Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Desarrollo y Operaciones (DevOps)

Desarrollo práctico de despliegue de infraestructura multi-nube y automatización de las etapas operativas del ciclo de vida DevOps en el aplicativo SINET

Trabajo fin de estudio presentado por:	Luis Felipe Hernández Zambrano
Tipo de trabajo:	
Director/a:	Juan Manuel Gonzáles Calleros

Fecha:	

Resumen

El presente trabajo de investigación trata de la implementación de metodologías y herramientas DevOps a un proyecto de software existente llamado SINET el cual tiene como finalidad el manejo de activos informáticos, está desarrollado con una arquitectura de microservicios que se despliegan en dockers por medio de docker compose. En él se detalla las herramientas que se utilizaron para conseguir el objetivo en seis de las siete fases del ciclo DevOps que son desarrollo, integración, despliegue y monitorización, también se abarca con detalle las herramientas que se utilizaron en cada una de las fases, al igual se detalla que ajustes se debieron realizar al código del sistema para poder implementar algunas de las herramientas anteriormente mencionadas.

Palabras clave: DevOps, Microservicios, Integración continua, Entrega continua, Infraestructura como código

Abstract

This research work deals with the implementation of DevOps methodologies and tools to an existing software project called SINET which has the purpose of managing computer assets, it is developed with a microservices architecture that is deployed in dockers through docker compose. It details the tools that were used to achieve the objective in four of the seven phases of the DevOps cycle, which are development, integration, deployment and monitoring, it also covers in detail the tools that were used in each of the phases, by It also details what adjustments had to be made to the system code in order to implement some of the previously mentioned tools.

Keywords: DevOps, Microservices, Continuous Integration, Continuous Delivery, Infrastructure as code

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

1.1. Justificación del trabajo

En la actualidad, la implementación de infraestructura como código e integración y entrega continua se ha convertido en un aspecto crítico en el desarrollo de software, ya que permite la creación de infraestructuras de manera eficiente y la entrega de software de manera rápida y confiable. Sin embargo, este proceso puede ser complejo y propenso a errores si se realiza manualmente.

Por esta razón, en este trabajo se busca abordar el problema de la implementación de infraestructura como código e integración y entrega continua en un proyecto, con el objetivo de lograr independencia de proveedor y mejorar la eficiencia y la calidad del proceso de desarrollo de software.

Para lograr la independencia de proveedor, se utilizará Terraform, una herramienta de infraestructura como código que permite definir, crear y administrar la infraestructura como si fuera código. Terraform es compatible con varios proveedores de nube, lo que permite la creación de infraestructuras complejas de manera eficiente y garantiza que la infraestructura creada sea independiente del proveedor de nube, lo que facilita la portabilidad y la escalabilidad en diferentes entornos.

Por otro lado, para la integración y entrega continua, se utilizará GitHub Actions, una herramienta de CI/CD que permite automatizar los procesos de construcción, prueba y entrega de software directamente en GitHub. GitHub Actions es una herramienta superior a otras opciones como Jenkins, debido a su facilidad de uso y configuración, y permite integrar y automatizar todo el proceso de construcción, prueba y entrega de software en un solo lugar.

Al utilizar Terraform y GitHub Actions en conjunto, se puede crear un flujo de trabajo completo de infraestructura como código e integración y entrega continua que garantiza la independencia de proveedor y mejora la eficiencia y la calidad del proceso de desarrollo de software. Esto puede proporcionar beneficios significativos, como mayor flexibilidad, portabilidad, escalabilidad y control en el proceso de creación y administración de la

infraestructura y en la automatización del proceso de construcción, prueba y entrega de software.

En conclusión, la implementación de infraestructura como código e integración y entrega continua en un proyecto utilizando Terraform y GitHub Actions es una solución eficaz para lograr independencia de proveedor y mejorar la eficiencia y la calidad del proceso de desarrollo de software. Esta solución puede proporcionar beneficios significativos y facilitar el proceso de creación, administración y automatización de la infraestructura y el software en un entorno multi-nube.

1.2. Planteamiento del problema

El software SINET actualmente está siendo trabajado de manera manual en todas las etapas del proceso de desarrollo y despliegue de software. Este enfoque manual conlleva a una serie de problemas, como la propensión a errores humanos, la lentitud en el tiempo de entrega y la limitación en la escalabilidad del software. Además, la falta de herramientas de infraestructura como código y de orquestadores hace que el software dependa de la realización de procesos manuales por parte del personal técnico encargado de su desarrollo y despliegue.

La necesidad de implementar soluciones de infraestructura como código, integración y entrega continua, y orquestadores, surge de la necesidad de mejorar la eficiencia del ciclo de vida del software y permitir su escalabilidad. Para lograr esto, se requiere la implementación de herramientas y procesos para la creación de infraestructura automatizada, la configuración de ambientes, la integración de pruebas y despliegues automáticos, la gestión de versiones y la automatización de la orquestación de contenedores.

Por lo tanto, la propuesta del proyecto es aplicar varias de las herramientas tratadas en el máster DevOps, para mejorar la eficiencia del ciclo de vida del software SINET. Esto permitirá la escalabilidad del software y mejorará su desempeño en empresas donde se encuentre implementado, a través de la automatización de procesos y la eliminación de errores humanos en su desarrollo y despliegue.

1.3. Estructura de la memoria

En el presente trabajo, se aborda el problema de la falta de automatización en el ciclo de vida del software SINET, y se propone la implementación de soluciones de infraestructura como código, integración y entrega continua, así como orquestadores, para mejorar la eficiencia del ciclo de vida del software y permitir su escalabilidad.

En este sentido, la estructura de la memoria está diseñada para presentar de manera clara y organizada el desarrollo del proyecto y los resultados obtenidos. El documento está dividido en capítulos, cada uno de los cuales se enfoca en una etapa específica del proyecto.

Capítulo 2 Contexto y estado del arte

En este capítulo se proporciona un contexto general sobre el proyecto y se revisa el estado del arte en las tecnologías y herramientas relacionadas con la infraestructura como código, integración y entrega continua, orquestadores, y otros aspectos relevantes para el desarrollo de software. Se exploran las principales tendencias y desafíos en la industria, así como las soluciones existentes y las mejores prácticas en la implementación de estas tecnologías. Además, se analizan casos de estudio y proyectos similares que se han desarrollado en el mercado actual, para poder tener una visión completa y actualizada del sobre el trabajo del máster.

Capítulo 3 Objetivos y metodología de trabajo

Se presentan el objetivo general y los objetivos específicos, que definen la finalidad y los logros concretos que se persiguen con el trabajo. Además, se detallará la metodología que se utilizará para llevar a cabo el proyecto, la cual permitirá desarrollar el trabajo de forma organizada. En resumen, este capítulo sienta las bases para la realización del proyecto y establece la dirección y los principios que se seguirán a lo largo del mismo.

Capítulo 4 Desarrollo especifico de la contribución

Describe la implementación de la solución propuesta, en el orden de las fases del ciclo DevOps, empezando desde la fase 1, donde se define el plan de trabajo; la fase 2 de desarrollo tiene un énfasis especial sobre los requisitos y posteriores ajustes que debe cumplir un software existente para que pueda ser orquestado y posteriormente acoplado a

al ciclo DevOps; en resumen, este capítulo se detallan los procesos de configuración de la infraestructura automatizada.

Capítulo 5 Conclusiones y trabajo futuro

Presenta los resultados obtenidos durante la implementación de la solución propuesta, incluyendo mejoras en la eficiencia del ciclo de vida del software y la escalabilidad de este.

2. Contexto y estado del arte

En esta investigación se aborda la actualización de un software basado en microservicios desplegado actualmente con Docker Compose, para adaptarlo a metodologías del marco DevOps. El objetivo es mejorar la colaboración entre desarrollo y operaciones, agilizar la entrega de software y lograr una mayor calidad del producto final, además y muy importante recalcar la entrega continua de valor del producto con despliegue a corto plazo de actualizaciones que el software requiera.

Se revisará el estado del arte en las metodologías que abarca DevOps, el manejo de la infraestructura como código, orquestación y la distribución y entrega continua. Se analizarán las prácticas y herramientas más relevantes en cada una de las fases del ciclo DevOps, para así establecer una base teórica sólida.

Se estudiarán investigaciones previas sobre la adaptación o desarrollos desde cero basadas en la arquitectura de microservicios desplegados sobre herramientas DevOps y sistemas orquestados, así como las experiencias en diferentes nubes. Se identificarán casos de éxito, limitaciones y áreas por mejorar.

Este capítulo proporcionará una comprensión clara del estado actual de las metodologías y herramientas DevOps y su aplicación especifica basada en despliegues de aplicaciones basadas en microservicios. Además, sentará las bases para la investigación propuesta, contribuyendo al conocimiento existente y guiando futuras investigaciones en el campo.

2.1. Contextualización y antecedentes

2.1.1. Contexto Empresarial

El software SINET es una solución informática desarrollada para el manejo de inventario de activos informáticos en entornos empresariales con ajustes a la media solicitados por los diferentes clientes de la empresa. Su objetivo es facilitar la administración y seguimiento de activos desde el momento de la compra, su asignación, ubicación, actualizaciones, solicitudes de soporte sobre los mismos, hasta su baja por depreciación, obsolescencia o daño irreparable.

La actualización del software SINET hacia metodologías DevOps es relevante y urgente de cierta forma ya que, gracias al éxito de este, la administración, adecuaciones y mantenimiento se está volviendo cada vez más difícil y tiene un sin número de problemas que trataremos en el siguiente apartado las cuales se pretende resolver adoptando metodologías y herramientas DevOps.

La integración de DevOps a esta plataforma permitirá aprovechar la flexibilidad y escalabilidad que esta nos brinda optimizando los recursos; en resumen, la adopción de metodologías y herramientas DevOps buscan mejorar la eficiencia y productividad en la administración de la plataforma SINET permitiéndole crecer de una manera controlada y bajando la incertidumbre que genera la realización de procesos manuales.

2.1.2. Problemas y limitaciones actuales del software SINET

El software SINET presenta limitaciones significativas debido a su enfoque tradicional, sin utilizar herramientas y practicas DevOps, estas limitaciones que en el momento se solucionan de manera manual por falta de automatización de procesos al final se traducen en costos elevados en mantenimiento del software.

Entre los principales problemas que se han detectado por su modelo de trabajo tenemos los siguientes.

1. Falta de automatización den los procesos de desarrollo, despliegue y gestión del software, Esto puede resultar en tareas manuales repetitivas, retrasos en la entrega de nuevas funcionalidades y dificultades para mantener un ritmo constante en las entregas de diferentes actualizaciones particulares que se desarrollan para los clientes del sistema.
2. Dificultades en la escalabilidad del software en entornos empresariales en crecimiento, debido a la ausencia de herramientas de orquestación y gestión de infraestructura lo cual dificulta el manejo la correcta operación del sistema.
3. Limitada visibilidad del estado y disponibilidad de los servicios en operación, ya que no existe monitorización de los microservicios desplegados y en caso de falla, la recuperación de estos se realiza de manera manual.
4. Downtime elevado en procesos de actualización, debido a que las actualizaciones se realizan de forma manual, si bien estas son programadas, los horarios deben ser

fuera de tiempos de uso del sistema, ocasionando generación de cargos extra al personal encargado de estos procesos, lo cual se traduce en costos de operación en mantenimiento de la infraestructura.

