UNIVERSIDAD NACIONAL DE SAN JUAN

FACULTAD DE CIENCIAS EXACTAS FÍSICAS Y NATURALES

DEPARTAMENTO DE INFORMÁTICA

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

TRABAJO FINAL

Desarrollo de sistemas de microservicios, evaluación de modelos arquitecturales y mecanismos de seguridad en relación al rendimiento.

AUTOR

García Fontana, Sebastián José

ASESOR

Nelson Rodriguez

SAN JUAN

2026

Resumen

Los sistemas de microservicios se han convertido en la actualidad en un estándar en la industria para el desarrollo, debido a las múltiples ventajas que otorgan como facilidad para el escalado, menor complejidad para implementar cambios, robustez ante caídas de servicio, entre otras. Por lo tanto no es de sorprender la popularidad de tecnologías service mesh, añadiendo una capa de comunicación extra entre servicios, proveyendo un mayor grado de seguridad, observabilidad y control de tráfico, abstrayendo la complejidad en el networking del sistema.
El presente trabajo se enfoca en los aspectos de seguridad de los service mesh, más concretamente en el impacto que genera, en latencia, consumo de CPU y de Memoria, las implementaciones de mecanismos de seguridad mTLS, JWT y WAF en distintos modelos arquitecturales, API Gateway, Sidecar y Sidecar-Less.
Los resultados indican que tanto JWT como mTLS no presentan un impacto real en el rendimiento del sistema, mientras que WAF implica un gran consumo de recursos y debe ser utilizado con cautela, además el consumo de CPU puede significar un cuello de botella para el rendimiento si no se monitorea apropiadamente.

Palabras clave: Microservicios, Ciberseguridad, Sistemas Distribuidos, Kubernetes.

Abstract

Microservices systems have become an industry standard for development, this because of the multiple advantages they provide, such as easier scaling, less complexity to implement changes, and resilience to app failure, and more. So it's of no surprise the rise in popularity of service mesh technologies, adding an extra layer of communication between services, providing security, observability and traffic control, and abstracting the complexity of the system networking.
This work focuses on the security aspects of service meshes, specifically the impact in system resources, latency, CPU and Memory usage, that different security mechanisms, mTLS, JWT and WAF generate throughout multiple architectural models like API Gateway, Sidecar and Sidecar-Less.

The results show that both JWT and mTLS do not generate any impact in the systems performance, however WAF comes with a much higher resource consumption and therefore should be used cautiously, also CPU usage could become a bottle neck for system performance if not properly monitored.

Keywords: Microservices, Cyber Security, Distributed Systems, Kubernetes.

Índice

Glosario

AES (Advanced Encryption Standard) – Algoritmo estándar para la encriptación en Estados Unidos.

API (Application Programming Interface) – Interfaz de programación que permite la comunicación entre distintos dispositivos o programas.

CPU (Unidad Central de Procesamiento) – Componente de un computador encargado de procesar instrucciones y realizar operaciones aritméticas y lógicas.

HTTP (Hypertext Transfer Protocol) – Protocolo de comunicación que permite transferencia de información a través de diferentes formatos de archivos.

JWT (JSON Web Token) – Protocolo basado en archivos JSON para la creación de tokens de acceso que proveen un mecanismo de autenticación.

mTLS (mutual TLS) – Protocolo de seguridad basado en TLS donde ambos miembros de una comunicación deben autenticarse al crear una conexión.

OAUTH 2.0 (Open Authorization) – Protocolo de seguridad estandarizado para que aplicaciones puedan acceder a recursos de usuario almacenados en otros servicios sin que el mismo tenga que compartir su información de acceso.

SSL (Secure Socket Layer) – Protocolo de seguridad que cifra la información que se envía entre cliente y una página web.

SOA (Service Oriented Architecture) – Modelo arquitectural donde el sistema se plantea como servicios como unidades funcionales básicas del sistema.

SO (Sistema Operativo) – Conjunto de software de sistema encargado de gestionar hardware y recursos de software, así como proveer servicios básicos del sistema.

REST (Representational State Transfer) – Estilo arquitectural diseñado como guía para describir el desarrollo de arquitecturas en internet.

RSA (Rivest, Shamir y Adleman) – Sistema criptográfico de clave pública.

TLS (Transport Layer Protocol) – Protocolo criptográfico que encripta comunicaciones en internet, sucesor de SSL.

URL (Uniform Resource Locator) – Tipo de identificador de recursos uniforme (URI) que se utiliza para identificar la dirección de recursos variables.

WAF (Web Application Firewall) – Tipo de firewall que provee seguridad a aplicaciones web supervisando comunicaciones HTTP entrantes y salientes.

Introducción

En la actualidad existe una gran demanda para el desarrollo de sistemas de microservicios (SDM), que con el paso del tiempo se han comprobado los beneficios que presentan por encima de las arquitecturas monolíticas, por lo que no es de extrañar una expectativa de constante crecimiento de uso en la industria [1].

En las arquitecturas monolíticas la complejidad que genera ampliar o modificar las funcionalidades, así como también la corrección y actualización de componentes, rápidamente se convierten en un cuello de botella para aplicaciones con necesidades de crecimiento. Es en este contexto que los SDM surgen como alternativa para solucionar los inconvenientes de trabajar con arquitecturas monolíticas. Un SDM funciona creando unidades individuales llamadas servicios que trabajan de forma independiente, comúnmente llevando a cabo una sola actividad de negocio por servicio, y que luego, se comunican a través de mensajes para resolver actividades más complejas.

A pesar de sus ventajas los SDM traen consigo otros problemas propios de su naturaleza como sistemas distribuidos, entre los que se destacan algunos como: mayor superficie de ataque al generar mayor cantidad de puntos de entradas, el manejo de la autentificación, inmutabilidad de contenedores, etc. Estos problemas se hacen frente con protocolos y herramientas como HTTPS, JWT, WAF, OAUTH 2, SSL,TLS, etc. En éste trabajo se ahondan en aspectos de seguridad y su impacto en la eficiencia del sistema.

Un SDM de gran tamaño puede llegar a poseer cientos o miles de servicios en funcionamiento [2], los mismos se suelen instanciar en contenedores individuales, que debido a su enorme cantidad, causa que se vuelva inviable la administración manual de cada instancia. Es por esto que se utilizan orquestadores, que se encargan de automatizar la administración de los contenedores, quitándole esa carga al programador. Aun así los servicios que brindan los orquestadores suelen ser básicos para el correcto funcionamiento y administración del sistema, debido a eso en muchas ocasiones es deseable la implementación de otros mecanismos como por ejemplo API Gateway o Service Mesh.

Por un lado, un API Gateway facilita la comunicación entre usuario y sistema al crear un punto individual para acceder a los servicios, en lugar de necesitar múltiples puntos de acceso para cada servicio, mejorando la comunicación Norte-Sur (Usuario-Sistema). Mientras que por otro lado el modelo Service Mesh funciona como una capa extra dentro del sistema que facilita la comunicación entre los servicios dentro del sistema, mejorando la comunicación Este-Oeste (Sistema-Sistema).

Todas estas opciones para la implementaciones de comunicaciones y seguridad presentan una decisión interesante cuando se debe elegir cómo se utilizarán los mecanismos de seguridad en las distintas capas que componen el SDM, como por ejemplo el API Gateway o Service Mesh, esta última a su vez contando con 2 modalidades en la actualidad sidecar y sidecar-less. Es por esto que en el presente trabajo se busca realizar un análisis de dichas implementaciones, el impacto que tienen en el rendimiento del sistema, beneficios y desventajas.

Justificación

Existen estudios en los que se realizan análisis de rendimientos en SDM como:

- M. Matías et al. [3] en donde se realiza un estudio de los tiempos de respuesta cuando se utiliza SSL en un SDM.
- Y. Dawei. et al. [4] plantea el uso de los algoritmos RSA y AES junto con un API Gateway para mejorar la seguridad de los sistemas, junto con un análisis del impacto en los tiempos de respuesta que genera.
- M. R. S. Sadeghpour et al. [5] que realiza la implementación de un Circuit Breaker propio en un service mesh como alternativa los Circuit breaker estáticos, el impacto en las peticiones respondidas y no respondidas, así como las diferencias en tiempos de respuesta.
- X. Zhu et al. [6] indica como la implementación de un service mesh genera una gran cantidad de overhead al sistema que los desarrolladores muchas veces desconocen, incrementando la latencia y uso de CPU. Es por esto que realiza un estudio sobre los factores que conllevan a este overhead y desarrolla un software que ayuda a estimar el overhead generado por un service mesh.

- Y. Elkhatib y J. P. Poyato [7] compara dos tecnologías de service mesh muy populares: Istio y Linkerd, en el ámbito de tecnologías edge y su repercusión en los tiempos de respuesta, memoria y uso de CPU.
- A. B. Barr et al. [8] hace un análisis del consumo de recursos (Latencia, CPU y Memoria) de mTLS en diferentes tecnologías service mesh de amplio uso.

En los trabajos encontrados en su mayoría realizan análisis comparativos de diferentes tecnologías service mesh sin la implementación de mecanismos de seguridad, o aquellos que si los implementan el enfoque se mantiene en una de las 2 implementaciones API Gateway o Service Mesh, ya sea sidecar o sidecar-less, sin compararlas.

Aun así es importante entender porque se elegiría una opción de implementación sobre la otra. El motivo de mayor importancia por el cual las empresas deciden implementar un service mesh es la seguridad, entre muchas otras. Pero, la performance del sistema se mantiene como un desafío técnico relevante [9], lo que puede apuntar a implementar seguridad solo en el API Gateway, ya que requiere de muchos menos recursos y están enfocadas en ser mejores para la performance del sistema.

Es aquí donde surge una problemática, si no se conoce con exactitud el consumo de cada alternativa, y los requisitos de seguridad del sistema, se puede dar una situación donde la implementación de un service mesh resulte excesivo produciendo un overhead que dañe la experiencia usuario, genere incrementos en los costos operacionales y decrementos en los ingresos [6]. O por el contrario que una implementación solo de gateway, enfocado exclusivamente en la búsqueda de maximizar performance provoque falencias en la seguridad del sistema, y por extensión del usuario.

