UNIVERSIDAD DE ANTIOQUIA

Aplicación de una arquitectura basada en Service Mesh para una plataforma cognitiva utilizando Kubernetes e Istio

Autor(es)

Daniel Santa Rendón

Universidad de Antioquia.

Facultad de ingeniería, Departamento de sistemas.

Medellín, Colombia

2021

Aplicación de una arquitectura basada en Service mesh para una plataforma cognitiva utilizando Kubernetes e Istio.

Daniel Santa Rendón

Tesis o trabajo de investigación presentada(o) como requisito parcial para optar al título de:
Ingeniero de sistemas

Asesores:
Ph. D. Leonardo Augusto Pachón Contreras
Ing. Martín Elías Quintero

Universidad de Antioquia
Facultad de ingeniería, Departamento de sistemas
Medellín, Colombia
2021.

Tabla de Contenidos

Resumen

La concurrencia en el desarrollo, la falta de elasticidad para adaptarse a una alta demanda de peticiones y la complejidad del mantenimiento son algunos de los problemas de los sistemas monolíticos que las aplicaciones distribuidas intentan resolver. La solución a estos problemas trae consigo nuevos desafíos como el manejo de errores, tiempos de respuesta, capacidad de monitoreo y métricas del sistema. En este reporte se estudia y se aplica el despliegue y configuración de una aplicación cognitiva basada en microservicios implementando Kubernetes conjunto con Istio. De esta manera se espera garantizar una alta disponibilidad, seguridad, rendimiento y un seguimiento riguroso en términos de monitoreo del sistema, proporcionando al producto altos estándares de calidad y mantenibilidad.

Dentro del alcance se propone un ambiente de desarrollo local que proporcione un entorno reproducible utilizando bajos recursos de hardware. Allí se realizará el despliegue de la aplicación configurando e instalando Istio como primer acercamiento práctico a las mallas de servicio. Posteriormente, se exploran los servicios de la nube por medio de un proveedor cloud (GCP en este caso) y finalmente se pondrá la solución a disposición de los clientes en un administrador de Kubernetes enteramente en producción.

Introducción

En la actualidad, las arquitecturas tradicionales no son suficientes, en algunos casos, para cubrir por completo las necesidades que el desarrollo moderno de software requiere[1]. En particular, la dificultad para escalar y mantener una aplicación construida bajo las arquitecturas tradicionales ha llevado a explorar nuevas estrategias que permitan superar las desventajas de los monolitos.[2]

En comparación con la arquitectura de monolitos tradicionales, una arquitectura basada en microservicios proporciona una mejor escalabilidad, fácil soporte en términos de funcionalidades, reusabilidad, portabilidad, entre otras [3]. Sin embargo, ésta presenta nuevos desafíos como la comunicación, despliegues, políticas de reintento y control de errores. Por esta razón, es necesario incluir tecnologías que faciliten el proceso de desarrollo, gestión y administración de los microservicios.

Para estandarizar los ambientes de desarrollo se ha adoptado el uso de contenedores, principalmente contenedores Docker, que se definen como unidades ejecutable de software que contienen el código de la aplicación, junto con sus bibliotecas y dependencias [4]. En esa línea de ideas, Kubernetes se usa ampliamente como orquestador de todos estos contenedores sin interferir con ninguna implementación a nivel de aplicación, es decir, facilita la gestión y administración del despliegue de los contenedores a nivel de infraestructura y hardware [5].

Por último, se estudia la implementación de una malla de servicios, en concreto se utilizó Istio. El objetivo principal es añadir una capa adicional de software versátil a cualquier arquitectura distribuida que facilite la monitorización y el control de las comunicaciones internas de las aplicaciones. Istio gestiona la comunicación, la generación de métricas, el seguimiento de errores y las políticas de reintento de los microservicios que componen la aplicación, añadiendo otro nivel de seguridad y fiabilidad al sistema [6].

Objetivos

Objetivo Principal

Implementar una arquitectura basada en mallas de servicio utilizando *Istio* para soportar la orquestación de los microservicios de una plataforma cognitiva.

Objetivo Específicos

1. Construir el diseño de una malla de servicios enfocado en la alta disponibilidad y supervisión de cada artefacto de software.
2. Desarrollar servicios de software para soportar los microservicios de la plataforma mediante reglas de tráfico y monitoreo.
3. Ofuscar el diseño e implementación final con el objetivo de plantear métricas en términos de rendimiento que permitan establecer la viabilidad de los sistemas basados en mallas de servicio frente al despliegue actual de la plataforma.