5. Ausencia total de autohealing, ya que al no trabajar con servicios orquestados los cuales realizan monitorización constante de la operación de sus contenedores, no es posible la implementación de esta característica esencial en los sistemas de hoy en día, muy necesaria para garantizar la recuperación y disponibilidad del sistema en operación.
6. Ausencia total de estrategias de despliegue (Las más utilizadas son Blue Green y Canary), al igual que el punto anterior, necesarias para garantizar disponibilidad de los servicios, ya que con estas estrategias el downtime por actualización desaparece ya que estas se encargan de gestionar despliegues controlados.
7. Ausencia total de procesos automáticos en procesos de building como son ejecución de pruebas automatizadas, necesarias para garantizar la integridad en el funcionamiento del software luego de la realización de nuevas funcionalidades.

Como podemos ver, son grandes las falencias existentes que se pretende resolver al aplicar las metodologías y herramientas DevOps actuales que pueden garantizar la solución de los problemas y dificultades anteriormente mencionados.

2.1.3. Tecnologías Clave

En este punto, se identificarán las tecnologías clave relacionadas con la implementación de metodologías DevOps en la actualización de la infraestructura necesaria para el despliegue del software SINET. Estas tecnologías aquí descritas desempeñan un papel fundamental para lograr el objetivo de este trabajo de fin de máster. A continuación, se presenta las tecnologías claves a considerar, con citas bibliográficas referentes a la explicación de que es cada una de ellas

2.1.3.1. Maven

Herramienta popular para la construcción y gestión de proyectos Java. Proporciona una forma fácil de organizar, compilar y empaquetar el código fuente, además de manejar las dependencias de manera automática. Con Maven, se puede simplificar el proceso de

desarrollo y asegurar de que un proyecto esté correctamente configurado y listo para su distribución.

2.1.3.2. Terraform

Herramienta open source, desarrollada por Hashicorp la cual se encarga de crear, gestionar y configurar una infraestructura como código, su configuración está basada en un modelo de configuración declarativo y tiene soporte de despliegue en varias nubes, por mencionar las más importantes como AWS, Azure, Google Cloud, Digital Ocean entre otras.

2.1.3.3. Docker

Es una plataforma de código abierto que permite la creación de máquinas virtuales ligeras (contenedores) desplegadas generalmente para un fin común, como puede ser la ejecución de un microservicio en una tecnología especifica, sea java, nodejs, python, por mencionar algunas, a diferencia de las máquinas virtuales convencionales, esta comparte el kernel del anfitrión consiguiendo así su gran ventaja que es el ahorro de recursos de hardware.

2.1.3.4. Kubernetes

Es una plataforma de código abierto cuyo objetivo es orquestar y gestionar contenedores, proporcionando entornos que permiten la automatización de tareas como esenciales en los entornos actuales como son la escalabilidad y la autogestión de los recursos que administra, garantizando alta disponibilidad de los recursos que administra.

2.1.3.5. GitHub Actions

Es una herramienta implementada por GitHub de gran popularidad que permite la implementación de integración y entrega continua (CI/CD), a diferencia de Jenkins esta funciona como una SaaS, por tanto ya se encuentra configurada para la cuenta de Git, para poder hacer uso de esta solamente es necesario tener una organización con configuración paga con alguno de los planes ofrecidos por GitHub, su configuración es muy intuitiva y goza de grandes beneficios ya que como se encuentra integrada a GitHub donde se tiene el repositorio de código fuente, su implementación es mucho más sencilla.

2.1.3.6. GitHub Container Register (ghcr)

Es un servicio ofrecido por GitHub similar a DockerHub, permite almacenar y administrar imágenes de contenedores de Docker dentro de la plataforma de GitHub. Proporciona a los

desarrolladores un registro de contenedores privado y seguro, directamente integrado con sus repositorios de código fuente.

Este servicio es utilizado en CI/CD para despliegues en Kubernetes para compartir imágenes dockerizadas de contenedores las cuales posteriormente son desplegadas como pods.

2.1.3.7. EKF

La pila EFK (Elasticsearch, Fluentd y Kibana) es una combinación de tres herramientas ampliamente utilizadas en la administración y análisis de registros y datos para la fase de monitorización en el DevOps.

- **Elasticsearch:** Es un motor de búsqueda y análisis distribuido que se utiliza para almacenar y consultar datos en tiempo real. Es altamente escalable y está diseñado para manejar grandes volúmenes de datos estructurados y no estructurados. Elasticsearch es especialmente adecuado para la búsqueda de texto completo y análisis de datos en tiempo real.
- **Fluentd:** Es un recolector y envío de registros (log) de datos, que permite centralizar y unificar los registros de diferentes fuentes en un único lugar, como Elasticsearch. Fluentd recopila y transporta registros desde diversas fuentes, normaliza y etiqueta los registros y los envía a los destinos configurados, como Elasticsearch para su almacenamiento y análisis.
- **Kibana:** Es una interfaz web de código abierto que proporciona una plataforma de visualización y análisis de datos para Elasticsearch. Kibana permite crear tableros interactivos, gráficos, diagramas y visualizaciones basadas en los datos almacenados en Elasticsearch. Es una herramienta poderosa para la exploración y análisis de datos en tiempo real.

En conjunto, la pila EFK proporciona una solución integral para la gestión, análisis y visualización de registros y datos en tiempo real. Es especialmente útil en entornos distribuidos y en aplicaciones que generan grandes volúmenes de registros o datos que requieren un análisis y monitoreo continuo.

2.1.3.8. Kube Ops View

Esta herramienta de código abierto está diseñada para proporcionar una interfaz visual que permite monitorear y administrar clústeres de Kubernetes. Esta herramienta facilita la

visualización del estado, los recursos y las métricas del clúster de una manera más amigable y comprensible que la línea de comandos.

2.2. Trabajos relacionados

A continuación se presenta una serie de trabajos relacionados en la implementación de metodologías y herramientas DevOps, de los cuales se destaca que todos buscan objetivos relacionados a lo que se pretende conseguir con la implementación de una cultura DevOps, como son la aceleración en el tiempo de entrega, la mejora en la calidad del software, incremento en la escalabilidad y confiabilidad de los sistemas informáticos, colaboración y comunicación entre equipos, optimización de recursos y reducción de costos, eliminación de procesos manuales, entre otros.

2.2.1. Objetivos de la aplicación de la cultura DevOps.

Es indiscutible que la aplicación de la cultura DevOps y sus diferentes metodologías, buscan objetivos comunes, todos relacionados con agilidad, calidad, tiempo y eficiencia, buscando la satisfacción del cliente final; esto se puede evidenciar en los diferentes trabajos relacionados como lo mencionan a continuación las siguientes citas:

> *"se analizó el escenario inicial del equipo de trabajo, a fin facilitar la transición hacia un escenario ágil y adoptar el enfoque DevOps para el desarrollo de nuevas aplicaciones.* (Velásquez Santos, 2022)*"*

> "*. Ante esta situación, el presente trabajo de titulación define los procesos y flujo de trabajo para la implementación de una arquitectura basada en contenedores y buenas prácticas DevOps, con el fin de entregar productos de alta calidad en el tiempo previsto y con el presupuesto planificado.* (Alexander, 2021)"

> "*"La presente investigación da cuenta de la implementación de DevOps en el proceso de integración y despliegue de software en una organización del sector seguros de Lima, 2019, debido a que, anterior a la implementación se tenían problemas de velocidad y calidad. El objetivo general fue determinar el efecto de DevOps sobre el proceso de integración y despliegue de software en el sector de seguros del distrito de San Isidro, Lima, 2019.* (Yeren, 2020)"

> *En el mundo actual, dos factores inciden en la calidad del software, por un lado, la complejidad de los requerimientos y por otro, las prácticas que aplica la industria del software* (Hernández-Alonso, 2022)

De tal modo que para este trabajo no será una excepción enfatizar en los mismos objetivos, sin embargo, esta investigación se trabajará las 7 fases del ciclo DevOps y no una sola línea en específico del ciclo DevOps ni tampoco una sola herramienta, si no un conjunto de herramientas y metodologías relacionadas buscando ser una guía aplicable a futuro para este tipo particular de implementaciones de DevOps sobre tecnologías ya existentes.

2.2.2. Actualidad de la cultura DevOps

DevOps abarca una serie de metodologías, herramientas y etapas claramente definidas, lo que lo convierte en algo más grande que una guía, como lo explica la cita a continuación:

> *"DevOps es un marco de trabajo y una filosofía en constante evolución que promueve un mejor desarrollo de aplicaciones en menos tiempo y la rápida publicación de nuevas o revisadas funciones de software o productos para los clientes.* (netapp, 2023)"

En ese sentido si bien las metodologías y herramientas puede cambiar y efectivamente han ido cambiando y evolucionando como en el caso específico de los orquestadores donde se miró una evolución desde Docker compose hasta lo que ahora es Kubernetes, la filosofía y su cultura permanecerán constantes a lo largo del tiempo. Y esto se debe a que sus resultados ya han sido demostrados, por un ejemplo de un trabajo documentado:

> *La presente memoria se trata de la documentación de lo que supone el diseño y la implementación de un sistema local de integración continua. Este es un concepto que durante los últimos años ha ido cogiendo fuerza en la industria del desarrollo de software. Se explica qué es DevOps, por qué toma fuerza con respecto a los antiguos modelos de deslocalizar los equipos de desarrollo de los equipos de operaciones, prácticamente unificándolos. En este documento es descrito el proceso de creación de un entorno de desarrollo e integración continua en local, y el diseño del proceso DevOps creado. Se describirán las ventajas de la metodología, así como los pasos que se han seguido para la configuración del sistema y su uso.* (Durández, 2020)

2.2.3. Casos de éxito de implementación DevOps

Es indudable el éxito que ha tenido DevOps y cada día son más las empresas que deciden adoptar esta filosofía para el desarrollo de soluciones informáticas en sus empresas, por mencionar algunas tenemos a Netflix, donde en su blog detalla la infinidad de problemas que presentaban y como los fueron superando, aplicando un nuevo enfoque de trabajo basado en la cultura DevOps.

> "The year was 2012 and operating a critical service at Netflix was laborious. Deployments felt like walking through wet sand. Canarying was devolving into verifying endurance ("nothing broke after one week of canarying, let's push it") rather than correct functionality. Researching issues felt like bouncing a rubber ball between teams, hard to catch the root cause and harder yet to stop from bouncing between one another. All of these were signs that changes were needed."