Es por esto que un análisis cuantitativo puede resultar útil para la ayuda en la toma de decisiones, facilitando así poder determinar si la elección de una implementación se ajusta a los objetivos que se esperan del sistema.

Finalmente mencionar los recibimientos positivos tanto de una versión preliminar en la cual se comparó una evaluación del impacto de HTTPS y WAF en SDM de nodo singular y nodos múltiples en el Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI) 2025, que inspiró la continuación de una versión más completa en este trabajo. Como también un artículo con los resultados obtenidos en este presente trabajo enviado a la Revista Electrónica de Tecnología, Educación y Ciencia (ReTEC) de la Universidad Nacional de Salta, que al momento de escribir este informe se encuentra en proceso de impresión.

Objetivos

General: Desarrollar un sistema de microservicios basados en los modelos API Gateway y Service Mesh para evaluar el impacto de la implementación de los mecanismos de seguridad.

Específicos:

- Analizar alternativas tecnológicas para la implementación de sistemas de microservicios, de mecanismos de seguridad y medición.
- Diseñar y desarrollar un sistema para la experimentación.
- Validar propuesta.

Marco Teórico

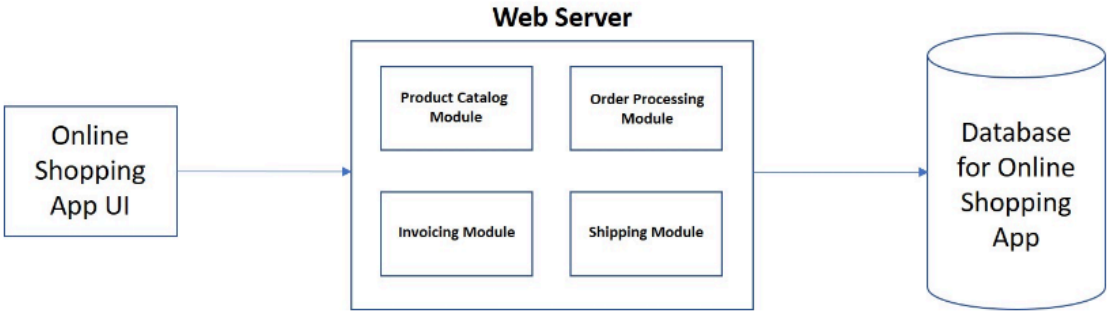
Arquitectura monolítica

Los primeros sistemas en ser desarrollados se hicieron bajo una arquitectura monolítica. Esta arquitectura implica el desarrollo y deploy de una aplicación en un solo gran componente individual, con un alto acoplamiento entre cada una de las partes que lo forman. Rápidamente surgieron problemas con éste diseño, por un lado en tiempo de ejecución, ya que la caída de una parte de la aplicación supone la caída de la aplicación en su totalidad en la mayoría de los casos. Pero también de escalabilidad y mantenimiento, debido al alto acoplamiento del sistema se vuelve casi imposible determinar el origen de un problema o la modificación de una parte de código sin tener efectos inesperados en las otras partes. Estos problemas ralentizan el trabajo de los desarrolladores y por extensión de la performance del sistema [10].

Arquitectura de Microservicios

Como alternativa a las arquitecturas monolíticas surgió la arquitectura de microservicios. La misma está compuesta por unidades básicas llamadas servicios, los cuales suelen (por buena practicas) estar asociados a una única actividad de negocio de las que debe realizar el sistema. Los servicios se desarrollan para poder trabajar de forma independiente, añadiendo mecanismos de comunicación para permitir el trabajo cooperativo y resolver las dependencias en tiempo de ejecución que surjan entre ellos [11].

A continuación se muestra una comparación esquemática entre un mismo sistema desarrollado con una arquitectura monolítica y una arquitectura de microservicios.



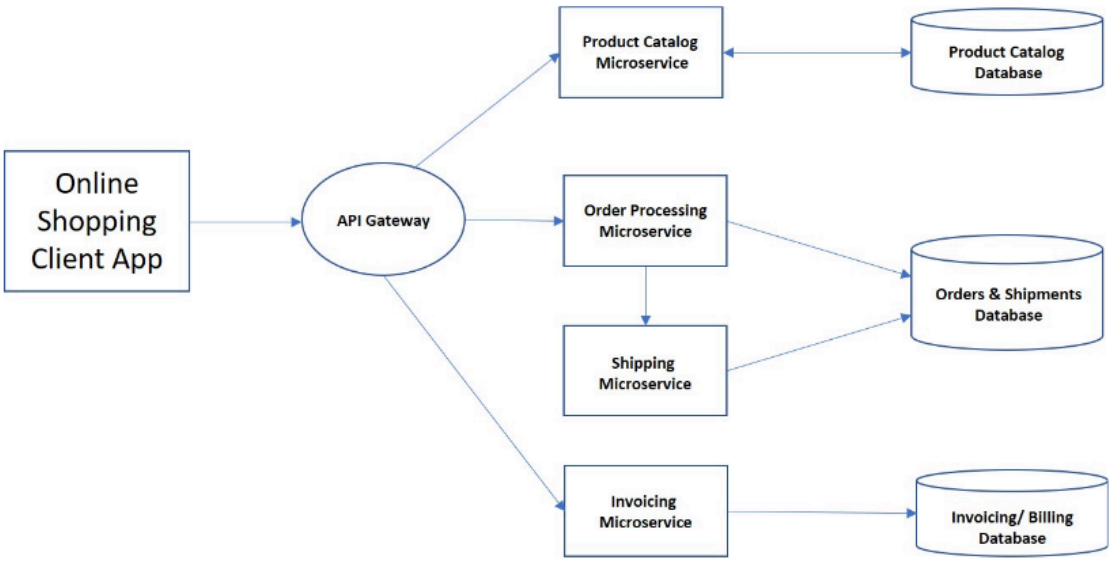


Figura 1: Comparación de un sistema monolítico y un sistema de microservicios. Reproducido de [11].

En este sentido son similares a un modelo de Arquitectura Orientada a Servicios (SOA), ya que ambos proveen servicios los cuales tienen capacidades de comunicación entre ellos con dependencias mínimas, pero cabe aclarar que los microservicios son solo una representación refinada del SOA [11].

Rápidamente por los problemas de la arquitectura monolítica y la aparición de propuestas basadas en la nube se ha producido un cambió en la adopción de microservicios por sobre arquitecturas monolíticas. Generando a su vez un gran mercado interesado en el uso e implementación de los microservicios, principalmente en la nube [1].

Entre sus ventajas tenemos [11]:

- Independencia entre equipos desarrollando componentes separados. Cada uno pudiendo elegir herramientas, lenguajes, middleware, hardware, entre otros a utilizar.
- Escalado independiente de los componentes.
- La implementación de interfaces de Transferencia de Estado Representacional (REST) permiten modificar la lógica de los componentes sin afectar al funcionamiento de las comunicaciones siempre que se mantengan dichas interfaces.
- Cada componente al poseer menor cantidad de contenido agiliza el desarrollo e implementación de actualizaciones.
- El bajo acoplamiento disminuye el impacto que puede generar la caída de un servicio en el sistema.
- El uso de manejo de eventos asincrónicos para la comunicación entre servicios disminuye más el impacto de la caída de un servicio.
- Al alinear cada servicio con una función de negocio, la arquitectura general del sistema se alinea con la estructura de la organización. Facilitando el cambio de los servicios ante cambios organizacionales.
- La independencia de servicios fomenta la reusabilidad de código.

Pero a su vez por su naturaleza distribuida se adicionan nuevas dificultades a resolver en la utilización de microservicios, como lo pueden ser [11] [12] [13] [14]:

- Mayor superficie de ataque, debido a las vulnerabilidad que generan los endpoints de cada servicio.
- La validación de las peticiones puede impactar al desempeño del sistema por su naturaleza distribuida. En una aplicación monolítica se realiza una única vez, pero en microservicios se puede generar redundancia al generar verificaciones para cada microservicio.
- Implementar confianza en los canales de comunicación entre servicios, y el trabajo necesario para el manejo de las credenciales que necesita cada microservicio para garantizar la confianza.
- La necesidad de la observabilidad para poder hacer seguimiento de peticiones que viajan entre múltiples servicios a través del uso de logs, métricas y trazas.
- Cada servicio se despliega en un contenedor propio, que por buenas prácticas debería ser inmutable. Esto implica que el servidor no debería cambiar los archivos una vez iniciado el contenedor ni manejar información extra durante el tiempo de ejecución, lo que dificulta el implementar listados de clientes permitidos y políticas para el acceso a los servicios.
- El diseño distribuido dificulta poder compartir el contexto de usuario entre servicios.
- Cada servicio puede ser escrito en un lenguaje distinto, lo que genera una necesidad de mayor habilidad en el desarrollo multi-equipo, donde cada equipo necesitará a un experto de seguridad para una tecnología concreta.
- Corrupción a la base de datos del servicio de registros que conlleva a redirecciones de peticiones a servicios incorrectos resultando en denegaciones de servicios, también, redirección a servicios maliciosos poniendo en riesgo al sistema en su totalidad.
- Fallos en cascada por la dependencia en las comunicaciones entre servicios.

Contenedores

Un contenedor es una unidad de software que está lista para correr de forma independiente, ya empaquetados con todas sus dependencias y configuraciones necesarias [15]. Están basados en tecnología de virtualización, siendo una alternativa popular a las máquinas virtuales, debido a ser mucho más ligeras, fáciles de transportar y crear, y por el aislamiento que confieren entre aplicaciones [12] [15].

Para entender mejor la importancia de los mismos se debe analizar su historia [16].

- **Era de despliegue tradicional:**
  Al principio, las organizaciones corrían sus aplicaciones en servidores físicos. No existía forma de definir límites a los recursos para las aplicaciones en los servidores físicos, lo que traía problemas con asignación de los recursos. Con casos donde por ejemplo una aplicación consumiría todos los recursos dejando en inanición a otra. Un posible solución era correr una aplicación por servidor, pero provoca que los recursos se subutilicen y genera grandes costos para las empresas.
- **Era de despliegue virtualizado:**
  Como solución se introdujo la virtualización. Esta permite correr múltiples máquinas virtuales (MVs) en la CPU de un solo servidor físico. La virtualización permite que las aplicaciones estén aisladas entre MVs y provee una capa extra de seguridad ya que una aplicación no puede acceder a la información de otra.
  La virtualización permite utilizar mejor los recursos del servidor físico y permite mayor grado de escalabilidad ya que las aplicaciones pueden ser añadidas o actualizadas fácilmente, reduce costos de hardware, entre otros.
  Cada MV es una máquina entera funcionando con todos sus componentes, incluyendo su propio sistema operativo, encima de hardware virtualizado.