Marco teórico

Las arquitecturas monolíticas (AM) son el camino tradicional para el desarrollo de software y han sido implementadas ampliamente por grandes empresas como Amazon, Netflix o eBay principalmente porque son sencillas de desarrollar, de testear y de desplegar [7]. Sin embargo, a medida que los proyectos avanzan y el código base empieza a ser más robusto, las AM generan un impacto negativo en la sostenibilidad, mantenimiento y soporte de las aplicaciones. Algunas desventajas de las AM son (i) la dificultad de tener un sistema que soporte cargas de trabajo dinámicas, (ii) conflictos de código debido a que todo el equipo trabaja sobre un mismo proyecto, (iii) el alto acoplamiento tecnológico y los contenedores sobrecargados que afectan a la productividad de los desarrolladores, además de (iv) lo poco efectivo que resultan ser los despliegues e integraciones. [8]

Desde otro ángulo, la arquitectura de microservicios (AMS) trata de solucionar los problemas de las AM separando la aplicación en pequeñas composiciones bien definidas según su dominio [9]. En la AMS la comunicación entre servicios se realiza mediante un mecanismo ligero como HTTP usando los principios RESTFul [10]. Al ser atómicas, las AMS ofrecen fácil mantenimiento, reusabilidad, escalabilidad, disponibilidad, facilidad en los despliegues automatizados, entre otras; estas características hacen de la estrategia de de microservicios un candidato excelente cuando el desarrollo aumentar su complejidad estructural [11]

Las AMS enfrentan sus propios desafíos, uno de los más problemáticos es la gran cantidad de ambientes, lenguajes, dependencias, módulos y requisitos que pueda tener cada uno de los servicios que conforman la aplicación. Para mitigar esta situación, se han considerado innumerables veces el uso de contenedores, pues estos "ofrecen un mecanismo de empaquetado lógico en el que las aplicaciones se pueden abstraer del entorno en el que se ejecutan" [12] Además, a diferencia de los aislamientos utilizando hipervisores, los contenedores virtualizan los recursos del sistema operativo haciéndolos unidades mucho más livianas y totalmente aisladas, ideal para el propósito de la aplicación (*ver Fig. 1*). Una de las herramientas de contenedores que más se ha popularizado en las comunidades de desarrollo y que es aceptada por las grandes desarrolladoras de software es Docker.

Docker es la tecnología de contenedores más popular y el estándar de facto para crear y compartir aplicaciones en contenedores, desde el escritorio hasta la nube [14]. Sin embargo, cuando el número de contenedores crece, la tarea de gestionarlos y administrarlos es compleja debido a la intrincada comunicación entre todos los servicios. Por esta razón, se hace necesario automatizar las tareas de orquestación y así lograr mantener todos los artefactos de software disponibles. Para lograr esta tarea, se utiliza software como Kubernetes que "es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios"[15].

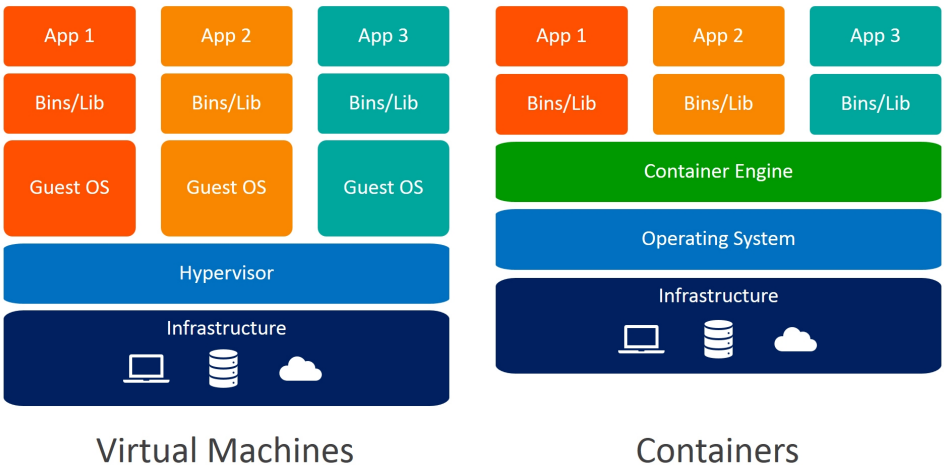

Figura 1. Máquinas virtuales vs. contenedores [13]

Kubernetes, ofrece la capacidad de administrar con facilidad los contenedores y la infraestructura donde están desplegados. Esto lo hace por medio de artefactos software como los *configmaps* que permiten almacenar información no confidencial. Un *pod* es el objeto más básico de Kubernetes y representa una instancia única de un proceso en ejecución en el clúster y puede contener uno o muchos contenedores [17].

Los *pods* pueden consumir los *configmaps* como variables de ambiente [16]; sin embargo, al ser efímeros y descartables, es necesario usar un artefacto de nivel mayor llamado *service*.

Un *service* expone de manera abstracta una aplicación que se ejecuta en un conjunto de pods como un servicio de red [18]; de esta manera controlar el ciclo de vida de una aplicación, como las imágenes de los contenedores a utilizar por la aplicación, el número de réplicas que debe haber y la forma en que deben actualizarse, se utilizan los *deployments* [19].