> "Corría el año 2012 y operar un servicio crítico en Netflix era laborioso. Los despliegues se sentían como caminar sobre arena mojada. Canarying se estaba convirtiendo en verificar la resistencia ("nada se rompió después de una semana de canarying, empujémoslo") en lugar de corregir la funcionalidad. Investigar problemas se sintió como hacer rebotar una pelota de goma entre los equipos, difícil de detectar la causa raíz y aún más difícil de evitar que reboten entre sí. Todos estos eran signos de que se necesitaban cambios. (Netfilx, 2023)"

2.2.4. Herramientas comunes en fases del ciclo DevOps

En la revisión que se hizo de los trabajos relacionados se encontramos que existen unas herramientas imprescindibles que se han utilizado comúnmente entre ellos, indiscutiblemente encontramos a Docker ya que esta es la herramienta base que nos permite la virtualización de componentes ligeros, por nombrar algunos apartes encontramos las siguientes:

> *"Se ha comunicado y acompañado a los miembros del equipo, habilitándoles en el uso de las herramientas Jira, Shell scripting, Jenkins, Docker, AWS y otras como parte de ese stack* (Hernández-Alonso, 2022)*"*

" La aplicación utilizará tecnología de contenedores como Docker y para el despliegue automatizado de la infraestructura tecnologías como Terraform y Ansible. (Suntaxi-Cocanguilla, 2020)*"*

"Poco a poco viene a ser una buena estrategia la implementación de dockers ya que está siendo adoptada por la mayoría de los proveedores de plataformas: AWS, Microsoft Azure, etc. La construcción y ejecución de contenedores se la puede realizar en cualquier sistema operativo (Alexander, 2021)*"*

Cuando se trabaja con contenedores, estos van de la mano de un orquestado, en este caso estamos hablando de Kubernetes, y es referente en el montaje de soluciones escalables y autogestionadas. y casi que por norma general el uso de esta dupla es la más utilizada, sobre Kubernetes, y al igual que Docker se puede encontrar muchos trabajos de investigación donde el interés principal es el uso de esta herramienta. A continuación, ponemos unos apartes donde mencionan su uso:

"En este Sprint se realiza el despliegue de la aplicación el cual se realizará en un clúster de Kubernetes. Para ello se creará un proyecto independiente el cual contendrá todas instrucciones necesarias para su funcionamiento (Alexander, 2021)*"*

"En el diagrama anterior, se muestran los módulos del sistema que están instalados en el clúster de Kubernetes y que forman nuestro sistema de integración continua. Pag 55 (Durández, 2020)*"*

Existen otras más que dependiendo de los alcances del proyecto a desarrollar pueden ser utilizadas o no, entre ellas encontramos a packer y Terraform, sin embargo, estas dos están siendo reemplazadas a menudo por soluciones que se encuentran como servicios por los proveedores de Cloud en el caso de AWS encontramos el AWS Cloud Formation o para el caso de Azure encontramos Azure Resource Manager.

En el caso de Jenkins, esta era una herramienta muy utilizada en la fase de integración y despliegue pero que en la actualidad está siendo reemplazada por un servicio muy popular y es GitHub Actions, que será el que utilizaremos como parte de la solución en este proyecto.

2.3. Conclusiones del estado del *arte*

La implementación de herramientas DevOps de algunas fases en concreto sobre sistemas existentes es una práctica muy común y se puede evidenciar en varios de los trabajos anteriormente mencionados, sin embargo, muy pocos de ellos tienen el concepto claro de lo que en realidad es DevOps, su filosofía y el énfasis en la colaboración lo cual se evidencia en varias de las referencias mencionadas cuando dicen que al aplicar algunas herramientas de fases especificas están aplicando DevOps.

Sin embargo, no se quita merito a los trabajos realizados porque así traten solo fases específicas del ciclo DevOps, evidencia éxito en las implementaciones expuestas. Igual es comprensible porque del mismo modo se puede observar que si es clara la dificultad del cambio de pensamiento en la adopción de nuevas metodologías, más aún la adopción de una nueva cultura de trabajo como lo es DevOps. En sí, esta resistencia va de la mano con el tipo de implementaciones que se quieren actualizar, ya que al igual que ellas, las personas que intervinieron en su creación vienen de un modelo de trabajo relegado a las metodologías del tiempo en el que las herramientas tecnológicas fueron construidas.

Una conclusión importante sobre lo anteriormente expuesto es que la implementación de herramientas DevOps es relativamente sencillo si se cuenta con el conocimiento para hacerlo, lo realmente difícil es el cambio de mentalidad de un equipo de trabajo hacia una nueva forma de trabajar, basada en la colaboración y eliminando la burocracia en procesos que uno podrían ser automatizados y dos la resistencia de pasar de un modelo jerárquico piramidal a un modelo lineal y colaborativo.

Otro punto importante para resaltar de las referencias de trabajos existentes es que todos buscan objetivos comunes que van de la mano a los que se pretende conseguir con la implementación de una cultura DevOps que se puede resumir en la colaboración entre equipos, la entrega continua, la mejora de la eficiencia y la calidad del software.

Sobre el proyecto específico que se pretende realizar al sistema SINET haciendo una selección de herramientas adecuadas en cada una de las fases del ciclo DevOps podemos concluir que es totalmente viable ya que esto también se evidencio en los trabajos mencionados anteriormente, no sin embargo es necesario que el aplicativo SINET cumpla unos requisitos los cuales se irán exponiendo con el desarrollo del proyecto.

También encontramos que a pesar de que DevOps viene desde el año 2009, este ha tomado un interés fuerte recientemente, tanto así que no se encontró bibliografía o artículos universitarios con más de 3 años de antigüedad a la fecha de este estudio.

En cuanto a las herramientas DevOps utilizadas, todas van alineadas a las fases correspondientes y muchas se siguen usando en la actualidad, y otras han pasado a ser reemplazadas por servicios específicos prestados por los proveedores de Cloud.

Finalmente podemos concluir que el desarrollo de software ha evolucionado de tal forma que es necesario conocer todo lo que esta tras bambalinas para un correcto despliegue y esto solo es posible si se tiene claro el objetivo principal de la cultura DevOps que es la comunicación de todas las especialidades que conciernen en un fin común que es la mejora continúa buscando alta calidad y confiabilidad en el producto entregado.

3. Objetivos y metodología de trabajo

En el ámbito del desarrollo de software, existe un creciente interés en la implementación de metodologías y herramientas que permitan mejorar la eficiencia y calidad de los procesos buscando en gran medida la automatización de estos. En este proyecto se busca documentar el paso a paso de cómo hacerlo con un desarrollo de software especifico, el cual actualmente enfrenta con desafíos relacionados a la eficiencia en el despliegue y la calidad en el desarrollo.

Este trabajo de investigación se enfoca en abordar estos desafíos mediante la implementación de metodologías y herramientas específicas, aprovechando las prácticas de desarrollo y despliegue que se derivan del uso de diversas herramientas específicas de cada una de las etapas del ciclo DevOps para mejorar la eficiencia y calidad en el proyecto SINET.

Para lograrlo, se realizará un análisis exhaustivo de las características actuales de SINET, incluyendo su arquitectura basada en microservicios y el método de despliegue mediante contenedores. Este análisis permitirá identificar las limitaciones y áreas de mejora del proyecto en términos de eficiencia y calidad.

Además, se explorarán las metodologías y herramientas DevOps más adecuadas para abordar estos desafíos. Estas metodologías y herramientas se centran en la colaboración entre equipos de desarrollo y operaciones, la automatización de procesos y la entrega continua de software.

Con base en este análisis y exploración, se realizarán ajustes y adaptaciones en el código del sistema SINET para habilitar la implementación de las metodologías y herramientas seleccionadas con el fin de lograr una mayor eficiencia en el despliegue del software, una mejor calidad en el desarrollo y una mayor agilidad en los procesos.

3.1. Objetivo general

El objetivo general de este trabajo de investigación es implementar metodologías y herramientas DevOps en un proyecto de desarrollo de software que actualmente no las maneja, en este caso específico es el proyecto SINET, con el fin de mejorar la eficiencia en el desarrollo y despliegue del software, así como la calidad del producto final. Se busca lograr

una mayor automatización de los procesos, una entrega continua de software y una mayor agilidad en el ciclo de vida del proyecto, contribuyendo así a la optimización de los recursos y la mejora de la experiencia del usuario.

3.2. Objetivos específicos

Entre los objetivos específicos de este trabajo tenemos los siguientes:

- Analizar la arquitectura de microservicios de SINET y evaluar si como fue desarrollada esta apta para ser desplegada y administrada por un orquestador, o si por el contrario debe ser ajustada para que cumpla con los requisitos de Kubernetes.
- Identificar las limitaciones actuales del método de despliegue basado en Docker Compose en términos de automatización y escalabilidad, y proponer mejoras utilizando herramientas de orquestación de contenedores como Kubernetes.
- Desarrollar un pipeline de integración continua (CI) que permita la automatización de las pruebas y la detección temprana de errores en el desarrollo de SINET.
- Comparar el proceso de CI/CD de una herramienta on-premise como Jenkins contra GitHub Actions contratada como SaaS
- Automatizar todos los procesos que se identifiquen como automatizables.
- Implementar un sistema de monitorización utilizando herramientas como ELK (Elasticsearch, Logstash, Kibana) para recopilar y analizar datos de rendimiento y registros del sistema, mejorando así la visibilidad y la capacidad de respuesta ante problemas.
- Evaluar el impacto de la implementación de metodologías DevOps en el tiempo de entrega de nuevas funcionalidades y en la estabilidad del sistema SINET.
- Documentar los procesos, las herramientas y las configuraciones utilizadas en la implementación de las metodologías DevOps en SINET, facilitando así la replicación y el mantenimiento futuro del sistema.
- Identificar y evaluar los beneficios del uso de herramientas independientes de proveedor cloud, con el fin de evitar el vendor lock-in y promover la interoperabilidad e independencia.

- Calcular el costo total de despliegue de la implementación de metodologías y herramientas DevOps en el proyecto SINET, en tres proveedores cloud que son, AWS,
- Azure y DigitalOcean.
- Comparar los costos asociados a la implementación del proyecto SINET en distintos proveedores de servicios cloud.
- Realizar un análisis detallado de los precios y tarifas de los servicios ofrecidos por cada proveedor, considerando aspectos como el costo de las instancias, el almacenamiento, la transferencia de datos y otros servicios adicionales relevantes.
- Evaluar las diferencias en los modelos de precios y opciones de descuento ofrecidas por cada proveedor, con el fin de determinar la opción más económica y rentable para el despliegue de la infraestructura y los servicios cloud requeridos en el proyecto.

3.3. Metodología del trabajo

Este trabajo de fin de Máster es una investigación acerca de la implementación de las fases del ciclo DevOps que involucran la implementación de herramientas utilizadas para lograr los objetivos principales de la implementación de este ciclo sobre sistemas existentes y que carecen de las mismas, empezando desde la fase de desarrollo.

Lo anterior con el fin de que posteriormente sirva de guía, para quienes requieran en un futuro actualizar un sistema convencional predominantemente on premise, a un modelo basado en ciclo DevOps con entornos virtualizados, para ello se inicia con una contextualización de las herramientas de trabajo.

En la fase de **desarrollo**, se abarcarán los ajustes que se deban hacer a los microservicios para ser compatibles con un modelo gobernado desde un orquestador, tales como la verificación de los archivos Dockerfile para su correcto empaquetamiento, gestión de recursos, exposición del servicio, manejo de estados si lo requiere y Health Check entre otros, igualmente y no menos importantes, la creación y verificación de pruebas unitarias.

En la fase de **integración**, se abordará lo relacionado a la configuración de GitHub Esta fase tiene como objetivo principal lograr una integración continúa fluida entre los diferentes componentes del proyecto y garantizar la calidad y estabilidad del software. En esta etapa,

se definen y configuran los flujos de trabajo de GitHub, que representan los pasos y acciones a seguir durante la integración continua. Esto implica establecer las políticas para el manejo de repositorios y flujo de trabajo.

En la fase de **despliegue**, se trabajarán diversas actividades relacionadas con la implementación y puesta en marcha del software en los entornos de pruebas y producción. En esta etapa, se utilizarán varias herramientas clave, entre ellas, Terraform para facilitar y automatizar el proceso de despliegue y nuevamente GitHub Actions para abordar la etapa de entrega continua, Importante resaltar que, en esta fase, se aprovecharán las herramientas anteriormente descritas para lograr un despliegue automatizado y confiable del software en entornos de producción. Estas herramientas permiten la configuración de flujos de trabajo, la creación de imágenes preconfiguradas y la gestión de la infraestructura como código, lo que contribuye a una implementación más eficiente, consistente y controlada del software.

En la fase de **operación**, se abarcará todo lo relacionado con la puesta a punto de Docker y Kubernetes, dos tecnologías clave para el despliegue y la gestión de contenedores en los entornos que se requiera desplegar.