- **Era de despliegue contenedorizado:**
  Los contenedores son similares a las MVs, pero con propiedades de aislamiento más relajadas al compartir el Sistema Operativo (SO) entre aplicaciones. Es por esto que son considerados mucho más ligeros, de forma similar a las MV poseen su propio sistema de archivos, parte de la CPU, memoria, espacio de procesamiento, y otros. Al estar desacoplados de la infraestructura subyacente se vuelven portables entre diferentes distribuciones de SO y de la nube.

En la siguiente imagen, se puede ver esquemáticamente el proceso evolutivo de los contenedores y la tecnología de virtualización.

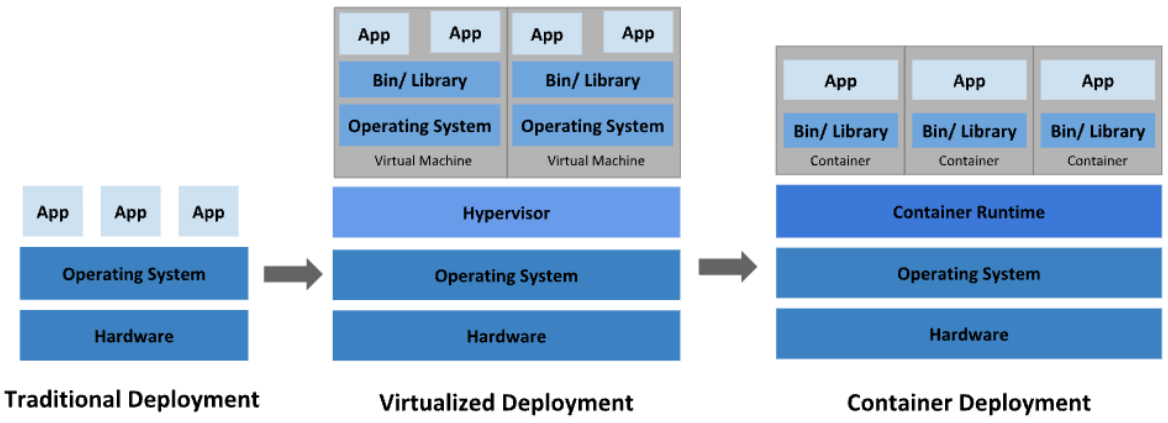


Figura 2: Esquema de evolución de las tecnologías de virtualización. Reproducido de [16].

En la actualidad el software más conocido para la creación de contenedores es Docker, el cual ofrece una plataforma que facilita la creación y verificación de contenedores, sustentado por una gran comunidad que comparte contenedores propios. Aun así existen otras alternativas como Podman, CRI-O, Containerd o LXC.

Orquestadores

Si bien la creación y uso de contenedores facilita una gran parte del trabajo para el uso de microservicio, estos traen consigo un gran problema: para aplicaciones complejas que utilizan gran cantidad contenedores la administración manual de los mismos se vuelve imposible en la práctica.

Para solucionar este problema es que se utilizan orquestadores. Los orquestadores funcionan como programas encargados de actividades relacionadas a la administración de contenedores como por ejemplo la creación, escalado, balanceo de carga, chequeo de salud, eliminación de contenedores no operativos, entre otros. En la actualidad uno de los orquestadores más utilizados es Kubernetes (k8s), es una alternativa de código abierto que funciona como plataforma de herramientas para la administración de contenedores. Otra alternativas son Docker Swarm u OpenShift.

API Gateway

Como se establece en [11] un API Gateway es un framework arquitectónico que funciona como un punto de entrada único para el sistema, desde el cual los clientes pueden acceder, el mismo redirige los mensajes a los servicios correspondientes evitando así tener que conocer y crear comunicaciones punto a punto con cada servicio.

Si bien su función principal es la de enviar los mensajes a cada servicio, también es capaz de realizar otras tareas como traducción de protocolos o composición de las peticiones.

Además es recomendable que esté preparado para manejar actividades tales como: descubrimiento de servicios, autentificación, acceso de control, balanceo de carga, cache, API personalizadas para cada cliente, chequeos de salud de servicios, monitoreo, detección y respuesta de ataques, registro y monitoreos de seguridad y circuit breakers.

A continuación en la figura 3 se hace una representación gráfica de un API Gateway y todas sus funcionalidades.

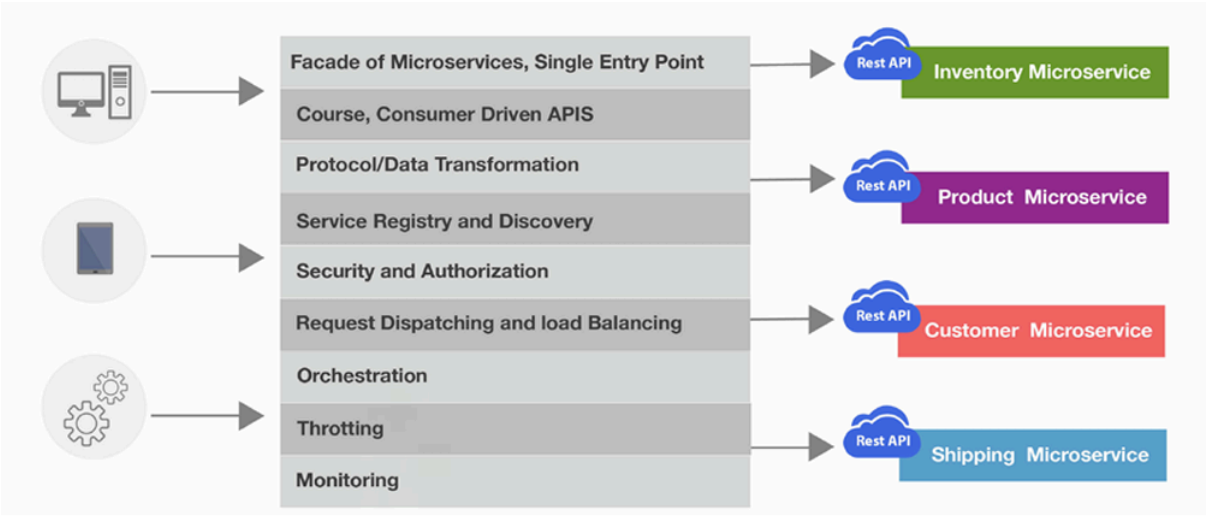


Figura 3: Funciones de un API Gateway. Reproducido de [17]

El API Gateway puede implementarse de manera monolítica o distribuida. En una implementación monolítica se establece un único Gateway en el borde de la aplicación proveyendo todos los servicios correspondientes. En cambio una implementación distribuida consta de varios micro gateways, que consta de un punto de entrada de un tamaño mucho más acotado y fácilmente configurable que es mucho más apropiado para aplicaciones con servicios que posean requisitos de seguridad independientes [11].

A su vez se puede tomar la decisión sobre si implementar un API Gateway con estado o sin estado. Una implementación sin estado se alinea más a una arquitectura RESTful, mejora la escalabilidad horizontal y simplifica el cacheo y recuperación de errores, mientras que una implementación con estado puede mejorar la experiencia de usuario final y permitir interacciones más complejas [18].
Es importante distinguir la diferencia entre un API Gateway con un reverse proxy. Si bien ambos pueden parecer similares en primera instancia no funcionan de la misma manera. Se entiende por proxy a un sistema o dispositivo que realiza de intermediario entre un usuario e internet. Un proxy actúa en nombre de los usuarios administrando el tráfico de salida, mientras que un reverse proxy actúa en nombre del sistema y administra el tráfico de entrada.

Un reverse proxy principalmente realiza el enrutamiento de los clientes a los servicios de backend. Tiene responsabilidades como: balanceo de carga, terminación SSL y enrutamiento simple basado en URL o nombres de dominios. Por lo general son sin estado, y operan a nivel de capa de red o transporte, por lo que tienen poca visibilidad de la capa de aplicación. Por otro lado, el API Gateway está diseñado específicamente para comunicación API en las arquitecturas de microservicios. Por lo que provee capacidades más diversas como las expresadas anteriormente. En resumen un reverse proxy ofrece enrutamiento básico y balanceo de carga, mientras que el API Gateway provee interfaz centralizada, para el manejo, seguridad y optimización de las interacciones entre clientes y microservicios [18].

Algunas de las tecnologías en el mercado para la implementación de API Gateway en la actualidad son:

- Agentgateway
- Cilium
- Envoy Gateway
- Google Kubernetes Engine
- HAProxy Ingress
- Istio
- NGINX Gateway Fabric
- Traefik Proxy

Service Mesh

Un service mesh funciona como una capa de infraestructura dedicada a facilitar la comunicación entre servicios. Como objetivo final tiene la función de facilitar la administración del tráfico en un ambiente altamente dinámico, con servicios que se crean y eliminan en tiempo de ejecución. Esto lo realizan con utilidades tales como: descubrimiento de servicios, enrutamiento y balanceo de carga interno, configuración de tráfico, encriptación, autenticación y autorización, métricas, monitoreo y observabilidad, circuit breaker o mecanismos de reintento. Todo esto provee a crear un sistema mucho más robusto y resistente a los constantes problemas de un sistema distribuido [2] [11].

El service mesh está compuesto por 2 planos: Un plano de información y uno de control.

El plano de información se encarga de crear el camino de datos para poder transmitir la información de las peticiones a los servicios, esto lo hace a través de proxies colocados próximos a cada servicio, a estos proxies se los suele llamar “sidecar proxy”, que están encargados de llevar a cabo en tiempo de ejecución las políticas de seguridad establecidas por el plano de control.

Pero en la actualidad también está incrementando la popularidad de alternativas “sidecar-less”. Dicha alternativas integran el service mesh directamente sobre la infraestructura en la que corren, se basan en herramientas como Daemon Sets, proxies a nivel de nodo, o tecnologías como eBPF o protocolos de aplicación a nivel de kernel. Entre sus ventajas se incluyen menor complejidad operacional o mejor eficiencia, pero con desventajas como falta de madurez o posibles deficiencias en seguridad comparado con un modelo sidecar [19].