Además, Kubernetes permite dividir lógicamente un cluster por medio de *namespaces que mejoran* la organización y separación de recursos del cluster, la seguridad e incluso el rendimiento [20]**.** En Kubernetes todos los artefactos son manejables por medio de archivos de configuración, lo cual lo vuelve flexible, replicable y mantenible.

A pesar de que Kubernetes es una solución para la gestión y el despliegue de los contenedores, una de sus desventajas es que agnóstico a las arquitecturas a nivel de aplicación que puedan implementarse. Es decir, si bien soluciona la mantenibilidad de cada contenedor, Kubernetes por sí mismo, no mejora la comunicación entre ellos. Una solución a esta falencia, puramente gestada desde el concepto de microservicios, es el uso de una malla de servicio o *service mesh,* que controla la forma en la que los servicios de la aplicación comparten datos entre sí. De manera que la malla de servicio facilita la optimización de las comunicaciones y previene el *downtime* cuando crece la aplicación [21]. En este proyecto se usará Istio, que mejora la seguridad, las métricas y políticas de reintento entre servicios [22].

Istio está compuesto principalmente por el *Control Plane* y *Data Plane:* el *Control Plane* son los *pods* que se ejecutan en el *namespace* de Istio (llamado istio-system) e implementan las funcionalidades de éste; la mayoría de utilidades se encapsulan en un solo pod llamada *istiod.* Istio también es responsable inyectar los *envoy proxies* [23] a los *namespaces* con la etiqueta *istio-injection: enabled.* Estos proxies o *sidecars* se insertan en los *pods* y a través de ellos se redirige todo el tráfico de la aplicación (*ver Fig. 2*). Istio permite integrar a terceros para ampliar sus funcionalidades, por ejemplo, existe Kiali [25], Grafana [26] y Prometheus [27] para la telemetría, y herramientas para el trazado de las peticiones como Jaeger [28] y Zipkin [29]. Esto se hace por medio de los proxies inyectados que monitorizan y realizan el seguimiento del rendimiento de la aplicación, estado de las peticiones y comportamiento del cluster.
Sumado a esto, Istio dispone de mecanismos para la gestión y control del tráfico tales como los *virtual services* [30] y los *destinations rules* [31] que proporcionan la habilidad de reconfigurar los proxies de forma dinámica para un manejo de tráfico inteligente. Estas herramientas no son obligatorias, solo se usan si son requeridas. Además, Istio proporciona automáticamente la comunicación entre pods para que sea segura, es decir, por medio de protocolos HTTPS [32] y TLS [33], esto se conoce como mutual TLS [34].

Por tanto, Istio con Kubernetes y Docker disponen de toda la capacidad de administración, automatización, control y seguimiento que una AMS requiere.

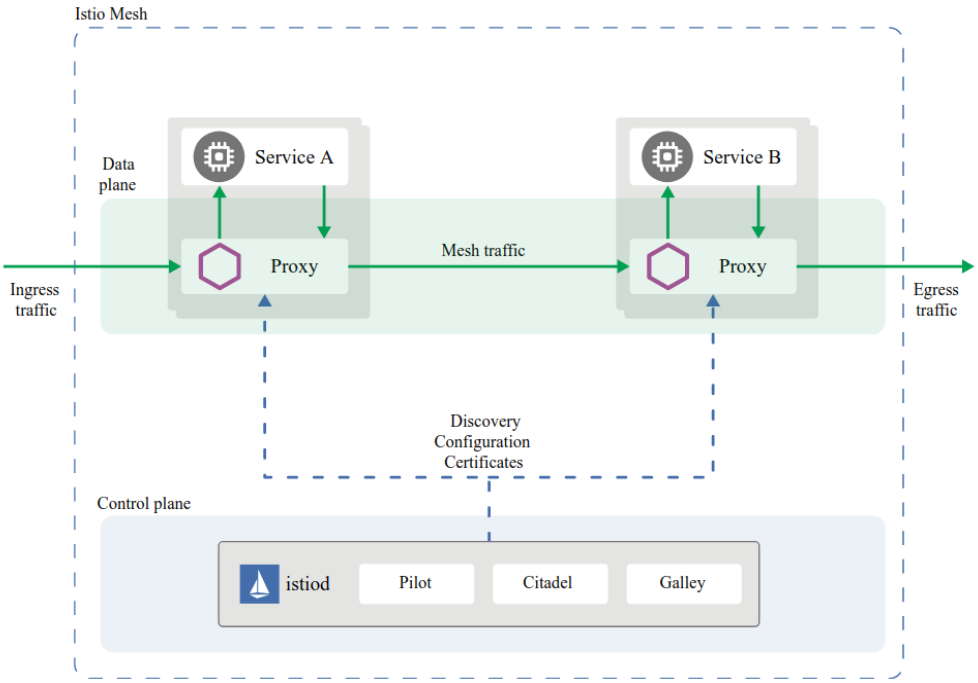

Figura 2. Arquitectura de Istio [24]

Metodología

La Fig. 3 muestra el diagrama de flujo que representa cada paso metodológico en etapas y técnicas empleadas para el desarrollo de la práctica empresarial.