En primer lugar, se llevará a cabo la configuración y optimización de Docker, que incluirá la definición de los Dockerfiles, los cuales especifican la configuración y los requisitos de cada contenedor. Además, se realizará la gestión de los registros de Docker, donde se almacenan las imágenes de los contenedores, para garantizar su disponibilidad y seguridad.

En cuanto a Kubernetes, se realizará la configuración y la implementación de los clústeres, que son conjuntos de nodos que ejecutan y gestionan los contenedores de manera distribuida. Se definirán los archivos de configuración de Kubernetes, como los despliegues, los servicios y las configuraciones de red, para asegurar un despliegue eficiente y escalable de los contenedores.

En la fase de **monitorización**, se trabajará en la implementación de la suite EKF (Elasticsearch, Kibana y Fluend) para la gestión de registros y análisis de datos en tiempo real. Elasticsearch se encargará de almacenar e indexar los registros generados por los microservicios desplegados en Kubernetes, mientras que Fluend será quien recopile y filtre los datos de registro. Por último, Kibana se utilizará como interfaz de visualización y análisis,

proporcionando paneles y gráficos intuitivos para monitorear estado y rendimiento del sistema. Con la integración de EKF, se obtendrá una visión completa de las métricas y registros de la aplicación, permitiendo una supervisión eficiente y la detección temprana de problemas para garantizar un entorno estable y confiable.

Para finalizar, en la fase de **retroalimentación** se documentarán las mejoras a futuro que se deberían tener en cuenta después de haber realizado lo anteriormente descrito, consideraciones de gran valor para trabajos subsiguientes que traten sobre la implementación del ciclo DevOps sobre desarrollos existentes.

Una vez terminadas las seis fases a tratar del ciclo DevOps, se realizará una comparación del despliegue de la infraestructura en las 3 nubes propuestas que fueron Azure, AWS y DigitalOcean, ventajas, desventajas, servicios prestados por cada proveedor vs costes medios mensuales de la implementación de la plataforma en cada una de ellas.

4. Implementación de las fases operativas del ciclo DevOps en el proyecto de software SINET

A continuación, se iniciará la implementación de las fases del ciclo DevOps tomando como insumo base el proyecto de software SINET para el manejo de Activos informáticos, aquí se explicará la configuración, problemas y soluciones que se fueron presentando en cada una de las fases.

4.1. Desarrollo

Como se observó en la fase de planificación y hecho el análisis del software, era necesario hacer unos ajustes para que este se adapte al despliegue en Kubernetes, dicho ajuste tiene que ver con el Health Check de cada uno de los microservicios los cuales no la tenían implementada.

El Health Check (o healtcheck) es una práctica común en la arquitectura de microservicios y se utiliza para verificar el estado y la salud de un servicio en un entorno orquestado como Kubernetes (k8s).

Al utilizar sondas de salud en Kubernetes, el orquestador puede tomar decisiones inteligentes basadas en el estado de los microservicios. Por ejemplo, si un servicio falla repetidamente la sonda Liveness de Kubernetes puede reiniciar automáticamente el pod o incluso redirigir el tráfico hacia otros pods o réplicas saludables.

El Health Check se realiza a través de comprobaciones regulares enviadas a los contenedores o pods mediante sondas de salud (health probes). Estas sondas se configuran con cierta periodicidad y ejecutan una solicitud específica al servicio para verificar su estado.

Existen dos tipos principales de sondas de salud utilizadas en Kubernetes:

Probe de Liveness (sonda de disponibilidad): Verifica si el servicio está en funcionamiento y debe seguir respondiendo a las solicitudes. Si el Liveness Probe falla, Kubernetes considera el servicio como no disponible y puede reiniciarlo o tomar medidas correctivas. Para el caso específico de los microservicios de SINET, estamos implementando un healtchek para esta sonda con un recurso basado en un método get.

Probe de Readiness (sonda de preparación): Verifica si el servicio está listo para recibir tráfico y servir solicitudes. Si el readiness Probe falla, Kubernetes deja de enviar tráfico al servicio y lo considera no listo. Para la verificación del readiness, no se realizará ningún ajuste en los microservicios ya que este se hará con la simple verificación de apertura de puerto TCP.

Actualmente SINET consta de un backend distribuido en 10 microservicios transaccionales desarrollados en java con el framework Spring Boot, los cuales se nombran a continuación:

- sinet_admin_contratacion
- sinet_admin_usuarios
- sinet_admin_personas
- sient_admin_empleados
- sient_admin_politicas_acceso
- sinet_admin_redes
- sinet_admin_soportes
- sinet_admin_cargos
- sinet_admin_componentes
- sinet_admin_activos

Todos estos microservicios deben ser ajustados adicionando un recurso para la verificación de su estado.

Para esto fue necesario la implementación de dos clases en cada uno de ellos, un Controller el cual se encarga de entregar la respuesta, esta tiene dos estados. un status 200 si el servicio se encuentra operativo o un status diferente si falla. En la imagen a continuación podemos observar la implementación de la clase en el servicio sinet_admin_activos.

Figura 1 *Controller para recurso /health*

```java
package co.com.sinet.controllers;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
@RequestMapping("api/v1")
public class HealthCheckController {

    @Autowired
    private HealthCheckService healthCheckService;

    @GetMapping("/health")
    public ResponseEntity<String> healthCheck() {
        boolean isDatabaseHealthy = healthCheckService.isDatabaseHealthy();
        if (isDatabaseHealthy) {
            return ResponseEntity.ok("Servicio saludable");
        } else {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error en la base de datos");
        }
    }
}
```

Fuente(propia)

El estado se obtiene a través de una segunda clase que es un Service, el cual simplemente lo que hace es una conexión a la base de datos, ejecutando un query y retorna un true si este es satisfactorio; en caso de que no por medio de una excepción controlada se entrega un false dando a entender que existen problemas con el servicio o la base de datos. En la siguiente imagen se puede observar la implementación de la clase:

Figura 2 *Service para la verificación del healtcheck del servicio*

```java
package co.com.sinet.services;

import javax.persistence.EntityManager;
import javax.persistence.Query;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class HealthCheckService {
    @Autowired
    private EntityManager entityManager;

    public boolean isDatabaseHealthy() {
        try {
            Query query = entityManager.createNativeQuery("SELECT 1");
            Object result = query.getSingleResult();

            // Verifica que se haya obtenido algún resultado
            return result != null;
        } catch (Exception e) {
            // Ocurrió un error al conectar o consultar la base de datos
            e.printStackTrace();
            return false;
        }
    }
}
```

Fuente(propia)

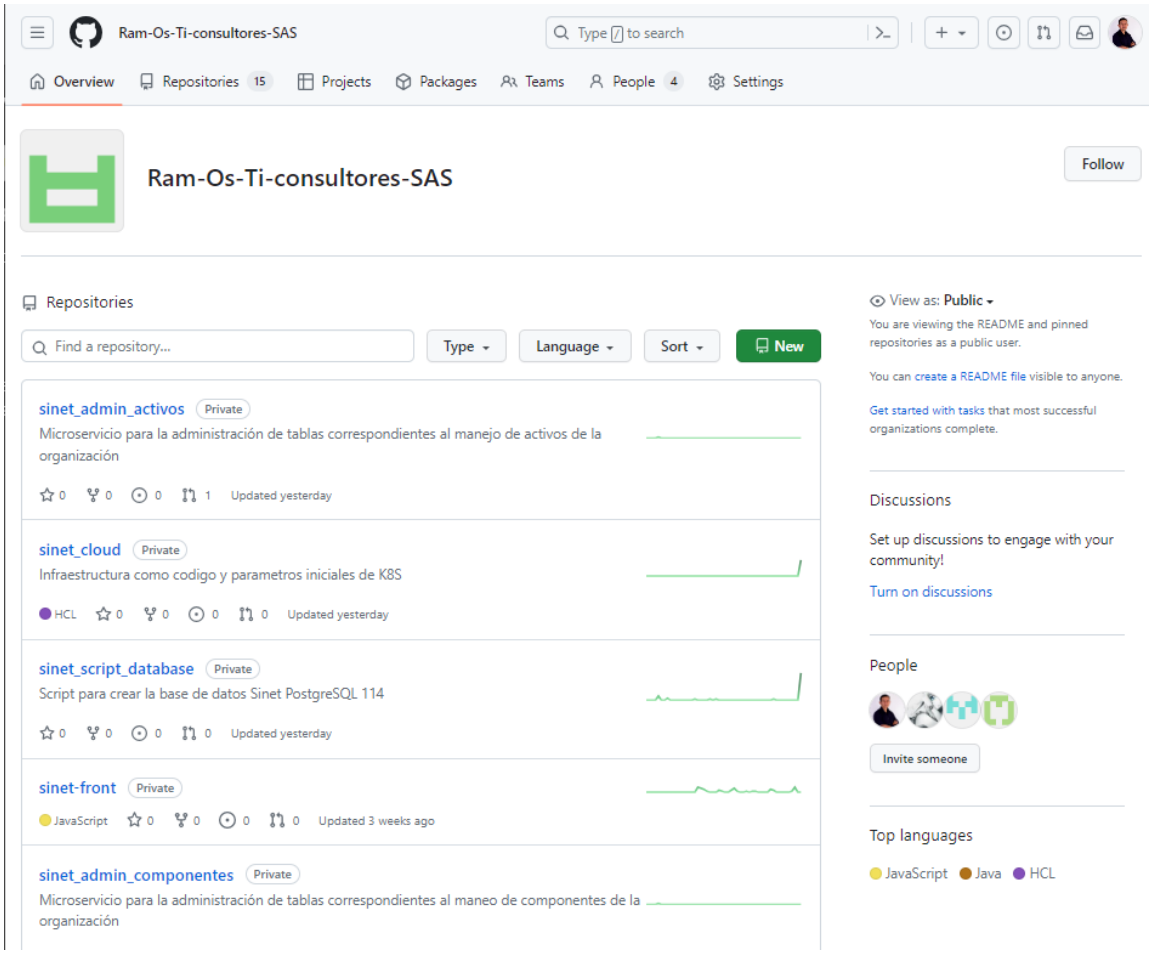
4.2. Integración

En esta fase, se realizan tareas relacionadas al versionamiento de las fuentes de SINET, para este caso ellos fueron estructurados 11 proyectos de microservicios y 2 dos proyectos adicionales los cuales se mencionarán más adelante.

Para el manejo de esta fase se utilizó el servicio de GitHub, donde se crearon las cuentas de usuario par el proyecto y una organización de nombre Ram-Os-Ti-consultores-SAS.

Una vez creada la organización se estructuraron los proyectos anteriormente mencionados, y se enlazaron las cuentas a la organización, como se observa en la siguiente imagen.