Mientras que el plano de control son API y herramientas que se utilizan para manejar la configuración y el correcto funcionamiento del plano de datos, es importante no confundirlo con el plano de control del orquestador [11] [20].

En la siguiente figura se muestra un esquema de cómo se compone un service mesh y la comunicación entre dichos componentes.

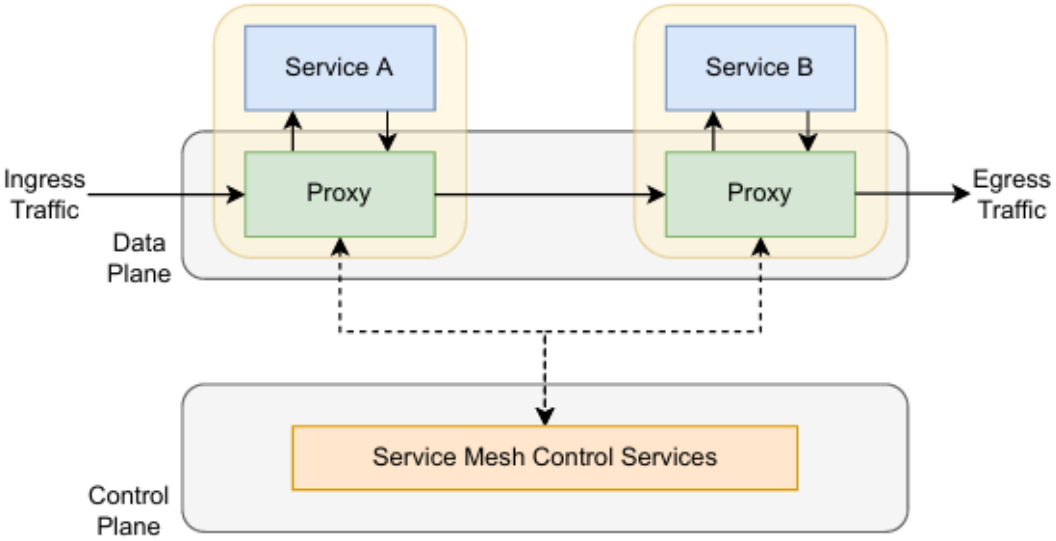


Figura 4: Componentes de un service mesh. Reproducido de [2]

En la actualidad algunas de las alternativas más usadas son:

- Istio
- Linkerd
- HashiCorp Consul
- Cilium

Amenazas y Mecanismos de seguridad

Se puede definir a los mecanismos de seguridad como herramientas técnicas y métodos técnicos que se utilizan para implementar los servicios de seguridad [21]. Estos mecanismos buscan implementarse para contrarrestar el riesgo producido por amenazas y ataques producidos contra el sistema. Entre algunos de los métodos que se pueden implementar para proteger al sistema se tiene: autenticación y autorización, monitorización de tráfico y principios criptográficos para confidencialidad e integridad de la información, entre otras.

Para el monitoreo del tráfico se puede realizar implementaciones de tecnologías Web Application Firewall (WAF) tanto en el API Gateway, para el monitoreo del punto de entrada al sistema completo, como dentro del Service mesh en los sidecar, para el monitoreo de cada servicio.

En cuanto a encriptación se refiere es altamente recomendable el uso de SSL/TLS tanto en el API Gateway como para la comunicación entre servicios, protegiendo de ataques como man-in-middle o la alteración de la información contenida en la comunicación [17]. Aunque el uso de SSL a nivel de servicios puede resultar costoso, una alternativa es mantener viva la conexión entre dos servicios luego de enviar el último dato, evitando tener que realizar un nuevo handshake ahorrando tiempo en la comunicación [11].

La autentificación es el proceso de determinar si alguien o algo es, de hecho, quien aseguran ser. Autorización es el proceso de otorgar a alguien o algo el permiso para hacer o poseer algo [14]. Lo más eficiente para la implementación de autenticación en SDM es el uso de punto centralizado para la administración de las políticas de acceso [11]. Este punto puede ser una entidad federal que acredite la identidad del usuario o servidor de autorización propio del sistema. En ambos casos toda petición a la API del sistema se deberá autenticar en el Gateway a través de un token de acceso, en donde se validará y dará un token con la información de autorización a los servicios.

Este esquema tiene varios inconvenientes como: el peligro del “confused deputy problem”, la homogeneidad de los requisitos de autorización para los servicios o un cuello de botella en el administrador de credenciales. El “confused deputy problem” es una situación donde un servicio con privilegios es comprometido. En este caso particular al explotar con intenciones maliciosas la verificación del Gateway, se produce que los servicios procesan peticiones dañinas provenientes del exterior ya que los mismos confían en las autorizaciones proveídas por el Gateway [14] [17].
Es por esto que conviene añadir validaciones extra a nivel de servicio, y a su vez se puede crear lógica extra para validaciones específicas que puedan requerir cada servicio. Esto crea una clara diferenciación entre sistemas en los cuales existe confianza entre los servicios y aquellos en los cuales toda comunicación se considera insegura incluso entre servicios del mismo sistema o “zero trust”.

Marco Legal

Para destacar como marco legal se encuentran las licencias de los softwares utilizados para la investigación, que al tratarse de open source son más permisivas y por lo tanto no se encontraron conflictos para el desarrollo del trabajo.

Hipótesis

Según lo planteado en el marco teórico, es muy posible que una implementación de los mecanismos de seguridad en un API Gateway resulte mucho más eficiente que en un Service Mesh al implicar menor cantidad de comunicaciones, lo cual debería de verse reflejado en una menor latencia y utilización de recursos.

Metodología

Se seleccionó una metodología experimental, desarrollando un sistema que intenta simular lo máximo posible un cluster pequeño real de producción de kubernetes, sobre el cual aislar cada uno de los mecanismos a evaluar y realizar pruebas. Dichas pruebas consisten en generar peticiones HTTP/S que lleven al límite al sistema para poder obtener métricas cuantitativas sobre el consumo de recursos, con las que luego comparar las distintas implementaciones.

Desarrollo del trabajo

Herramientas

Debido a la necesidad del trabajo de crear contenedores de servicios propios para la evaluación se optó por Docker. Esto debido a su amplia popularidad, sencillez de uso y compatibilidad con las herramientas de orquestador elegidas, Kubernetes y Kind.

Como tecnología de orquestación se eligió Kubernetes, la cual es una plataforma open source para la administración de contenedores. Funciona principalmente a través de la API que expone para manipular el estado de los objetos dentro del sistema. Dichos objetos pueden ser definidos de forma declarativa en archivos .yaml, que serán luego las plantillas a partir de las cuales el sistema intentará buscar su estado ideal. Algunos objetos a destacar serían los Deployment en donde se definen las características de los servicios a desplegar (número de réplicas, nombres, distribución, imágenes, etc) y los Service que permite exponer una red entre cada instancia de los servicios, facilitando la comunicación con los mismos.

Los componentes de k8s se organizan en una estructura jerárquica. En primera instancia se encuentra el cluster, la unidad dentro de la cual corre el sistema en su totalidad. Luego posee nodos, estos nodos son los encargados de de correr los servicios tanto los propios de k8s y como los del sistema desarrollado, entre ellos se realiza un balanceo de carga de las múltiples instancias de los servicios y establecen comunicaciones entre sí a través de proxys y registros propios. Finalmente están los pods, los cuales son los servicios en sí mismos con sus contenedores, que por lo general consiste en un contenedor por pod pero en este caso el service mesh puede inyectar sus sidecar en cada pod, por lo que finalmente se tiene 2 contenedores por pod, uno de aplicación y otro sidecar.

Kubernetes posee múltiples formas para poder implementarse. En una primera instancia se había optado por utilizar minikube ya que es una opción recomendada por Kubernetes para comenzar el desarrollo de un sistema. Pero por limitaciones en el funcionamiento para sistemas multi-nodo finalmente se eligió kind, al ser también avalado por k8s y haber sido desarrollado principalmente para el testing de k8s.

Para la implementación del API Gateway y Service mesh se optó por Istio principalmente por la capacidad para implementar ambos modelos arquitecturales (sidecar y sidecar-less) y soporte nativo para los mecanismo de seguridad a evaluar (JWT, mTLS, y WAF). Istio es un service mesh open source que se aplica sobre aplicaciones distribuidas ya existentes. Por parte del API Gateway istio tiene 2 alternativas para la implementación.

Por un lado se puede utilizar el Gateway API, un Custom Resource Definition (CDR) que brinda kubernetes. Funciona como una API que permite definir de forma declarativa recursos con los que implementar un API Gateway. Estos mismos se organizan entre diferentes tipos de API que tienen relaciones interdependientes, GatewayClass, Gateway y Routes. Un objeto Gateway debe estar asociado a un único GatewayClass, donde el GatewayClass describe el controlador asociado encargado del manejo de los Gateways que corresponda a esta clase. Finalmente se puede crear una o varias Routes, ya sean HTTP o gRPC.

El diagrama de flujo de la comunicación entre componentes de un Gateway API de Kubernetes sería como se ve a continuación:

Figura 5: Diagrama de flujo entre componentes Gateway. Reproducido de [22]

Esta API solo permite definir la infraestructura para enrutamiento avanzado, pero no se ocupa de la implementación real o ejecución, Eso recae en manos de servicios de 3ros. Es así que se dividen entre implementaciones conformantes, que pasaron las pruebas de calidad requeridas por kubernetes (entre las que se encuentra istio), semi-conformantes, que están en proceso de adaptarse a los requisitos de kubernetes y deprecadas.

Otra alternativa es utilizar el Gateway API propio de istio, que si bien es similar a la propuesta de Kubernetes posee algunas diferencias [23]:

- El Gateway API de istio se configura sobre un Deployment/Service ya creado. El de Kubernetes configura y crea un Gateway.
- En Istio todos los protocolos se configuran dentro del mismo recurso. En Kubernetes diferencia rutas HTTP y gRPC.
- El gateway de Kubernetes es constantemente trabajado para que sea compatible con todas las características que posee istio.