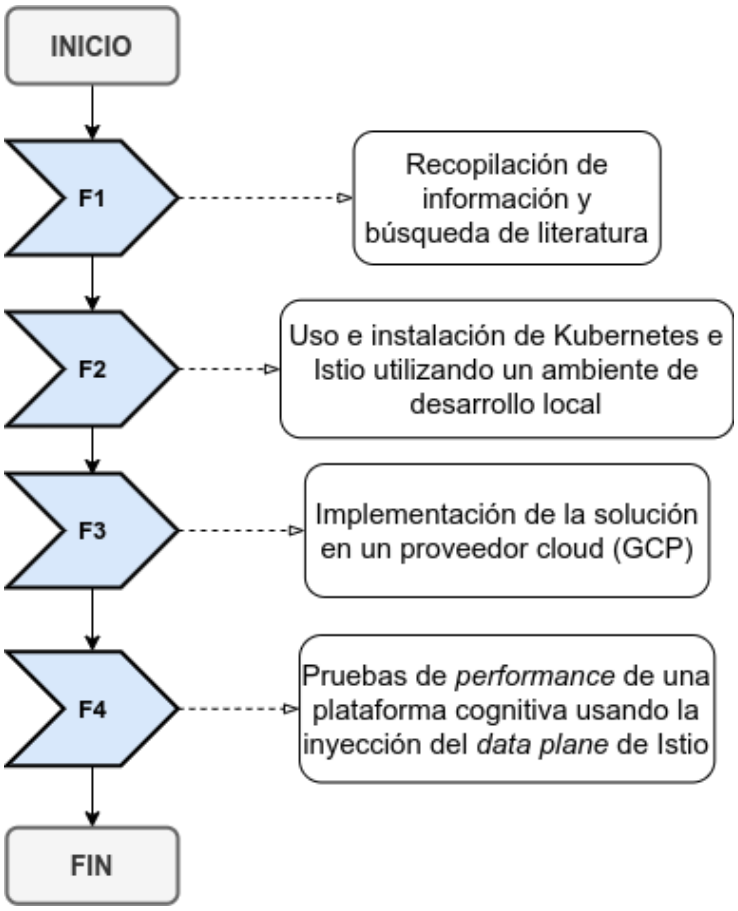

Figura 3. Diagrama de flujo de la metodología aplicada

F1. Recopilación de información y búsqueda de literatura

Los conceptos fundamentales de las tecnologías a usar para este proyecto se exploraron desde de numerosas fuentes de información [xx-xxx]. Inicialmente se estudió la definición de los contenedores, comenzando desde sus inicios hasta llegar a docker. Posteriormente, se indagó a profundidad sobre la viabilidad de herramientas agregadas a Docker para la gestión de la configuración. Con Docker Swarm se hizo el primer acercamiento a la administración y gestión de *clusters*. Después se estudiaron rigurosamente los conceptos y arquitectura de Kubernetes para entender la infraestructura necesaria para una aplicación

basada en microservicios en la nube Finalmente, se profundizó en la noción de Istio como *service mesh*, en donde se exploraron los conceptos de una malla de servicios, cuáles son sus beneficios, dificultades, desventajas e implementación.

F2. Uso e instalación de Kubernetes e Istio utilizando un ambiente de desarrollo local

Una vez se exploró la literatura y fueron claros los conceptos básicos sobre Kubernetes e Istio se procedió a la implementación de un *cluster* local utilizando Kind [35], un administrador de máquinas locales que se ejecutan como un cluster, allí se desplegó una aplicación orientada a microservicios sencilla (ver Fig 4). Para poner en marcha la aplicación, se construyeron las imágenes de los contenedores y se almacenan en un repositorio especializado para guardar dichas imágenes, un registrador; como se estaba trabajando en un ambiente de desarrollo, se usó un registrador local. Las aplicaciones se describen en Kubernetes de manera declarativa; por tanto, es necesario desarrollar archivos de configuración en extensión .yaml, en donde se definen los *pods*, los servicios, *deployments* y demás recursos de Kubernetes (ver *figs 5, 6 y 7*). La comunicación con Kubernetes ocurre mediante el uso de una interfaz de línea de comandos, Kubectl [36], que se encarga de enviar estas descripciones declarativas directamente a un artefacto de Kubernetes llamado Kube-Api es decir que toda la interacción con el *cluster* ocurre por medio de Kubectl. Luego de tener los archivos de configuración listos, se procedió a instalar el *Control Plane* de Istio. Por último, se aplicaron los manifiestos de Kubernetes para levantar todos los componentes de la aplicación; al hacer esto, se deben ejecutar dentro de cada *pod* un contenedor adicional, los cuales son los proxies que Istio se encargó de inyectar.