Figura 3 *Configuración del proyecto en GitHub*



Fuente(propia)

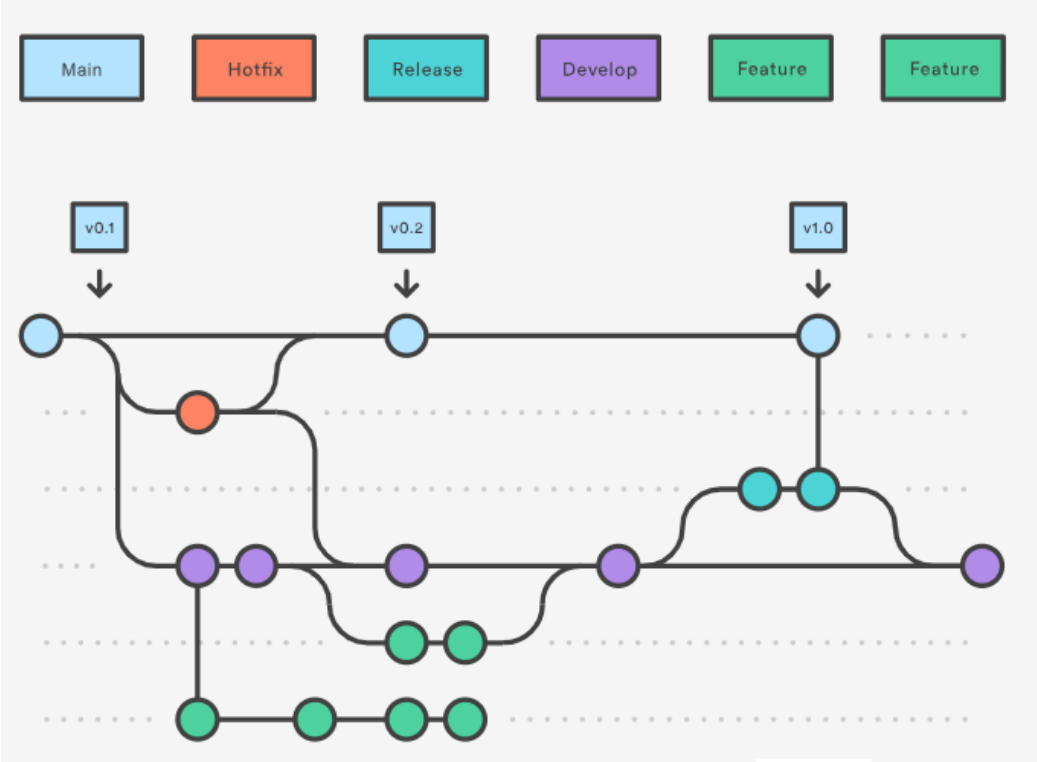
Finalmente se estableció como política de flujo de trabajo gitflow en el manejo de ramas, adicional a ello, se decide no utilizar la rama main como rama por defecto para producción, si no se crea una rama especifica con el nombre de producción, para este trabajo se crearán

ramas productivas para cada una de las nubes en las que se va a desplegar los microservicios y estas fueron nombradas de la siguiente manera:

- produccion
- produccion_aws
- produccion_az
- produccion_do

La razón por la que se descarta la rama main, es debido a que, por experiencias pasadas, los desarrolladores suelen confundirla con la rama base para iniciar sus desarrollos, y en flujo gitflow, la rama base siempre es desarrollo, por tanto, si al clonar la rama main, es más fácil percatarse del inconveniente ya que esta no entregara código fuente para trabajar.

Figura 4 *Flujo de trabajo gitflow*



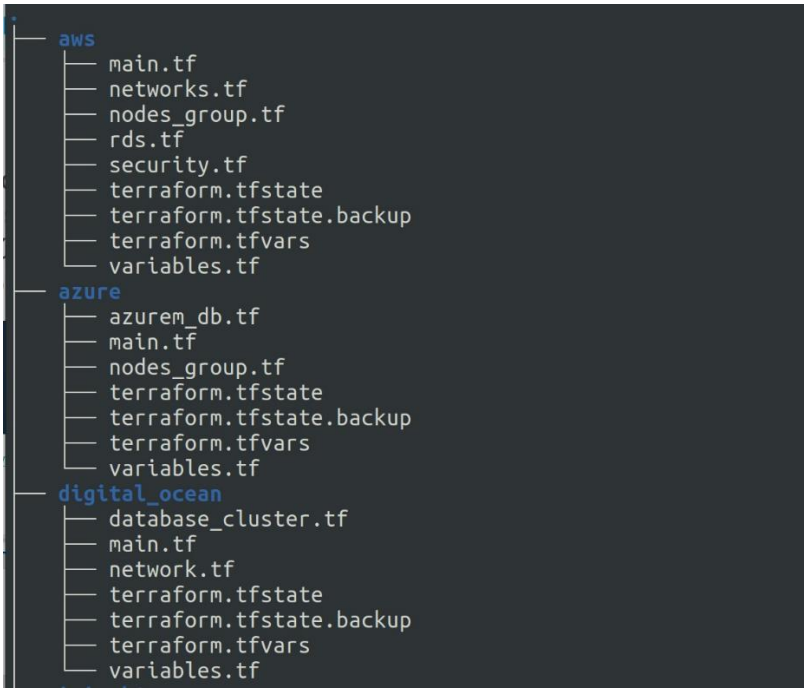
Fuente: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

Aparte de los proyectos anteriormente mencionados, SINET tiene dos proyectos que no son fuentes para la creación de microservicios, pero son necesarios para su correcto despliegue y se explican detalladamente a continuación:

sinet_scripts_database: En este proyecto se encuentra el Backus de la estructura de tablas de la base de datos y los datos semilla necesarios para que los microservicios funcionen correctamente.

sinet_cloud: En este proyecto, se almacenan los archivos de configuración necesarios para el despliegue de la infraestructura como código utilizando Terraform. Estos archivos contienen las definiciones y configuraciones que describen los recursos de infraestructura requeridos, como servidores, redes, bases de datos, almacenamiento, entre otros, para ello se creó la estructura de directorios y archivos:

Figura 5 *Estructura de directorios proyecto sinet_cloud*



Fuente(propia)

Como se puede observar existe una carpeta con el nombre de cada una de las nubes en la que se desplegara la infraestructura necesaria para la puesta en funcionamiento de los microservicios. Si bien es cierto que se puede desplegar en un mismo grupo de archivos todas las nubes que el proyecto requiera, esta no es la idea para este, sino más bien tener una configuración específica para desplegar en la nube que el cliente que contrate SINET lo

requiera, es por ello por lo que se separó configuraciones de Terraform específicas para cada nube.

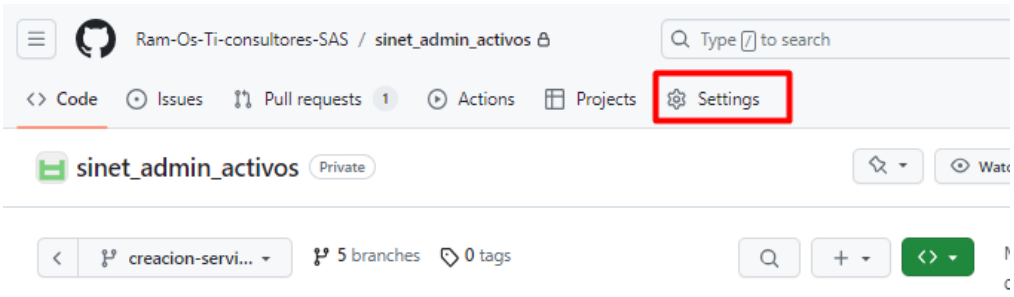
4.3. Despliegue

En esta fase se trata todo lo relacionado con integración y entrega continua, partiendo desde las configuraciones iniciales en git, pasando por la creación de los workflows con todas las tareas relacionadas hasta la finalización con el despliegue de los pods en Kubernetes.

4.3.1. Configuraciones Iniciales

Para poder configurar un flujo de trabajo con GitHub Actions, es necesario que el usuario tenga permisos de owner en la cuenta de la organización, para así poder acceder a la opción settings del proyecto a configurar como se observa en la siguiente imagen

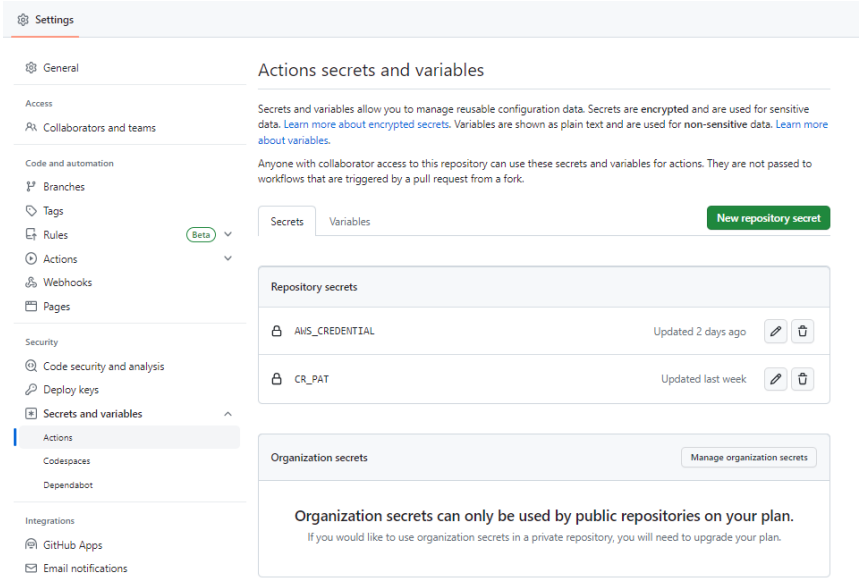
Figura 6 *Settings del proyecto sinet_admin_activos*



Fuente (propia)

Dentro de las opciones de settings se encuentra secretos y variables, donde se deben almacenar como su nombre lo dice todos los datos sensibles y variables que necesitan los workflows para poder funcionar, a continuación, se relaciona una imagen con las opciones mencionadas.

Figura 7 *Secretos y variables GitHub*



Fuente (Propia)

4.3.2. Workflows

Los workflows o flujos de trabajo son archivos en los cuales se definen los pipelines que GitHub seguirá secuencialmente para conseguir el despliegue del artefacto configurado.

Para configurar los workflows se debe crear un directorio llamado .GitHub en la raíz del proyecto, y dentro de este directorio se debe crear un directorio adicional llamado workflows, una vez creados estos directorios, dentro de workflows se procede a crear archivos en formato yaml en los cuales se definen los pipelines los cuales se explican en el siguiente apartado

4.3.2.1. Configuración de un archivo workflow

Un archivo de workflow de GitHub Actions es un archivo de configuración en formato YAML que define una serie de pasos y acciones a ejecutar en respuesta a eventos específicos en un repositorio de GitHub. A continuación, se nombran las etiquetas básicas y como también una particularidad muy interesante que tienen las actions que es la reutilización de componentes elaborados por terceros para la ejecución de algunas acciones como por ejemplo hacer un checkout del proyecto.

