que Kubernetes es un *software* relativamente moderno y la experiencia puede ser difícil de encontrar, la CNCF (*Cloud Native Computing Foundation*) ha introducido certificaciones como CKA (Administrador de Kubernetes) y CKAD (Desarrollador de aplicaciones de Kubernetes), que se pueden obtener pasando un examen. Si bien emplear administradores certificados por CNCF es una posibilidad, no todos pueden contratar personal nuevo.

En relación a las futuras extensiones del trabajo pueden mencionarse:

- Establecer una comparación con otras herramientas de *service mesh*, exponiendo ventajas y desventajas, funcionalidades y características.
- Analizar Istio como *software* de *service mesh,* en clústeres distribuidos.
- Implementar posibles soluciones con la limitación de rastreo distribuido.
- Investigar ambientes mixtos de producción, pre-producción y *testing* (*canary release*, *test A/B*, etc.).

9. Índice de imágenes

10. Referencias bibliográficas

Apasoft Training (2020). Kubernetes al completo. Recuperado de: https://www.udemy.com/course/kubernetes-al-completo/. (Fecha de consulta: 17 de junio de 2021).

Blancarte Iturralde, O. (2020). Arquitectura de Microservicios. Recuperado de: https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/microservicios. (Fecha de consulta: 17 de junio de 2021).

Chesterwood, R. (2021). Istio Hands-On for Kubernetes. Recuperado de: https://www.udemy.com/course/istio-hands-on-for-kubernetes/. (Fecha de consulta: 17 de junio de 2021).

Cloud Native Computing Foundation (2021). Building sustainable ecosystems for cloud native software. Recuperado de: https://www.cncf.io/. (Fecha de consulta: 17 de junio de 2021).

Cougil Grande, R. (30 de noviembre 2019). Observabilidad: logs, métricas y trazabilidad. Recuperado de: https://medium.com/@rcougil/software-observabilidad-logs-m%C3%A9tricas-y-trazabilidad-d5bcca56608d. (Fecha de consulta: 17 de junio de 2021).

F5 (2021). Build a Kubernetes Cluster. Recuperado de: https://clouddocs.f5.com/training/community/containers/html/appendix/appendix2/lab2.html. (Fecha de consulta: 17 de junio de 2021).

Fiz, J. M. (2019). ¿Sabes qué es SRE y qué puede hacer por tu negocio?. Recuperado de: https://www.paradigmadigital.com/techbiz/sabes-que-es-sre-y-que-puede-hacer-por-tu-negocio/. (Fecha de consulta: 17 de junio de 2021).

Fontani, D. (11 de mayo de 2020). Is Kubernetes on Premise viable?. Recuperado de: https://towardsdatascience.com/is-kubernetes-on-premise-viable-8b488368af56. (Fecha de consulta: 17 de junio de 2021).

Holloway, C. (15 de noviembre de 2018). ¿Qué son los contenedores de software y cómo aportan valor a las organizaciones?. Recuperado de: https://itmastersmag.com/noticias-analisis/que-son-los-contenedores-de-software-y-como-aportan-valor-a-las-organizaciones/. (Fecha de consulta: 17 de junio de 2021).

Istio documentation (2020). Learn how to deploy, use, and operate Istio. Docs. Recuperado de: https://istio.io/latest/docs/ (Fecha de consulta: 17 de junio de 2021).

Kubernetes (2021). Kubernetes documentation. Recuperado de: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. (Fecha de consulta: 17 de junio de 2021).

Platform 9 (21 de enero de 2020). Kubernetes On-premises: Why, and How. Recuperado de: https://platform9.com/blog/kubernetes-on-premises-why-and-how/. (Fecha de consulta: 17 de junio de 2021).

Posta, C. (2021). Make your app architecture cloud-native with a service mesh. Recuperado de: https://techbeacon.com/app-dev-testing/make-your-app-architecture-cloud-native-service-mesh. (Fecha de consulta: 17 de junio de 2021).

Rodríguez, T. (10 de septiembre de 2019). De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo. Recuperado de: https://www.xataka.com/otros/docker-a-kubernetes-entendiendo-que-contenedores-que-mayores-revoluciones-industria-desarrollo. (Fecha de consulta: 17 de junio de 2021).

St-Amand, D. (7 de septiembre de 2020). Installing a Kubernetes cluster on VMware vSphere and what I've learned. Recuperado de: https://www.domstamand.com/installing-a-kubernetes-cluster-on-vmware-vsphere-and-what-ive-learned/. (Fecha de consulta: 17 de junio de 2021).

Toribio, J. (5 de febrero de 2018). Consolida tu arquitectura de microservicios con Service Mesh. Paradigma Digital (Blog). Recuperado de: https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/. (Fecha de consulta: 17 de junio de 2021).

11. Anexo

➢ 01-helloapp.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v1
  labels:
    app: hello
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello
      version: v1
  template:
    metadata:
      labels:
        app: hello
        version: v1
    spec:
      containers:
      - name: hello
        image: gcr.io/google-samples/hello-app:1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
        resources:
```

```
          requests:
            memory: "64Mi"
            cpu: "200m"
          limits:
            memory: "128Mi"
            cpu: "500m"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v2
  labels:
    app: hello
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello
      version: v2
  template:
    metadata:
      labels:
        app: hello
        version: v2
    spec:
      containers:
      - name: hello
        image: gcr.io/google-samples/hello-app:1.0
        imagePullPolicy: IfNotPresent
        ports:
```

```
        - containerPort: 8080
        resources:
          requests:
            memory: "64Mi"
            cpu: "200m"
          limits:
            memory: "128Mi"
            cpu: "500m"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v3
  labels:
    app: hello
    version: v3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello
      version: v3
  template:
    metadata:
      labels:
        app: hello
        version: v3
    spec:
      containers:
      - name: hello
        image: gcr.io/google-samples/hello-app:1.0
```

```
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "64Mi"
            cpu: "200m"
          limits:
            memory: "128Mi"
            cpu: "500m"
---

kind: Service
apiVersion: v1
metadata:
  name: hello-svc
  labels:
    app: hello
    service: hello-svc
spec:
  selector:
    app: hello
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

➢ 02-nginx-proxy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        volumeMounts:
          - name: config
            mountPath: /etc/nginx/conf.d
      volumes:
        - name: config
          configMap:
            name: nginx-config
            items:
              - key: proxy.conf
```

```
                path: proxy.conf

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  proxy.conf: |-
    server {
      listen 80;
      location /hello {
        proxy_pass http://hello-svc;
        # https://github.com/envoyproxy/envoy/issues/170
        proxy_http_version 1.1;
      }
    }
---
kind: Service
apiVersion: v1
metadata:
  name: nginx-svc
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

➢ 03-ingress.yaml

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: hello-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway
implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello
spec:
  hosts:
  - "*"
  gateways:
  - hello-gateway
  http:
  - match:
    - uri:
        prefix: /hello
```

```
    route:
    - destination:
        host: nginx-svc
        port:
          number: 80
```