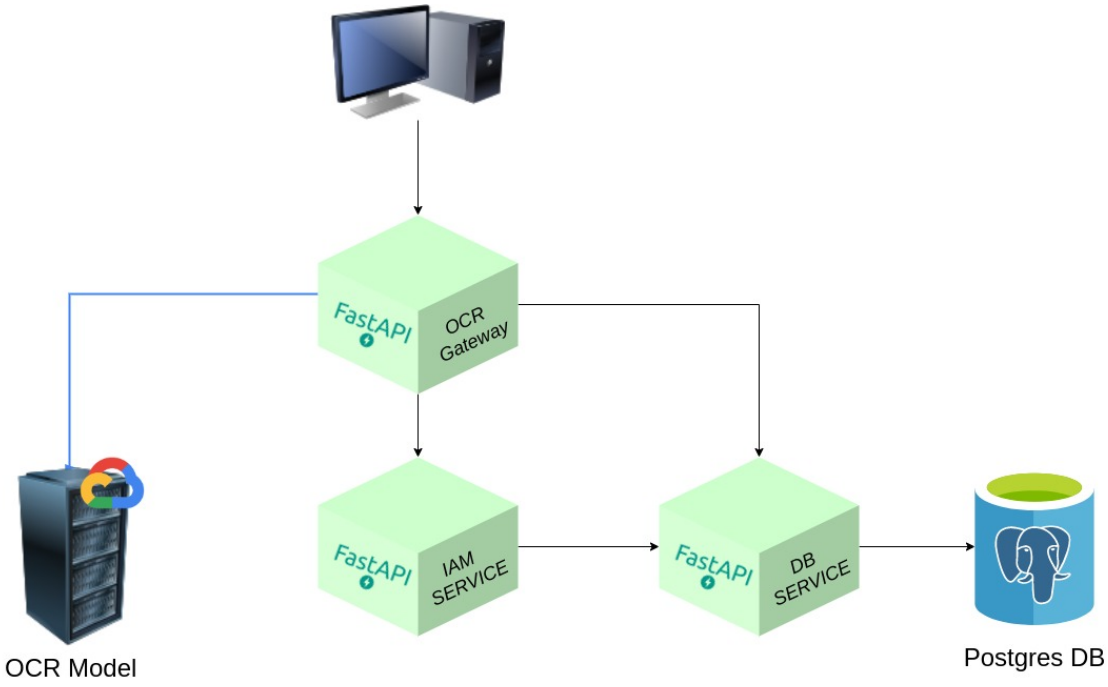

Figura 4. Arquitectura de la aplicación cognitiva a desplegar

```yaml
apiVersion: v1
kind: Service
metadata:
  name: guane-ocr-service
  namespace: default
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
  selector:
    role: guane-ocr-service
```

Figura 5. Definición de un servicio de Kubernetes

F3. Implementación de la solución en un proveedor cloud (GCP)

Una vez se realizó el despliegue y las pruebas en el ambiente local, se procedió a habilitar un cluster en Google Cloud [37]; de igual manera que en un ambiente local se instaló el *Control Plane* de Istio y se habilitó la inyección de los proxies. Luego de esto, se desplegó y configuró la aplicación, orquestada por Kubernetes, con los manifiestos construidos anteriormente. Adicional a esto, se instalaron las integraciones necesarias para el monitoreo del cluster, estas fueron Kiali, Prometheus, Jaeger y Grafana.

Debido a que Kubernetes es un ambiente homogéneo en cualquier ambiente, el despliegue y configuración de cluster en la nube fue exactamente igual al del ambiente local, por lo que el proceso fue rápido y eficiente.

F4. Pruebas de performance de una plataforma cognitiva usando la inyección del data plane de Istio