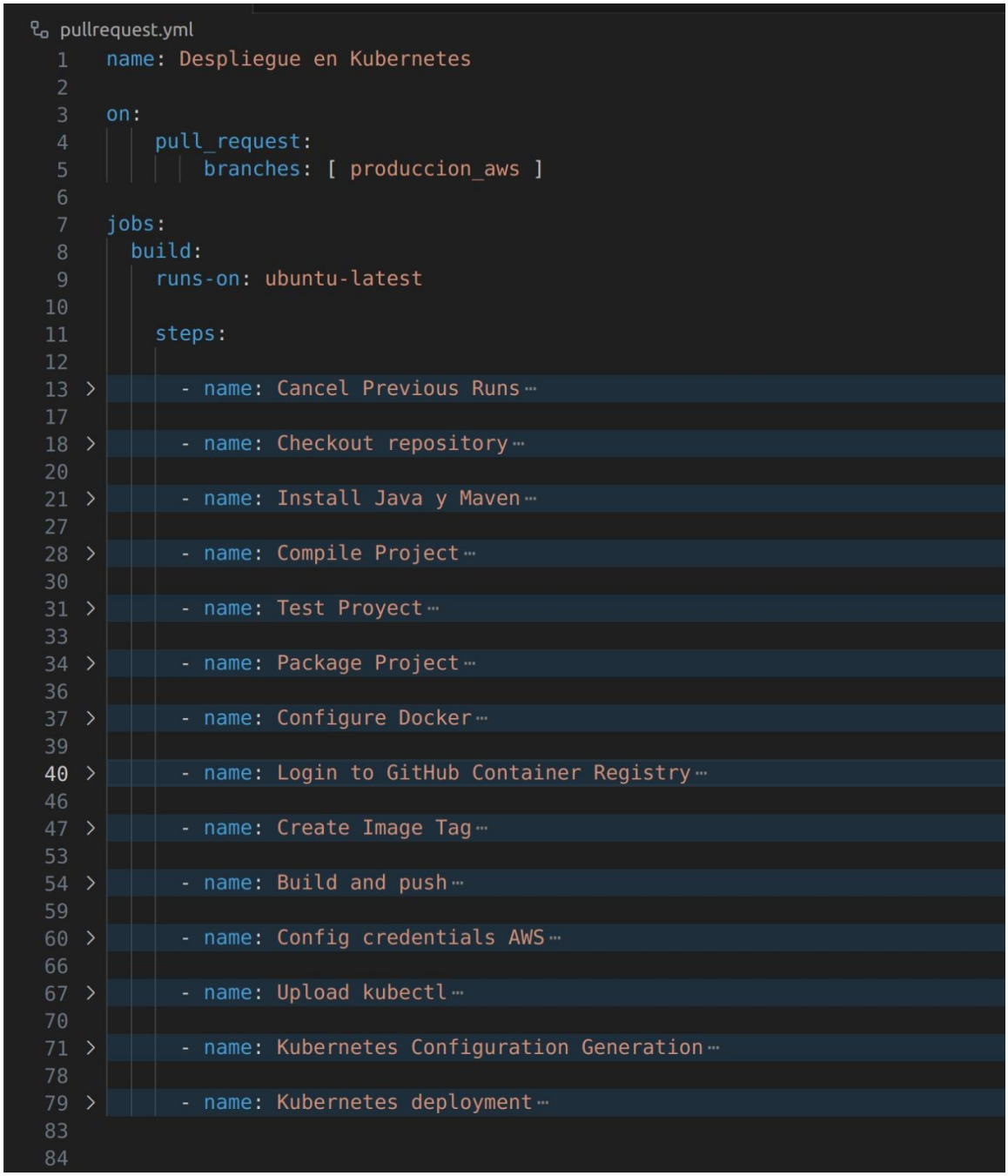
Las principales etiquetas de un archivo de workflow de GitHub Actions son las siguientes:

- name: Permite asignar un nombre descriptivo al workflow.

- on: Define los eventos que desencadenarán la ejecución del workflow, como confirmaciones de cambios (push) o solicitudes de extracción (pull request).
- jobs: Representa una o más tareas que se ejecutarán en paralelo dentro del workflow.
- steps: Define los pasos individuales dentro de un job. Cada paso representa una acción o tarea que se ejecutará, como compilar el código, ejecutar pruebas o realizar un despliegue.
- La palabra clave "use" es especialmente importante para reutilizar el trabajo de terceros en GitHub Actions. Con esta etiqueta, se puede especificar una acción existente para ejecutar dentro de tu workflow. Es posible utilizar acciones proporcionadas por la comunidad de GitHub Actions o crear acciones propias personalizadas. Estas acciones encapsulan una tarea específica y pueden ser utilizadas en múltiples workflows.

En la siguiente imagen, se relaciona la imagen de un workflow de forma general, del proyecto sinet_admin_activos el cual se encarga de desplegar un deployment de Kubernetes para la nube de AWS, donde se ira explicando cada uno de los pasos que fueron necesarios para conseguir un despliegue continuo.

Figura 8 *Despliegue en Kubernetes de un proyecto*



```
pullrequest.yml
name: Despliegue en Kubernetes

on:
    pull_request:
        branches: [ produccion_aws ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:

      - name: Cancel Previous Runs ⋯

      - name: Checkout repository ⋯

      - name: Install Java y Maven ⋯

      - name: Compile Project ⋯

      - name: Test Proyect ⋯

      - name: Package Project ⋯

      - name: Configure Docker ⋯

      - name: Login to GitHub Container Registry ⋯

      - name: Create Image Tag ⋯

      - name: Build and push ⋯

      - name: Config credentials AWS ⋯

      - name: Upload kubectl ⋯

      - name: Kubernetes Configuration Generation ⋯

      - name: Kubernetes deployment ⋯
```

Fuente (Propia)

La configuración del archivo deployment solamente se ejecuta cuando se realiza un pull request hacia la rama produccion_aws, no se tiene configurado acción de push debido a que las ramas principales están bloqueadas para poder ejecutar esta acción directamente por los usuarios.

Luego definimos que para la ejecución se realizara sobre una máquina virtual Ubuntu de la última versión disponible

Posteriormente se definen el paso a paso del pipeline como lo explicamos a continuación:

- **Cancel Previews Run:** A continuación, observamos la configuración de esta acción, la cual es la encargada de cancelar una ejecución un flujo de trabajo en GitHub Actions que se esté ejecutando en el momento que se envía una nueva ejecución del mismo pipeline, las ejecuciones anteriores se cancelan con el fin de evitar conflictos o duplicación de trabajo. La acción styfle/cancel-workflow-action simplifica este proceso al cancelar automáticamente las ejecuciones anteriores del mismo flujo de trabajo en el repositorio.

  La acción styfle/cancel-workflow-action se configura proporcionando un token de acceso válido ${{ github.token }} para autenticarse y acceder a la API de GitHub, utilizando este token, la acción busca y cancela las ejecuciones anteriores del flujo de trabajo basándose en su identificador.

  Para esta acción se puede observar una de las particularidades y a la vez ventajas de GitHub Actions, que es la reutilización de acciones de terceros, las cuales se pueden buscar en el Marketplace de GitHub y ayudan a simplificar el trabajo con la reutilización de scripts elaborados previamente.

Figura 9 *Cancel Previews Run*

Fuente(Propia)

- **Checkout repository:** Esta acción es muy sencilla, simplemente de lo que se encarga es de realizar el checkout del repositorio sobre la máquina virtual utilizando una acción de GitHub.

Figura 10 *Checkout del Repositorio*

Fuente(Propia)

- **Install Java and Maven:** Como su nombre lo dice, esta acción se encarga de instalar en la máquina virtual java y maven, necesarios para poder compilar el proyecto ya que como se mencionó anteriormente, este fue desarrollado con el framework Spring Boot y requiere de java 11 para su funcionamiento.

Figura 11 Instalación de Java y Maven

Fuente (Propia)

- **Compile Project:** Esta acción se encarga de limpiar el proyecto de archivos generados previamente y compilar el código fuente, generando los archivos .class
- **Test Proyect:** Esta acción se encarga de correr las pruebas automatizadas del proyecto, esto permite verificar que el código del proyecto funcione correctamente según las especificaciones de los casos de prueba establecidos previamente.
- **Package Project:** Una vez el proyecto ha sido compilado y las pruebas unitarias corrieron de forma satisfactoria, se procede a empaquetar el proyecto en un artefacto .jar, para eso se utiliza esta acción con el comando mvn package.

Figura 12: *Acciones de compilación, pruebas y empaquetamiento*

```
27
28        - name: Compile Project
29          run: mvn clean compile
30
31        - name: Test Proyect
32          run: mvn test
33
34        - name: Package Project
35          run: mvn package
36
```

Fuente (Propia)

- **Configure Docker:** Esta acción se utiliza para configurar el entorno Docker en un flujo de trabajo de GitHub Actions, es decir instala las herramientas y realiza las configuraciones necesarias para interactuar con Docker durante el proceso de construcción y publicación de imágenes de Docker.
- **Login to GitHub Container Registry:** Esta acción se utiliza para iniciar sesión en el Registro de Contenedores de GitHub (GitHub Container Registry). Realizando la autenticación al flujo de trabajo de GitHub Actions y así poder interactuar con el repositorio; se proporciona el nombre del registro (en este caso, ghcr.io) y las credenciales de inicio de sesión (nombre de usuario y contraseña) que se obtienen de las variables de entorno y secretos de GitHub. Al autenticarse correctamente, el flujo de trabajo puede acceder y manipular las imágenes del registro durante el proceso de construcción y publicación de contenedores.

Figura 13 *Configuración y autenticación con el servidor de registro de contenedores*

```
37        - name: Configure Docker
38          uses: docker/setup-buildx-action@v1
39
40        - name: Login to GitHub Container Registry
41          uses: docker/login-action@v1
42          with:
43              registry: ghcr.io
44              username: ${{ vars.USERNAME }}
45              password: ${{ secrets.CR_PAT }}
```

Fuente (Propia)

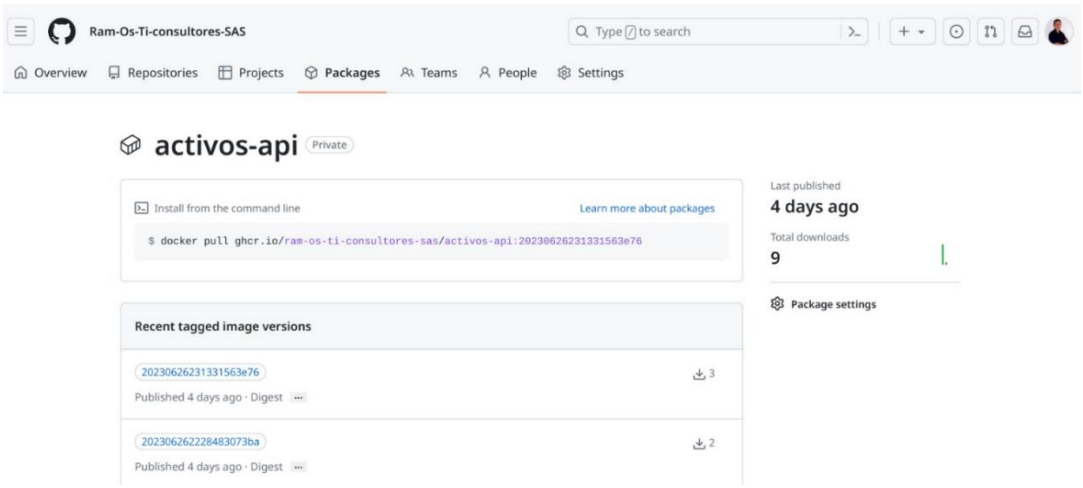
- **Create Image Tag:** Esta acción se encarga por medio de un juego de variables de entorno definir el nombre de la etiqueta de la versión del contenedor que se subirá al servidor ghcr, para ello se tiene en cuenta la fecha del sistema como el del hash de confirmación de git para crear un SHORT_SHA luego se concatena la variable de la fecha TIMESTAMP con el SHORT_SHA y este valor se asigna sobre una variable final llamada SNAPSHOT_TAG, con esto se obtiene una etiqueta única del contenedor, útil para identificar y rastrear versiones específicas de la imagen.
- **Build and push:** Esta acción se encarga de crear y enviar la imagen docker al Registro de Contenedores de GitHub (GitHub Container Registry). En el script, se exporta la variable de entorno IMAGE_GHCR con la ruta de la imagen en el formato "ghcr.io/${vars.ORGANIZACION}/activos-api:{env.SNAPSHOT_TAG}". Luego, se utiliza el comando "docker build" para construir la imagen a partir del contexto actual y se le asigna la etiqueta generada en la acción anterior. Posteriormente con el comando "docker push" para enviar la imagen al Registro de Contenedores de GitHub utilizando la ruta almacenada en la variable IMAGE_GHCR. Esta acción facilita el proceso de compilación y distribución de la imagen de contenedor al registro correspondiente. A continuación, podemos observar la configuración de las dos acciones anteriores y una imagen de verificación de las imágenes en ghrc.io.