En definitiva se utilizó la primera opción, utilizar el Gateway API de Kubernetes con Istio como controlador, ya que istio indica como su intención es establecer el Gateway API de Kubernetes como la opción por defecto y además de poder hacer que los resultados se pueden abstraer más fácilmente respecto del controlador utilizado al ser la opción más general.

En relación al service mesh Istio utiliza Envoy como tecnología para implementar los sidecar. Envoy es un proxy tanto de capa 3 y 4, con diferentes capacidades para filtrar HTTP,TCP,UDP,TLS,etc. Como de capa 7, con capacidades para buffering, rate limiting, routing/forwarding,etc. orientado para arquitecturas orientada a servicios [24]. Pero es importante hacer una aclaración de los distintos tipos de perfiles que trae istio. Por un lado está el perfil por defecto que inyecta a cada uno de los pods de aplicación indicados un sidecar de envoy (para el trabajo se utilizó el perfil minimal ya que el perfil por defecto también crea un gateway generico sin configurar, pero en otros aspectos son idénticos) y un perfil ambiente en donde en lugar de utilizar un proxy por pod se utiliza un proxy por nodo llamado Ztunnel que utiliza el protocolo HBONE para sus comunicaciones. Si bien el objetivo del trabajo no gira en torno a la comparación de las tecnologías propuesta por Istio (ya existen múltiples trabajos que ahondan esos aspectos) se consideró que podría ser interesante utilizar ambas alternativas en la implementación con los que evaluar y comparar resultados en relación a las implementaciones sidecar.
Aclaración: Istio no inyecta un sidecar al gateway.

Implementación

La implementación realizada para el trabajo consiste en un cluster con 4 nodos, 1 maestro sobre el que corren los servicios propios de k8s y 3 worker sobre el que corren las aplicaciones desarrolladas para las pruebas.
El siguiente esquema muestra la estructura del sistema desarrollado y la comunicación de sus componentes:

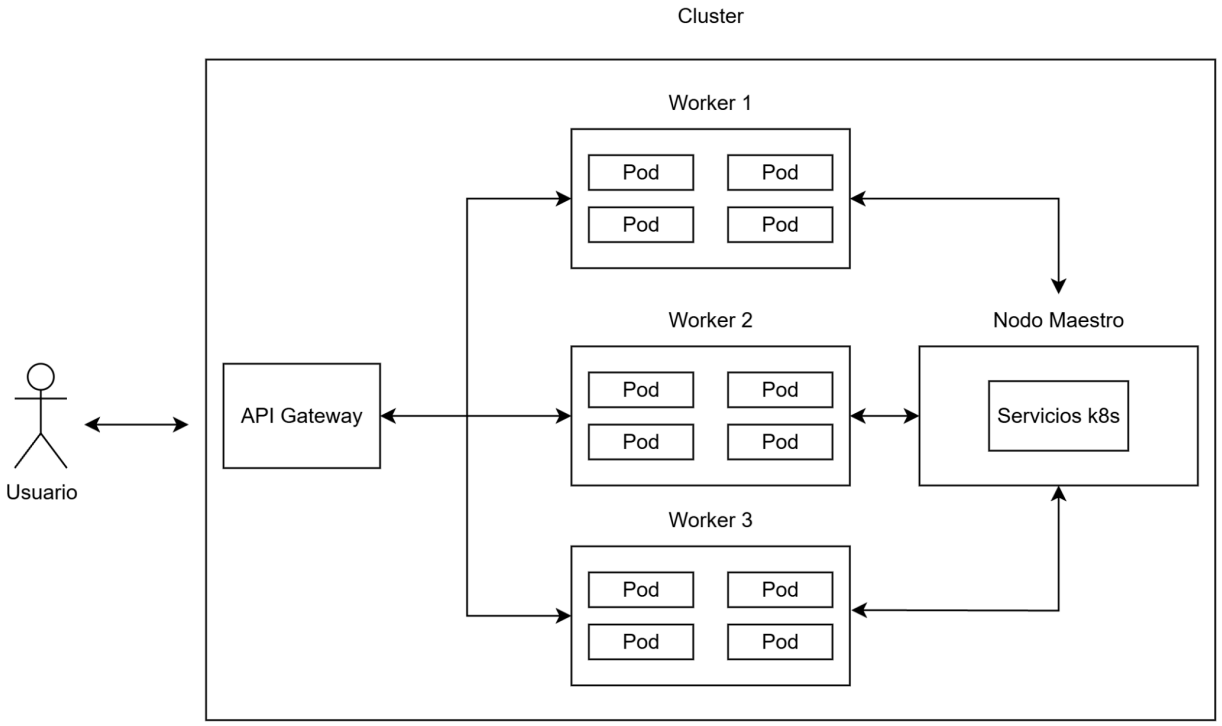


Figura 6: Representación esquemática de la estructura del sistema desarrollado.

Se desarrollaron 3 aplicaciones que se desplegaron sobre el sistema. Primero una aplicación “CPU-bench” que busca poner en estrés la CPU realizando una operación de multiplicación de matrices de órdenes elevadas. Segundo una aplicación “Mem-bench” que pone en estrés el uso de memoria al correr un script en C que aloje un bloque de gran tamaño en memoria para luego liberarlo. Y finalmente una aplicación “Front” que sea las que reciba las peticiones y las redirige a los otros 2 servicios y entrega los resultados al cliente. Para poder utilizarlas dentro del cluster kind provee un comando para cargarlas dentro de todos los nodos y los mismos puedan crear copias, se optó esta opción por su sencillez y debido a que la diferencia entre utilizar un registro local recae en los tiempos de espera a la hora de crear y destruir pods de forma dinámica, característica que no fue parte de esta evaluación.

Tras algunas pruebas iniciales se determinó que el punto límite para este sistema pequeño era de aproximadamente 3 réplicas por aplicación, distribuidas entre los nodos worker. Los resultados obtenidos se deberían de poder extrapolar con buenos resultados a sistemas pequeños (entre 1 y 5 nodos, con decenas de servicios) y posiblemente algunos sistemas medianos (cercanos a los cientos), siempre considerando como la gran cantidad de variables en juego para la configuración del sistema pueden causar grandes diferencia incluso en el mismo sistema. Para el caso de sistemas de gran escala (cientos a miles de servicios en simultáneo) es posible que estos resultados no sean fácilmente aplicables, ya que para mantener la eficiencia de sistemas a esa escala se necesita de una configuración mucha más compleja y monitoreo estricto, que no se asemeja a la implementación realizada en este trabajo.

A continuación se muestra una representación gráfica del flujo de datos dentro del sistema, el gráfico fue tomado desde el servicio de monitoreo Kiali.

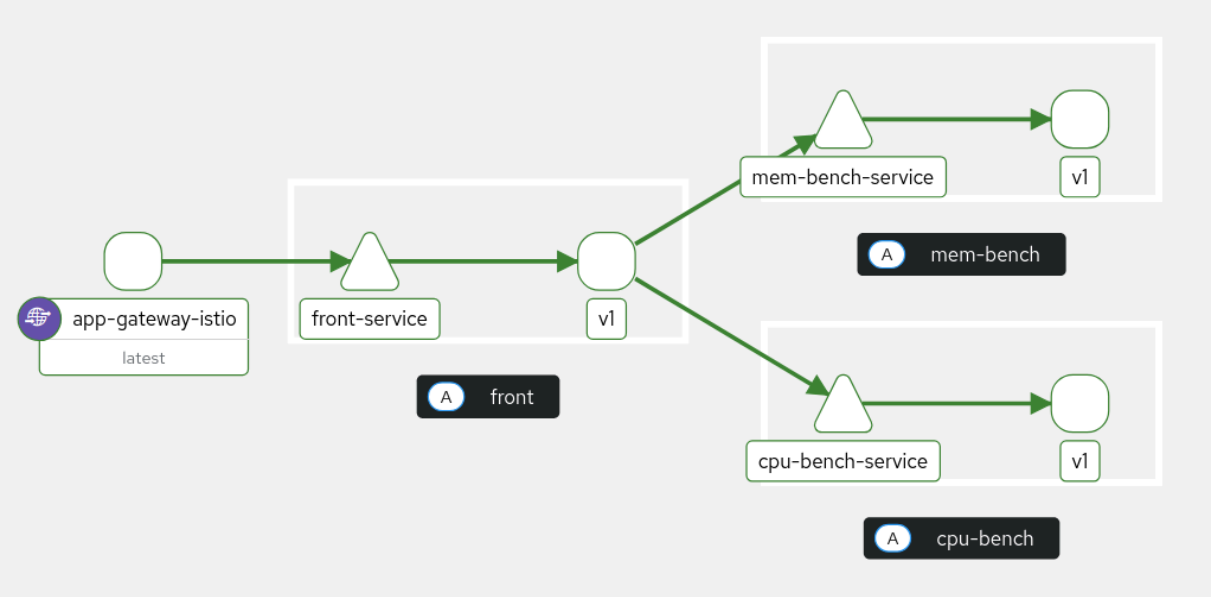


Figura 7: Flujo de ejecución del sistema desarrollado, tomado desde el servicio de monitoreo Kiali.

Para implementar los mecanismos de seguridad, Istio soporta de forma nativa mTLS y JWT. Por defecto mTLS funciona de forma permisiva, lo que significa que toda comunicación de ser posible va a utilizar mTLS, en caso contrario se realizan en texto plano. Para JWT solo se debe configurar en cada componente una autoridad y tipo de token al cual revisar. Mientras que WAF se debió implementar como un plugin, el WAF en particular utilizado fue Coraza al ser compatible con Istio.

Una vez desplegado el sistema se realizan las configuraciones correspondientes a cada uno de los servicios y de cada protocolo según los archivos .yaml, y sobre cada posible configuración se ejecutó una prueba. Dichas pruebas consistieron en generar peticiones HTTP/S utilizando autocannon, con una carga de 10000 peticiones, 100 concurrentes, un total de 20 veces por prueba. Después de cada prueba se utilizó el servicio de monitoreo Prometheus para obtener las métricas de consumo de CPU y Memoria del sistema.