Con la aplicación cognitiva configurada en la nube se continuó simulando cargas de trabajo reales para observar el rendimiento del cluster. Además, se verificó la tolerancia a fallos y la flexibilidad del sistema en nuevos despliegues mediante artefactos propios de Istio para simular situaciones como despliegues canarios o fallos del servicio.
Se prestó mucha atención a los búferes del sistema y a los momentos críticos en los que la resistencia de Kubernetes debe demostrar las políticas con las que se debe establecer la configuración inicial. En efecto, los factores de replicación y las configuraciones horizontales de los pods entraron en acción. El principio asíncrono de la aplicación ayudó enormemente a evitar el bloqueo de tareas en el sistema. En los momentos críticos en los que el hardware no era suficiente, nuestro proveedor de servicios daba las alertas pertinentes, al igual que Istio, en las que, desde la interfaz gráfica, se evidenciaba la pérdida de paquetes. En cualquier caso, la carga masiva representaba unas 3 veces el tráfico esperado y se comportó como se esperaba.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guane-ocr-deploy
  namespace: default
  labels:
    role: guane-ocr-service
spec:
  replicas: 1
  selector:
    matchLabels:
      role: guane-ocr-service
      tier: web-service
  template:
    metadata:
      labels:
        role: guane-ocr-service
        tier: web-service
    spec:
      containers:
        - name: guane-ocr
          image: localhost:5000/guane-ocr:1.0.0
          imagePullPolicy: Always
          args:
            [
              "gunicorn",
              "app.main:app",
              "-w",
              "1",
              "-k",
              "uvicorn.workers.UvicornWorker",
              "-b",
              "0.0.0.0:80",
              "--timeout",
              "3600",
            ]
          ports:
            - containerPort: 80
          envFrom:
            - configMapRef:
                name: guane-ocr-config
```

Figura 6. Definición de un deployment de kubernetes

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: guane-ocr-config
  namespace: default
data:
  TESTING: "0"
  WEB_APP_VERSION: "3.0.0"
  WEB_APP_TITLE: "OCR Guane API"
```

Figura 7. Definición de un configmap de kubernetes

Resultados y análisis

Luego de tener la aplicación desplegada en Kubernetes y con Istio totalmente configurado, se tiene visualización y control total de la aplicación. Istio permite la configuración de *virtual services* (ver Fig. 8) y *destination rules* (ver Fig. 9) para el enrutamiento personalizado del tráfico por medio de archivos de configuración; también aplica automáticamente mutual TLS a la comunicación entre pods evitando cualquier ataque de *man in the middle* a la comunicación intra-cluster.

```yaml
kind: VirtualService
apiVersion: networking.istio.io/v1alpha3
metadata:
  name: auth-canary-vs
  namespace: default
spec:
  hosts:
    - guane-ocr-auth-service.default.svc.cluster.local
  http:
    - route:
        - destination:
            host: guane-ocr-auth-service.default.svc.cluster.local
            subset: safe-group
          weight: 90
        - destination:
            host: guane-ocr-auth-service.default.svc.cluster.local
            subset: risky-group
          weight: 10
```

Figura 8. Definición de *virtual service* de Istio

```yaml
kind: DestinationRule
apiVersion: networking.istio.io/v1alpha3
metadata:
  name: auth-canary-dr
  namespace: default
spec:
  host: guane-ocr-auth-service
  subsets:
    - labels:
        version: safe
      name: safe-group
    - labels:
        version: risky
      name: risky-group
```

Figura 9. Definición de *destination rule* de Istio

Se logró tener total control y visualización de las métricas del cluster y la aplicación, tales como el tráfico entre componentes del sistema, tiempos de respuesta, códigos de estado de las peticiones, entre otros. Por medio de las integraciones de Istio: Kiali (ver fig. 10), Prometheus y Grafana (ver Fig. 11).

Uno de los requerimientos fundamentales para usar Jaeger es la propagación de las cabeceras de la petición a todo el sistema, solo es posible si se configura a nivel de la aplicación. En el actual reporte prescindimos de usar esta característica de Istio debido a que las aplicaciones a desplegar se encontraban en una etapa de desarrollo madura donde no fue posible modificar el proyecto. Sin embargo, se consiguió una aplicación tolerante a fallos usando técnicas como los circuit breakers para aislar microservicios defectuosos y evitar fallos en cascada en la aplicación.