Figura 14 *Definición de etiqueta de versionamiento y exportación de la imagen a ghrc.io*

```
47      - name: Create Image Tag
48        run: |
49            TIMESTAMP=`date +%Y%m%d%H%M%S`
50            SHORT_SHA=$(echo "${GITHUB_SHA}" | cut -c1-6)
51            SNAPSHOT_TAG="${TIMESTAMP}${SHORT_SHA}"
52            echo "SNAPSHOT_TAG=$SNAPSHOT_TAG" >> $GITHUB_ENV
53
54      - name: Build and push
55        run: |
56            export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
57            docker build . --tag ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
58            docker push $IMAGE_GHCR
59
```

Fuente (Propia)

Figura 15 *Verificación de importación de imagen a ghcr.io*



Fuente (Propia)

Las siguientes dos acciones son específicas para cada nube, y se encargan de la configuración de credenciales en la máquina virtual como de la actualización del archivo .kube/config para poder acceder a la instancia de Kubernetes y realizar la aplicación de los archivos deployment. A continuación, las explicaremos para acceder a la nube de AWS, pero en los anexos se puede encontrar los archivos yaml con la configuración de Azure y Digital Ocean.

- **Config credentials AWS:** para acceder a los servicios de AWS por medio del comando AWS, es necesario crear el directorio . AWS en el home del usuario y posteriormente ahí crear un archivo llamado credentias con el contenido de las credenciales proporcionadas por AWS. En esta acción se definió el contenido de las credenciales como un secreto el cual es accedido desde las Actions con la etiqueta ${{secrets.AWS_CREDENTIAS}}, adicional a ello es necesario especificar la región a la que nos vamos a conectar, esta se obtiene a través de una variable llamada ${{vars.AWS_REGION}}.
- **Update kubectl AWS:** Una vez configuradas las credenciales de AWS en el paso anterior, se procede a realizar la actualización del .kube/config para poder acceder al clúster de Kubernetes por medio del comando kubectl, esto se hace con la instrucción aws eks update-config –name NOMBRE_CLUSTER –region REGION.

Figura 16 *Configuración de credenciales de AWS y clúster kubernetes*

```
60      - name: Config credentials AWS
61        run: |
62            mkdir -p ~/.aws
63            touch ~/.aws/credentials
64            echo "${{ secrets.AWS_CREDENTIAL }}" > ~/.aws/credentials
65            echo "region=${{ vars.AWS_REGION }}" >> ~/.aws/credentials
66
67      - name: Update kubectl AWS
68        run: |
69            aws eks update-kubeconfig --name ${{ vars.AWS_CLUSTER_NAME }} --region ${{ vars.AWS_REGION }}
70
```

Fuente(Propia)

Una vez actualizado el archivo de configuración de Kubernetes en la máquina virtual se procede a preparar los archivos deployment de Kubernetes, la cual se detalla a continuación:

- **Kubernetes Configuration Generation:** este paso consiste en reemplazar variables de entorno con sus valores correspondientes, en este caso son 3 variables de entorno necesarios que son:

  IMAGE_GHCR: Esta variable nos entrega la URL del servicio GHCR (registro de contenedor de imágenes similar a DockerHub, pero de GitHub)

  URL_RDS: Contiene la URL de la base de datos a la que apuntan los microservicios y se obtiene a partir del nombre que se le dio a la instancia en el momento de su creación

  URL_DATABASE: Esta variable contiene la URL de conexión a la base de datos indicando el driver y el nombre de la base de datos más la URL del servicio RDS obtenida anteriormente.

  Una vez se tiene las variables se reemplazan en el archivo deployment original que se encuentra en los repositorios de cada uno de los proyectos, esto se hace en comando envsubtr y se crea un nuevo archivo el cual se utilizara para desplegar los servicios en Kubernetes. A continuación, se adjunta la imagen de la parametrización mencionada:

Figura 17 *Kubernetes Configuration Generation*

```
70
71      - name: Kubernetes Configuration Generation
72        run: |
73            export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
74            echo $IMAGE_GHCR
75            export URL_RDS=$(aws rds describe-db-instances --db-instance-identifier postgresql-sinet2022 --query 'DBInstances[0].Endpoint.Address' --output text)
76            export URL_DATABASE=jdbc:postgresql://$URL_RDS/sinet2022
77            envsubst < ~/./work/sinet_admin_activos/sinet_admin_activos/deployments/activos-deployment.yaml > ~/activos-deployment-ok.yaml
78
```

Fuente (Propia)

- **Kubernetes Deployment:** Terminados todos los plazos anteriores, solo falta realizar el deployment de los servicios de Kubernetes configurados en el archivo yaml anterior, para ello solo es necesario correr la instrucción kubectl apply -f y se desplegara la configuración realizada en el clúster. En la siguiente figura podemos observar esta última configuración:

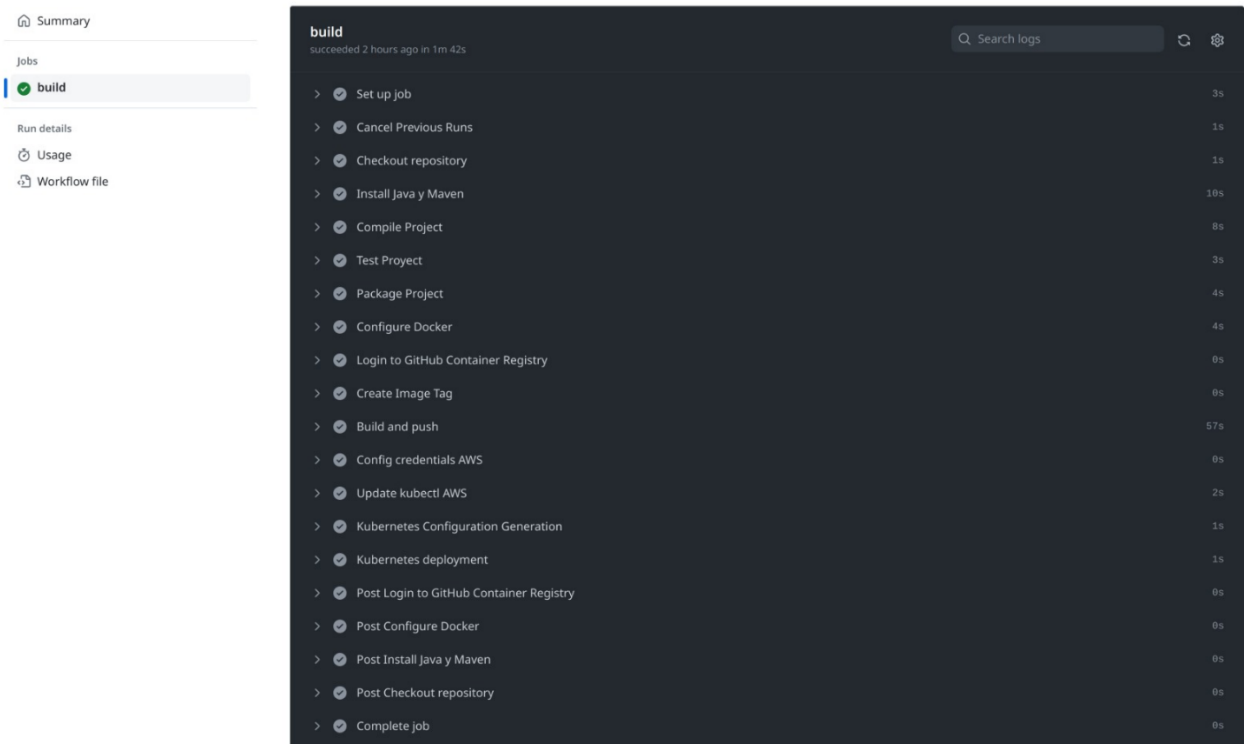
Figura 18 *Kubernetes deployment*

Fuente (Propia)

4.3.2.2. Ejecución Git Actions

Una vez configurado el pipeline solo falta ejecutarlo, para ello simplemente es necesario realizar un Pull Request de cualquier rama haca la rama a la que el workflow fue enlazado, en este caso produccion_aws, esto dispararía el workflow, para empezar a realizar el despliegue como se evidencia en la siguiente imagen:

Figura 19 *Ejecución workflow GitHub Actions*



Fuente (Propia)

Con lo anterior finalizamos esta fase del ciclo DevOps, indicando paso a paso desde su configuración hasta su ejecución logrando exitosamente una integración y despliegue continuo totalmente operativo con GitHub Actions, aprovechando las bondades que nos da este servicio ampliamente utilizado el cual para lo que necesitamos específicamente aquí no represento costos adicionales.

4.4. Operación

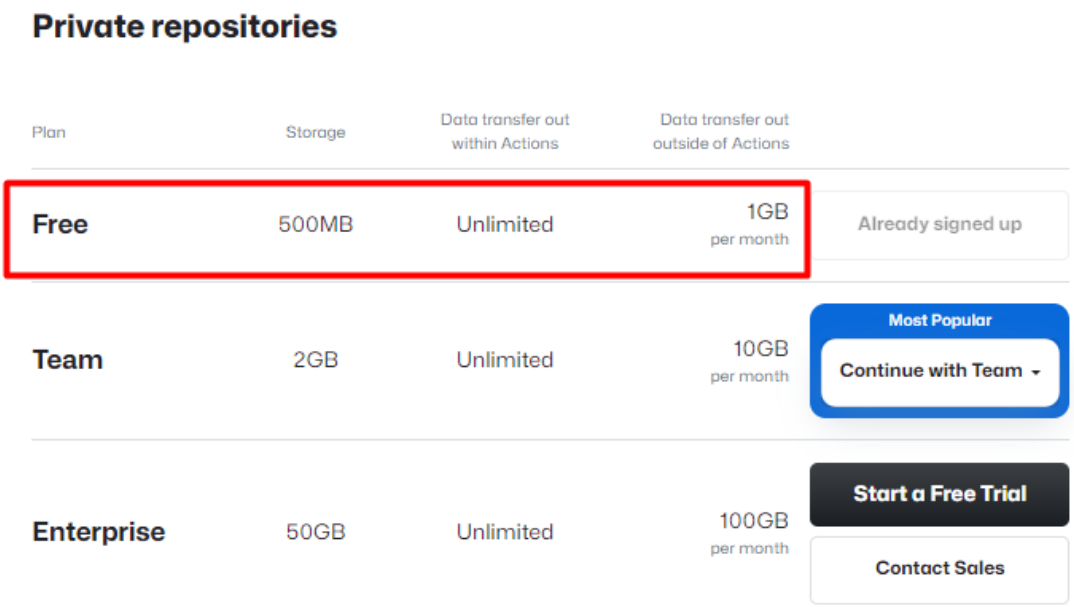
En la fase de operación se explicará paso a paso la configuración de dos herramientas muy importantes que son docker y Kubernetes, las cuales trabajan de la mano y a pesar de que las nombramos en la fase anterior, en esta fase daremos el detalle de todo lo relacionado a su configuración

4.4.1. Docker

En esta sección se pondrá especial cuidado en lo referente a la generación de una imagen optimizada en tamaño, y la razón es muy simple, costos de almacenamiento y velocidad de despliegue del workflow de GitHub Actions.