Aclaraciones:

- Cuando se hace referencia a mTLS en los sidecar implica comunicación mTLS entre todos los pods, incluido las comunicaciones del gateway hacia el lado del sistema. La implementaciones de mTLS en el gateway se debieron implementar como tls simple, a través de HTTPS, para las comunicaciones hacia el lado del cliente.
- La implementación de JWT no realiza rotación de tokens, se utilizó un único token de prueba que todos los servicios verifican.
- Se utilizó coraza para el WAF implementado, el mismo fue configurado en modo de sólo detección, ya que de otra forma bloquea todas las pruebas realizadas al considerarlas un ataque al sistema.

Todo el código fuente del trabajo se puede encontrar en: https://github.com/Sakarrex/Microservices-security-evaluation

Resultados

En una primera instancia del trabajo se optó por realizar las pruebas en una máquina virtual con ubuntu, pero debido a la posibilidad de que el hardware de la pc física que realizó de host fuera insuficiente para las pruebas en segunda instancia se optó por correr las pruebas en un una pc de Elastic Compute Cloud (EC2) t3.2xlarge de aws. Se añaden los resultados de ambas pruebas para realizar comparaciones.

Especificaciones:

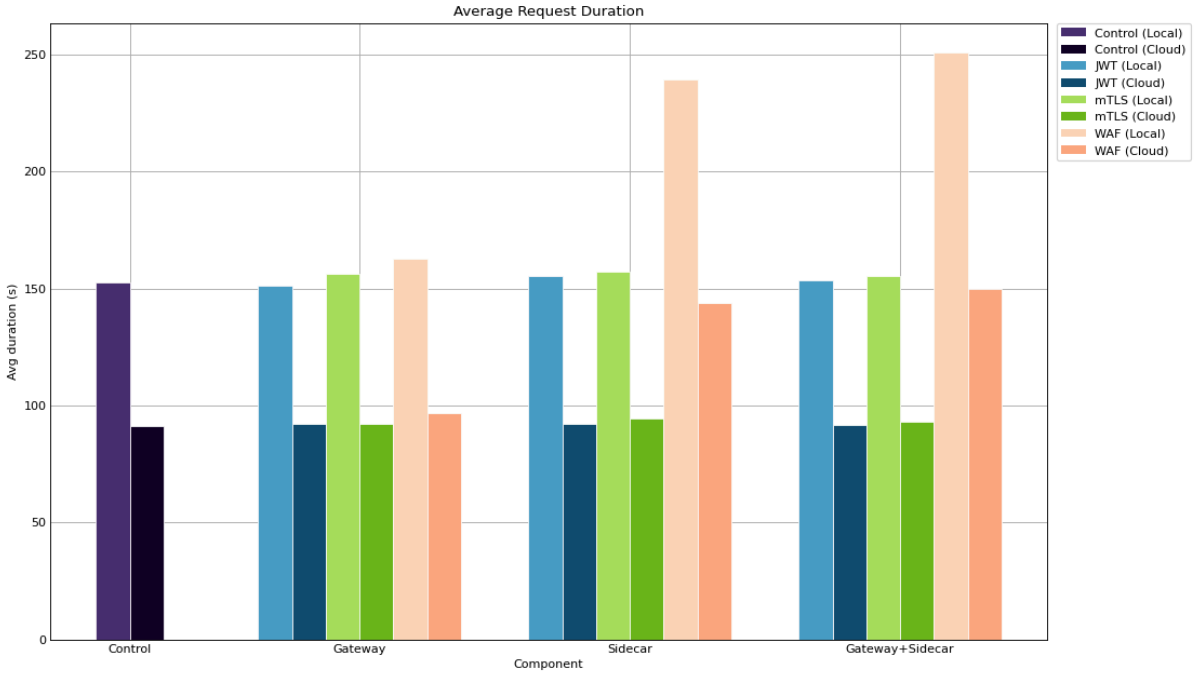
- **Máquina virtual:**
    - **OS:** Ubuntu 24.04.3
    - **Núcleos Virtuales:** 4
    - **Memoria:** 13 GB
- **Aws t3.2xlarge:**
    - **OS:** Ubuntu 24.04.3
    - **Núcleos Virtuales:** 8
    - **Memoria:** 32 GB

En los siguientes gráficos se muestran los resultados obtenidos de las evaluaciones. En cada gráfica se encuentran los 3 mecanismos de seguridad (mTLS, JWT, WAF) divididos en grupos: Grupo de control, grupo implementación en gateway, grupo implementación en sidecar y grupo implementación gateway+sidecar.
Las imágenes (a) muestran la duración en segundos de los experimentos, (b) el consumo de CPU en núcleos y (c) el consumo de memoria en MB.

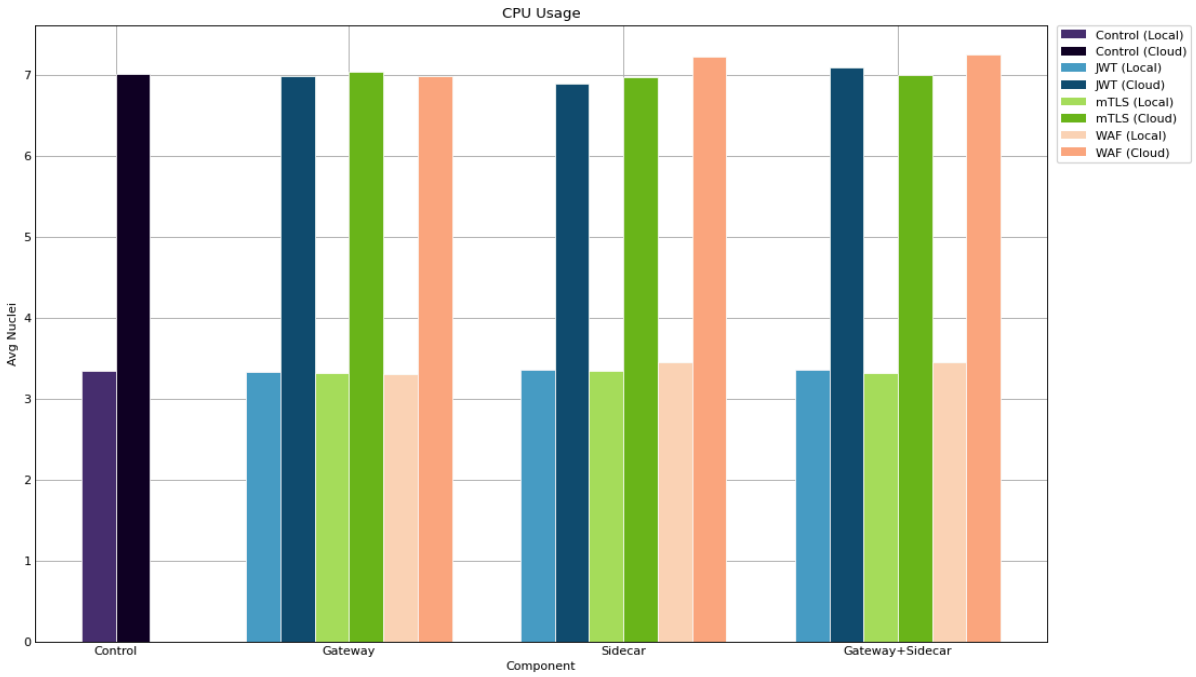
Cada valor está emparejado con los resultados obtenidos localmente (color claro) y en cloud (color oscuro).

Finalmente se presentan 2 pares de resultados, uno obtenido corriendo el service mesh con sidecars y por otro lado el obtenido corriendo el service mesh sidecar-less.

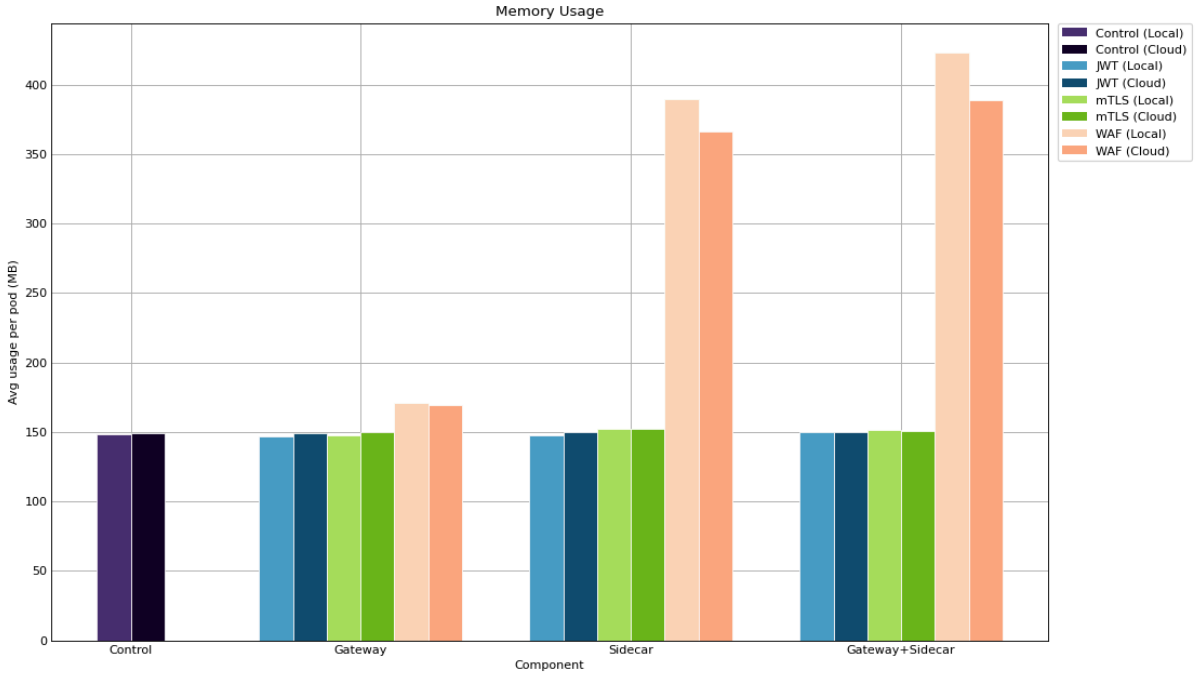
Istio modo minimal (sidecar)