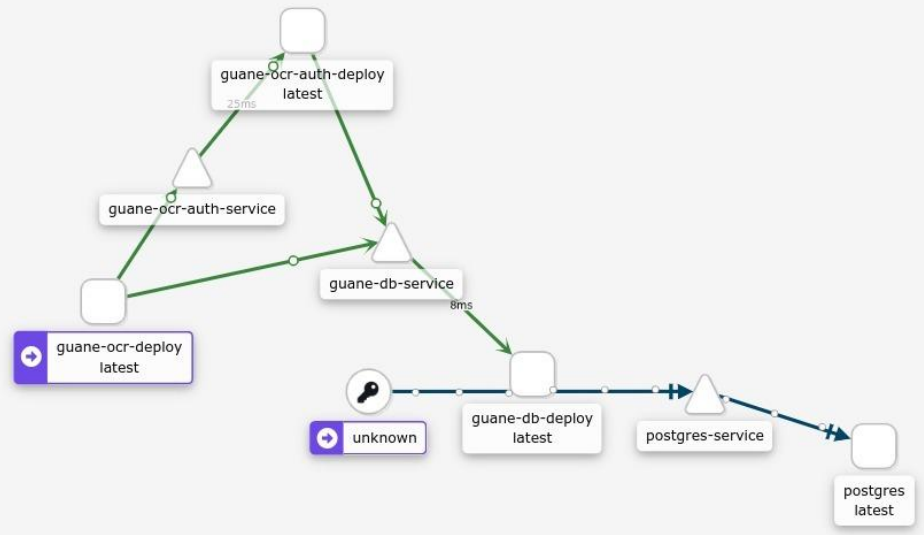

Figura 10. Visualización de la aplicación desde la interfaz gráfica Kiali

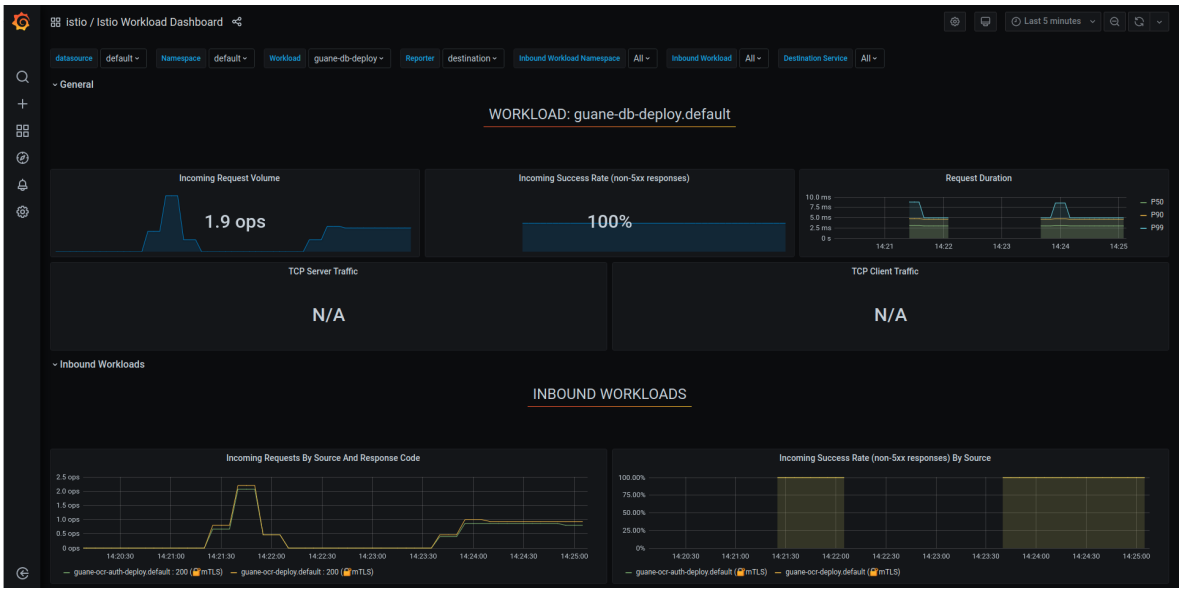

Figura 11. Visualización de métricas de la aplicación desde la interfaz gráfica de Grafana

Conclusiones

- El diseño de Kubernetes dispone de un ambiente robusto, dinámico y eficiente para una aplicación orientada a microservicios, controlado por archivos de configuración estructurados que permite controlar toda la infraestructura y realizar cambios sin afectar el rendimiento de la aplicación.
- Istio ofrece herramientas de monitoreo potentes y confiables, como Kiali, Prometheus y Grafana, para la supervisión y obtención de métricas del flujo de peticiones a través de toda la aplicación.
- Istio proporciona mecanismos de enrutamiento de tráfico, ideales para despliegues canarios. Además, permite añadir a la aplicación *Circuit Breakers* para evitar fallos en cascada sin tener que modificar código de la aplicación.
- Es posible manipular y gestionar el tráfico del sistema sin representar ningún cambio a nivel de la aplicación.
- El seguimiento de peticiones a través del sistema distribuido es una tarea compleja; con herramientas como jaeger se puede hacer fácilmente pero es obligatorio cumplir con algunos requisitos a nivel de código.
- La homogeneidad y uniformidad de Kubernetes permite que funcione totalmente igual en un ambiente local o en la nube.
- Combinando soluciones modernas como Kubernetes e Istio en un entorno de ejecución en la nube (GCP) se obtiene un sistema que es resiliente con una alta disponibilidad, elástico y eficiente para cargas de trabajo variable. Particularmente es muy útil y recomendable para llevar a cabo sistemas cognitivos por lo variante que pueden llegar a ser el volumen de transacción que estos manejan.

Referencias bibliográficas

[1] Bosch, J. (2004). Software Architecture: The Next Step. *Software Architecture: The Next Step*, 194-199. https://doi.org/10.1007/978-3-540-24769-2_14

[2] Abbasi, A. A., Abbasi, A., Shamshirband, S., Chronopoulos, A. T., Persico, V., & Pescape, A. (2019). Software-Defined Cloud Computing: A Systematic Review on Latest Trends and Developments. *IEEE Access*, *7*, 93294-93314. https://doi.org/10.1109/access.2019.2927822

[3] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2018). Microservices: How To Make Your Application Scale. *Lecture Notes in Computer Science*, 95-104. https://doi.org/10.1007/978-3-319-74313-4_8

[4] Wheatley, M. (2020, Junio). *Gartner says container adoption will grow rapidly, but it won't be that profitable*. SiliconANGLE. https://siliconangle.com/2020/06/25/gartner-says-container-adoption-will-grow-rapidly-wont-profitable/

[5] Brewer, E. A. (2015). Kubernetes and the path to cloud native. *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, 2015. https://doi.org/10.1145/2806777.2809955

[6] Li, W., Lemieux, Y., Gao, J., Zhao, Z., & Han, Y. (2019). Service Mesh: Challenges, State of the Art, and Future Research Opportunities. 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 1. https://doi.org/10.1109/sose.2019.00026

[7] Chen, Rui & Li, Shanshan & Li, Zheng (Eddie). (2017). From Monolith to Microservices: A Dataflow-Driven Approach. 466-475. 10.1109/APSEC.2017.53.

[8] *Monolithic Architecture pattern*. (2020). Microservices.io. https://microservices.io/patterns/monolithic.html

[9] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software (1.a ed.). Addison-Wesley Professional. https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/

[10] Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. *2017 IEEE International Conference on Web Services (ICWS)*, https://ieeexplore.ieee.org/abstract/document/8029803. https://doi.org/10.1109/icws.2017.61

[11] Francesco, P. D., Malavolta, I., & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *2017 IEEE International Conference on Software Architecture (ICSA)*, https://ieeexplore.ieee.org/abstract/document/7930195. https://doi.org/10.1109/icsa.2017.24

[12] *Qué son los contenedores y cuáles son sus beneficios | Google Cloud*. (2016). Google Cloud. https://cloud.google.com/containers?hl=es-419

[13] *Máquinas virtuales vs contenedores*. (2020). [Gráfico]. https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms

[14] *Why ?* (2020). Docker. https://www.docker.com/why-docker

[15] *¿Qué es Kubernetes?* (2020, 17 junio). Kubernetes. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[16] ConfigMaps. (2021, 18 febrero). Kubernetes. https://kubernetes.io/docs/concepts/configuration/configmap/

[17] Pod | Kubernetes Engine Documentation |. (2021). Google Cloud. https://cloud.google.com/kubernetes-engine/docs/concepts/pod

[18] Service. (2021, 17 febrero). Kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/

[19] What is a Kubernetes deployment? (2021). Red Hat. https://www.redhat.com/en/topics/containers/what-is-kubernetes-deployment

[20] Namespaces. (2021, 1 abril). Kubernetes. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[21] ¿Qué es Service Mesh? (2020). Red Hat. https://www.redhat.com/es/topics/microservices/what-is-a-service-mesh

[22] *What is Istio?* (2020). Istio. https://istio.io/latest/docs/concepts/what-is-istio/

[23] Envoy Proxy - Home. (2021). Envoy. https://www.envoyproxy.io/

[24] *Arquitectura de Istio*. (2020). [Gráfico]. https://istio.io/latest/docs/concepts/what-is-istio/

[25] Kiali: Service mesh observability and configuration. (2021). Kiali. https://kiali.io/

[26] Grafana Features. (2021). Grafana. https://grafana.com/grafana/

[27] Overview | Prometheus. (2021). Prometheus. https://prometheus.io/docs/introduction/overview/

[28] Jaeger: open source, end-to-end distributed tracing. (2021). Jaeger. https://www.jaegertracing.io/

[29] OpenZipkin · A distributed tracing system. (2021). Zipkin. https://zipkin.io/

[30] Virtual Service. (2020). Istio. https://istio.io/latest/docs/reference/config/networking/virtual-service/

[31] Destination Rule. (2020). Istio. https://istio.io/latest/docs/reference/config/networking/destination-rule/

[32] Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., ... & Steenkiste, P. (2014, Diciembre). The cost of the" s" in https. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (pp. 133-140).

[33] Krawczyk, H., Paterson, K. G., & Wee, H. (2013, Agosto). On the security of the TLS protocol: A systematic analysis. In Annual Cryptology Conference (pp. 429-448). Springer, Berlin, Heidelberg.

[34] Security. (2020). Istio. https://istio.io/latest/docs/concepts/security/

[35] kind. (2021). Kind. https://kind.sigs.k8s.io/

[36] Overview of kubectl. (2021, 11 febrero). Kubernetes. https://kubernetes.io/docs/reference/kubectl/overview/

[37] Google Cloud. (n.d.). Cloud Computing Services |. Retrieved January 29, 2021, from https://cloud.google.com/