(a) Duración



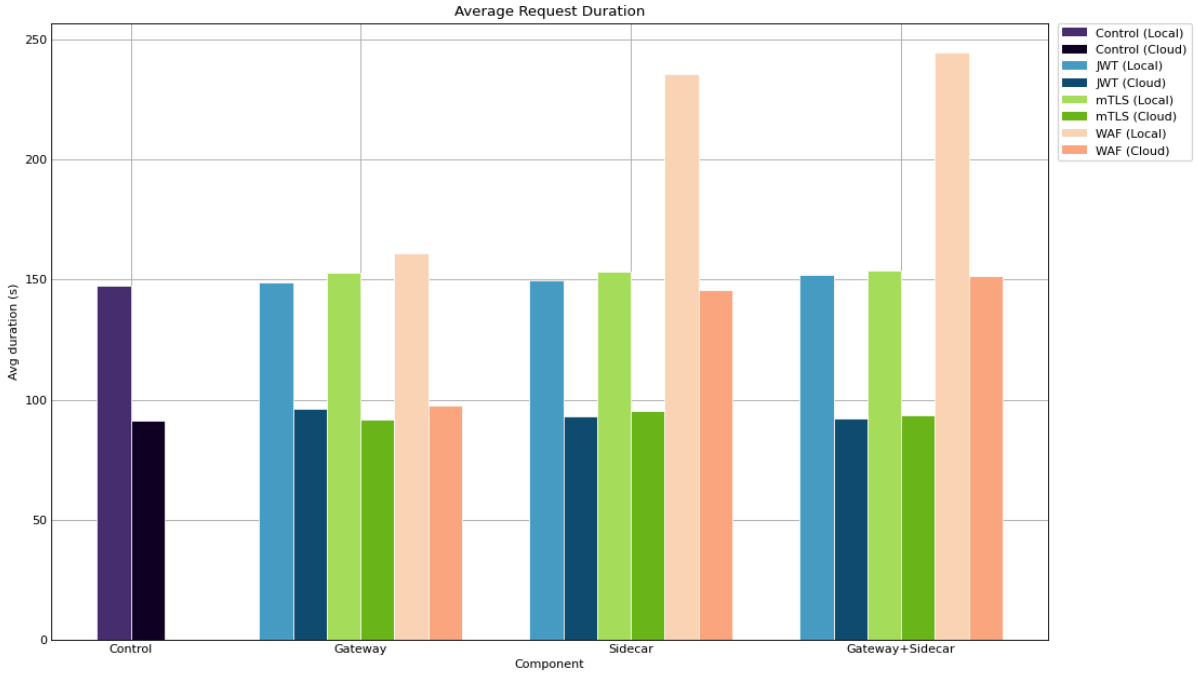
(b) CPU



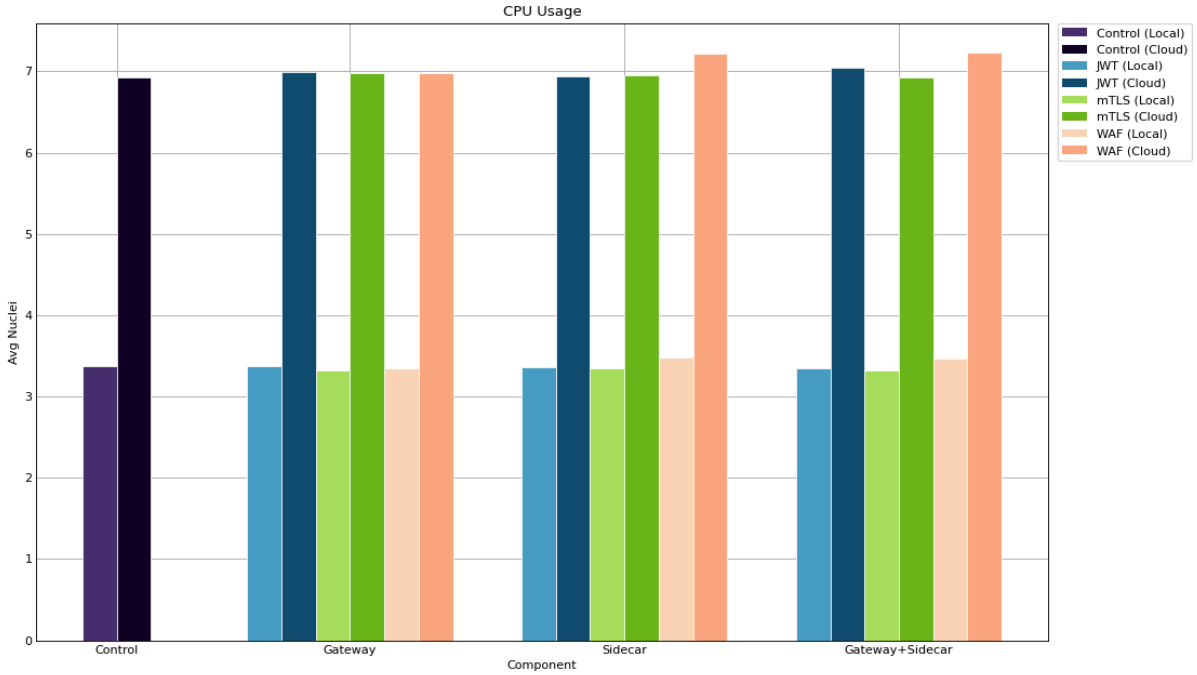
(c) Memoria

Figura 8: Resultados Istio modo minimal (sidecar)

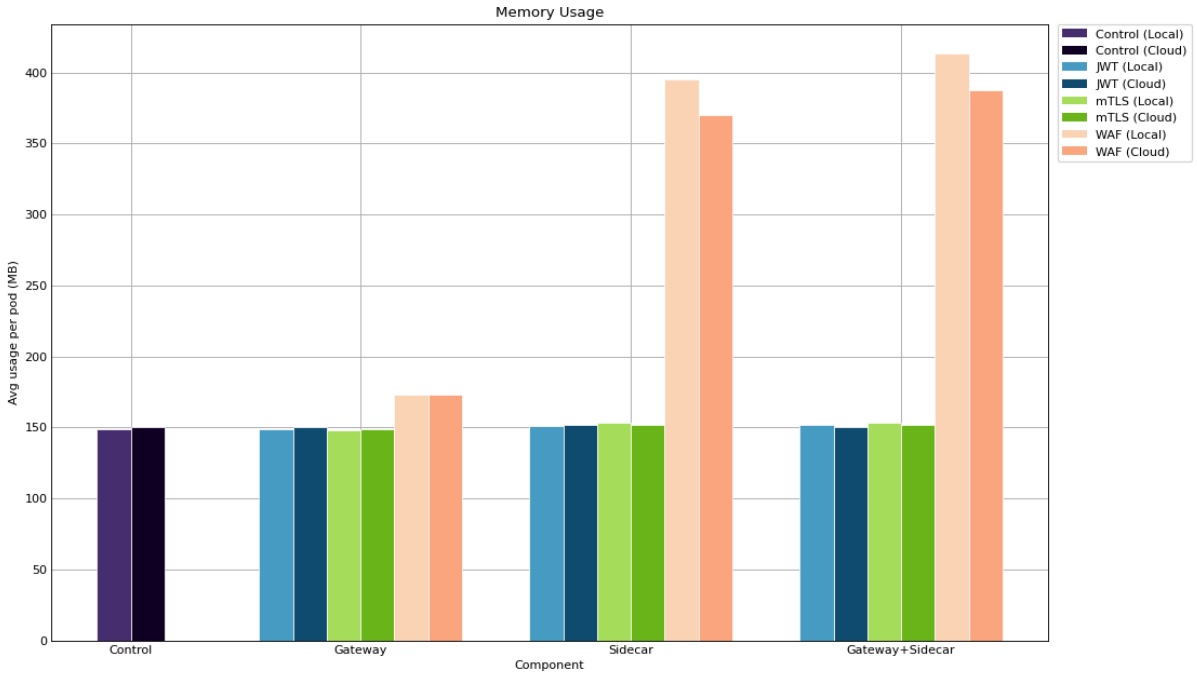
Istio modo ambiente (sidecar-less)



(a) Duración



(b) CPU



(c) Memoria

Figura 9: Resultados Istio modo ambiente (sidecar-less)

De los resultados se puede observar lo siguiente:

- A pesar de las diferencias en la potencia de hardware se puede comprobar un comportamiento similar tanto para los resultados obtenidos localmente y en cloud.

- Hay algunos valores en los cuales la implementación sidecar-less fueron ligeramente menores respecto a la implementación sidecar, los mismos no son significativos y no se puede apreciar una diferencia notable tanto en latencia, CPU y memoria entre ambos. Por lo tanto es posible que para observar una mayor distinción en los resultados se deberían realizar las pruebas en un sistema de mayor tamaño, en donde la cantidad de sidecars inyectados entre ambos modelos sea una diferencia significativa.

- En relación a la latencia del sistema, si bien los resultados obtenidos en cloud fueron mucho más veloces, gracias a la mayor potencia de hardware, se mantiene una proporcionalidad similar entre los mismos. Las implementaciones de JWT oscilaron alrededor de 0,3% y 1%, respecto al grupo de control, mientras que mTLS osciló entre un 2% y 3% en la mayoría de los casos. Ambos tuvieron resultados cercanos o menores al 3% respecto a las pruebas de control, con un caso inclusive siendo ligeramente menor a los tiempos de control, indicando que no tiene un impacto significativo en los tiempos de respuesta.
  Pero, por otro lado, la implementación WAF añadió un retraso en los tiempos de respuesta, uno más ligero en el gateway y otro masivo para sidecar y gateway+sidecar.

  En la siguientes tablas se colocan los incremento en porcentaje de la latencia de cada implementación en relación al grupo de control, para facilitar la comparación.

JWT

Entorno	Local	Cloud
Incremento Gateway	-0.85%	0.99%
Incremento Sidecar	1.70%	0.87%
Incremento Gateway+Sidecar	0.39%	0.22%

Tabla 1: Comparación del incremento en latencia en implementaciones JWT.

mTLS

Entorno	Local	Cloud
Incremento Gateway	2.40%	0.65%
Incremento Sidecar	2.95%	3.58%
Incremento Gateway+Sidecar	1.88%	1.75%

Tabla 2: Comparación del incremento en latencia en implementaciones mTLS.

WAF

Entorno	Local	Cloud
Incremento Gateway	6.62%	6,01%
Incremento Sidecar	56.8%	57,33%
Incremento Gateway+Sidecar	64.34%	64.12%

Tabla 3: Comparación del incremento en latencia en implementaciones WAF.

- Respecto al uso de CPU tanto en cloud como local se utilizó aproximadamente 87,5% de la capacidad de cómputo en todos los casos, donde el 12,5% restante posiblemente estuviera dedicado para el SO. Esto puede significar que el sistema necesita aún más capacidad de cómputo para su funcionamiento o que no que no posee un límite establecido para el uso de CPU por lo que siempre va a intentar utilizar la mayor cantidad posible.

- Finalmente, por parte del consumo de memoria, en todos los casos evaluados tanto el grupo de control, JWT y mTLS presentaron uso similar, alrededor de 150 MB por pod, con varios casos inclusive siendo ligeramente menor. Pero las implementaciones WAF se puede observar que incrementan la memoria utilizada, de forma similar a la latencia. En menor medida solo en gateway, pero con uso masivo en sidecar y gateway+sidecar, aunque curiosamente para los casos de sidecar y gateway+sidecar en cloud el consumo es ligeramente menor.

  En las tablas siguientes se encuentran el consumo de memoria de las implementaciones de cada mecanismo en relación al grupo de control en porcentajes.

JWT

Entorno	Local	Cloud
Incremento Gateway	-0.91%	-0.42%
Incremento Sidecar	-0.38%	0.20%
Incremento Gateway+Sidecar	0.96%	0.44%

Tabla 4: Comparación incremento en memoria por pod de implementaciones JWT

mTLS

Entorno	Local	Cloud
Incremento Gateway	-0.56%	0.22%
Incremento Sidecar	2.72%	1.59%
Incremento Gateway+Sidecar	2.18%	0.94%

Tabla 5: Comparación incremento en memoria por pod de implementaciones mTLS

WAF

Entorno	Local	Cloud
Incremento Gateway	14.88%	13.44%
Incremento Sidecar	162.49%	144.93%
Incremento Gateway+Sidecar	185.05%	159.98%

Tabla 6: Comparación incremento en memoria por pod de implementaciones WAF.

Conclusiones

En conclusión, según los resultados obtenidos, el mayor impacto para el funcionamiento del sistema son la capacidad de cómputo y la decisión sobre si va a implementar WAF como medida de seguridad, ya que estos fueron los factores que mayor influencia tuvieron en los tiempos de respuesta.

Esto se debe a que por un lado no se pudo establecer un techo real del consumo de CPU, lo cual en sistemas reales implica una necesidad de monitoreo constante y posiblemente establecer límites de consumo, ya que en el peor de los casos un uso desmedido puede afectar al funcionamiento del SO y por extensión del sistema de microservicios.

Por otro lado, implementar una solución WAF implica añadir a un pod toda la lógica extra de un firewall, con todas sus reglas, lo cual si se implementan sólo en el gateway no implica gran diferencia, pero si se deben implementar en cada uno de los pods que corren dentro del sistema como sidecar, supone un costo de recursos importante.

Por lo tanto, los mecanismos de JWT y mTLS deberían ser implementados siempre que resulte posible, mientras que WAF solo debería aplicarse al gateway a unos pocos servicios que sean de vital importancia mantener protegidos, mientras que también se debe de mantener un estricto monitoreo del uso de CPU en todo momento.

Amenazas a la validez:

Si bien en este trabajo se buscó dar unas pautas generales a tener en cuenta sobre la configuración y el consumo de recursos de sistemas de microservicios, durante la decisión de implementar un service mesh un desarrollador encuentra muchas alternativas que decidir. Desde las más generales como el servicio proveedor (Istio, Ciluim, Linkerd, etc), hasta configuraciones específicas del comportamiento de cada implementación, lo cual vuelve muy complejo poder medir el funcionamiento del sistema en cada una de las implementaciones posibles. Inclusive considerando como cada sistema tiene sus propios requisitos una misma configuración que resulte eficiente en uno, genere problemas en el rendimiento de otro. Es por esto que realizar pruebas en cada sistema y tener un monitoreo robusto de su funcionamiento es de gran importancia.

Trabajo a futuro

Como extensiones para el presente trabajo se propone lo siguiente:

- Realizar pruebas sobre sistemas de mayor escala.
- Comprobar si los resultados se mantienen con otras tecnologías.
- Implementar autenticación JWT que se adapte a tiempo real

Bibliografía

[1] Mordor Intelligence, «Cloud Microservices Market - Share, Trends & Size». Accedido: 25 de abril de 2026. [En línea]. Disponible en: https://www.mordorintelligence.com/industry-reports/cloud-microservices-market

[2] M. R. Saleh Sedghpour, C. Klein, y J. Tordsson, «An Empirical Study of Service Mesh Traffic Management Policies for Microservices», en *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, Beijing China: ACM, abr. 2022, pp. 17-27. doi: 10.1145/3489525.3511686.

[3] M. Matias, E. Ferreira, N. Mateus-Coelho, O. Ribeiro, y L. Ferreira, «Evaluating Effectiveness and Security in Microservices Architecture», *Procedia Comput. Sci.*, vol. 237, pp. 626-636, ene. 2024, doi: 10.1016/j.procs.2024.05.148.

[4] Y. Dawei, G. Yang, H. Wei, y L. Kai, «Design and Achievement of Security Mechanism of API Gateway Platform Based on Microservice Architecture», *J. Phys. Conf. Ser.*, vol. 1738, n.º 1, p. 012046, ene. 2021, doi: 10.1088/1742-6596/1738/1/012046.

[5] M. R. S. Sedghpour, C. Klein, y J. Tordsson, «Service mesh circuit breaker: From panic button to performance management tool», en *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, Online United Kingdom: ACM, abr. 2021, pp. 4-10. doi: 10.1145/3447851.3458740.

[6] X. Zhu *et al.*, «Dissecting Overheads of Service Mesh Sidecars», en *Proceedings of the 2023 ACM Symposium on Cloud Computing*, Santa Cruz CA USA: ACM, oct. 2023, pp. 142-157. doi: 10.1145/3620678.3624652.

[7] Y. Elkhatib y J. P. Poyato, «An Evaluation of Service Mesh Frameworks for Edge Systems», en *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, Rome Italy: ACM, may 2023, pp. 19-24. doi: 10.1145/3578354.3592867.

[8] A. B. Barr, O. Lavi, Y. Naor, S. Rampal, y J. Tavori, «Technical Report: Performance Comparison of Service Mesh Frameworks: the MTLS Test Case», 4 de noviembre de 2024, *arXiv*: arXiv:2411.02267. doi: 10.48550/arXiv.2411.02267.

[9] Cloud Native Computing Foundation (CNCF), «Service Meshes Are on the Rise — But Greater Understanding and Experience Are Required», Cloud Native Computing Foundation (CNCF), MicroSurvey / Report, 2022. [En línea]. Disponible en: https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf

[10] Y. Abgaz *et al.*, «Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review», *IEEE Trans. Softw. Eng.*, vol. 49, n.º 8, pp. 4213-4242, ago. 2023, doi: 10.1109/TSE.2023.3287297.

[11] R. Chandramouli, «Security strategies for microservices-based application systems», National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-204, ago. 2019. doi: 10.6028/NIST.SP.800-204.

[12] M. Souppaya, J. Morello, y K. Scarfone, «Application container security guide», National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-190, sep. 2017. doi: 10.6028/NIST.SP.800-190.

[13] W. K. A. N. Dias y P. Siriwardena, *Microservices Security in Action*. Simon and Schuster, 2020.

[14] M. G. De Almeida y E. D. Canedo, «Authentication and Authorization in Microservices Architecture: A Systematic Literature Review», *Appl. Sci.*, vol. 12, n.º 6, p. 3023, mar. 2022, doi: 10.3390/app12063023.
[15] Esperanza Erika Bonillo Valero, «Securización de microservicios orquestados con Kubernetes», Thesis, Universitat Oberta de Catalunya, España, 2024. [En línea]. Disponible en: https://hdl.handle.net/10609/149723
[16] Kubernetes, «Kubernetes documentation | Overview», Overview. Accedido: 7 de mayo de 2026. [En línea]. Disponible en: https://kubernetes.io/docs/concepts/overview/
[17] S. Gadge y V. Kotwani, «Microservice Architecture: API Gateway Considerations», GlobalLogic, Inc., White Paper, 2017. [En línea]. Disponible en: https://mainlab.cs.ccu.edu.tw/presentation/pdf/(2017)Microservice-Architecture-API-Gateway-Considerations.pdf
[18] A. Neelan, «The Evolving Role of API Gateways in Scalable Microservices Architecture», *Int. J. Emerg. Trends Comput. Sci. Inf. Technol.*, vol. 6, n.º 4, oct. 2025, doi: 10.63282/3050-9246.IJETCSIT-V6I4P102.
[19] I. McPhee, «Sidecarless Service Meshes: Are They Ready for Prime Time?», Sidecarless Service Meshes: Are They Ready for Prime Time? Accedido: 17 de abril de 2026. [En línea]. Disponible en: https://portal.gigaom.com/blog/sidecarless-service-meshes-are-they-ready-for-prime-time
[20] R. Chandramouli y Z. Butcher, «Building secure microservices-based applications using service-mesh architecture», National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-204A, may 2020. doi: 10.6028/NIST.SP.800-204a.
[21] IBM, «Security concepts and mechanisms». Accedido: 17 de abril de 2026. [En línea]. Disponible en: https://www.ibm.com/docs/en/ibm-mq/9.2.x?topic=overview-security-concepts-mechanisms
[22] Kubernetes, «Gateway API | Kubernetes», Gateway API. Accedido: 7 de mayo de 2026. [En línea]. Disponible en: https://kubernetes.io/docs/concepts/services-networking/gateway/
[23] Istio, «Istio / Kubernetes Gateway API», Istio / Kubernetes Gateway API. Accedido: 7 de mayo de 2026. [En línea]. Disponible en: https://istio.io/latest/docs/tasks/traffic-management/ingress/gateway-api/
[24] Envoy, «What is Envoy — envoy tag-v1.38.0 documentation», What is Envoy. Accedido: 7 de mayo de 2026. [En línea]. Disponible en: https://www.envoyproxy.io/docs/envoy/v1.38.0/intro/what_is_envoy