GitHub permite el almacenamiento de imágenes en su servicio ghcr.io (GitHub Container Registry), el cual, si bien es gratuito, este tiene limitantes en espacio y transmisión como se observa en la siguiente imagen:

Figura 20 *Precios por almacenamiento ghcr.io*



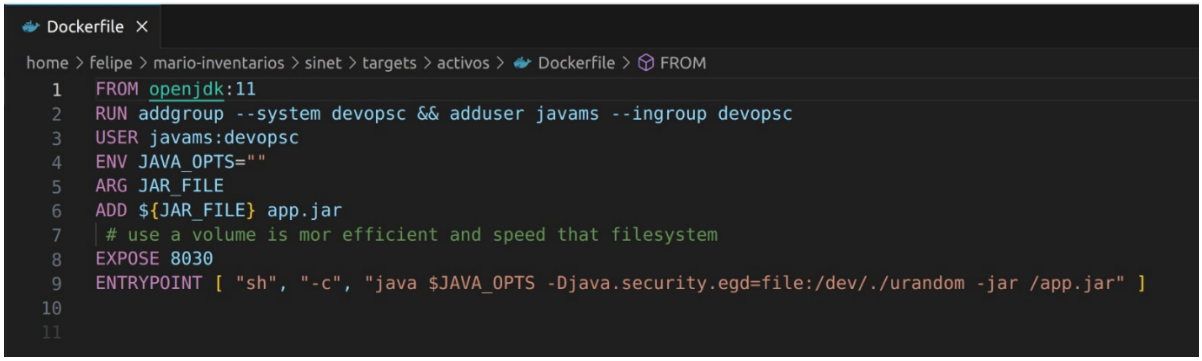
Fuente (https://github.com/features/packages)

Trabajando con una imagen por defecto el peso de esta es de 750 mb, un tamaño supremamente grande para un microservicio, lo cual repercute en lo anteriormente mencionado.

¿Pero porque repercute también en la velocidad de despliegue de GitHub Actions?, la respuesta es simple, porque en esto intervienen 3 pasos a los cuales repercute negativamente por su tamaño que son, la creación de la imagen (tiempo de procesamiento), el upload al servicio ghcr.io (tiempo de transmisión) y el download desde Kubernetes de la imagen recién creada (tiempo de transmisión), hospedada en ghcr.io.

A continuación, se observa el Dockerfile con una imagen base de java sin optimización, que se venía trabajando en el proyecto base.

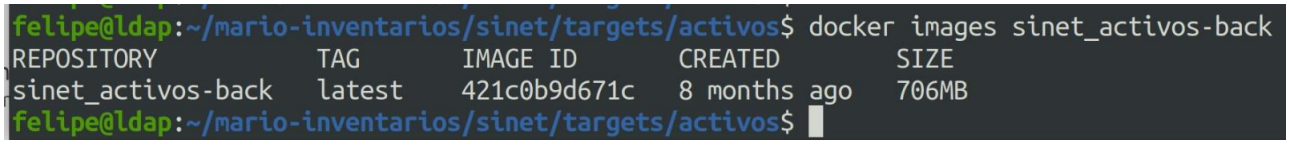
Figura 21 *Dockerfile sin optimizar*



Fuente (Propia)

Realizando la comprobación del tamaño de la imagen podemos observar que esta tiene un peso de 706MB.

Figura 22 *Verificación del tamaño de la imagen standard*



Fuente (Propia)

A continuación, observamos el archivo Dockerfile optimizado para generar un tamaño pequeño de la imagen final siguiendo las siguientes prácticas para lograr un tamaño adecuado.

Figura 23 Dockerfile optimizado

Fuente (Propia)

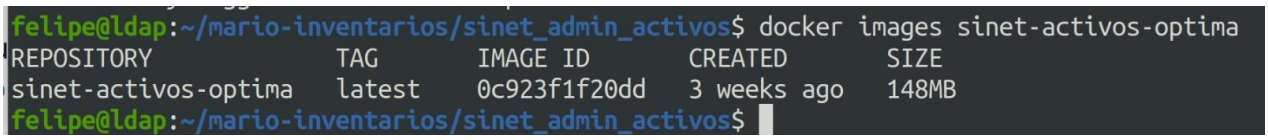
Entre las modificaciones más importantes que se le realizaron al Dockerfile tenemos las siguientes:

- Uso de Alpine Linux: El Dockerfile comienza seleccionando una imagen base de Alpine Linux, que es una distribución de Linux ligera. Alpine Linux tiene un tamaño de imagen pequeño y es conocido por su enfoque en la eficiencia y la seguridad.

- Selección de una versión específica de Corretto JDK: El primer paso en el Dockerfile es seleccionar una versión específica del JDK Corretto de Amazon, que ya es una imagen optimizada y liviana. Utilizar una versión específica ayuda a garantizar la consistencia y predecibilidad en el entorno de desarrollo.
- Creación de una imagen JRE personalizada: El Dockerfile utiliza jlink, esta herramienta se implementó en java a partir de la versión 9 y sirve para crear una imagen JRE personalizada, permite crear una imagen JRE reducida y específica para la aplicación, incluyendo solo los módulos necesarios. Al utilizar --strip-debug, --no-man-pages, --no-header-files y --compress=2, se eliminan los archivos y componentes innecesarios, lo que reduce significativamente el tamaño de la imagen final.
- Separación en etapas: El Dockerfile utiliza dos etapas para separar la construcción del JRE personalizado y la imagen principal de la aplicación. Esto permite que solo la imagen JRE personalizada se copie en la imagen final, reduciendo así el tamaño general de la imagen. La imagen Alpine Linux base se utiliza solo para la etapa final de la aplicación.

Realizados los ajustes anteriores procedemos a crear la imagen y a verificar su tamaño, como se puede observar a continuación ahora es de un tamaño de 148MB:

Figura 24 *Imagen docker optimizada*



Fuente (Propia)

Con lo anterior concluimos lo referente al uso de docker en la fase de operación, los inconvenientes encontrados y como se resolvieron.

4.4.2. Terraform

Se puede decir que para este proyecto esta es la herramienta más importante en lo referente a Infraestructura como código, ya que esta es la encargada de preparar toda la ambientación necesaria para poder realizar su despliegue.

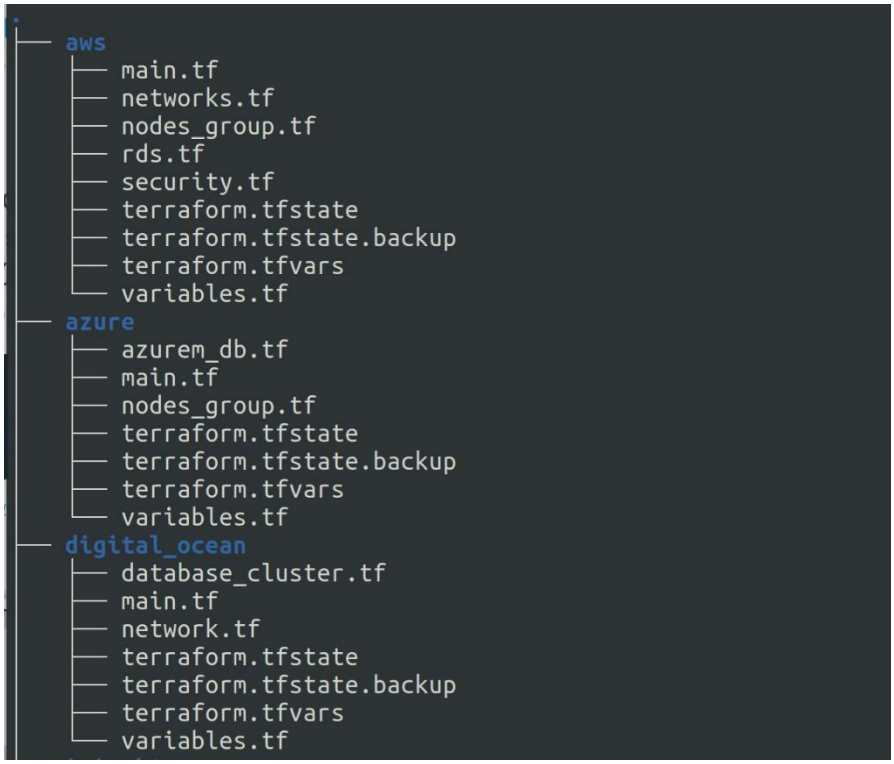
Los montajes que se van a realizar son específicos para las nubes de Amazon, Azure y DigitalOcean, ya que uno de los objetivos de este trabajo es desplegar la misma

infraestructura en las diferentes nubes, para posteriormente realizar comparaciones entre las mismas, tanto en dificultad de montaje, servicios ofrecidos y costos vs servicio de estas.

La parametrización de los archivos de Terraform consiste en la definición de recursos, los cuales son propios para cada proveedor de servicios en la nube. Cada uno de estos recursos está diseñado para trabajar con un determinado servicio o funcionalidad dentro de cada una de ellas.

Todo lo relacionado con infraestructura como código, se definió en un proyecto llamado sinet_cloud y dentro de este proyecto se encuentran carpetas específicas para cada nube, con los archivos propios para su correcto despliegue como se observa en la imagen siguiente:

Figura 25 *Estructura Terraform multi-nube*



Fuente (Propia)

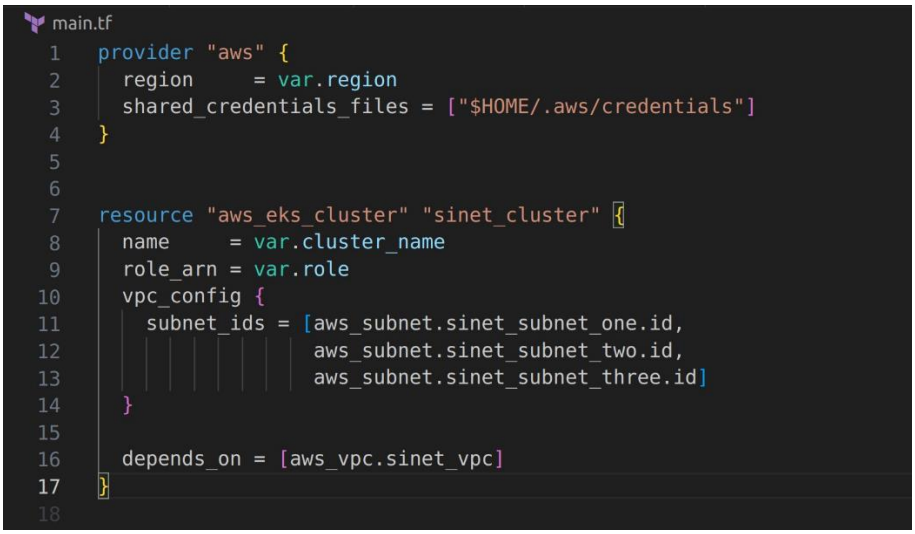
4.4.2.1. AWS

La nube de Amazon es una de las más completas y permite realizar un despliegue granular al nivel que se desee, y a diferencia de las otras dos, la administración de la seguridad de sus servicios está totalmente gobernada por un mismo componente que son los Security Groups.

Para el despliegue de la infraestructura en Terraform se clasifico en 5 archivos, cada uno enfocado a servicios específicos y se explicaran a continuación:

- main.tf: La infraestructura que se va a desplegar en Terraform tiene como servicio principal Kubernetes, por tanto, en este archivo se definió todo lo referente a la creación del servicio EKS (ElasticSearch Kubernetes Service) y las credenciales correspondientes para poder acceder a AWS desde Terraform, a continuación, se puede observar el contenido del archivo donde se observan las configuraciones mencionadas.

Figura 26 *archivo main.tf AWS*

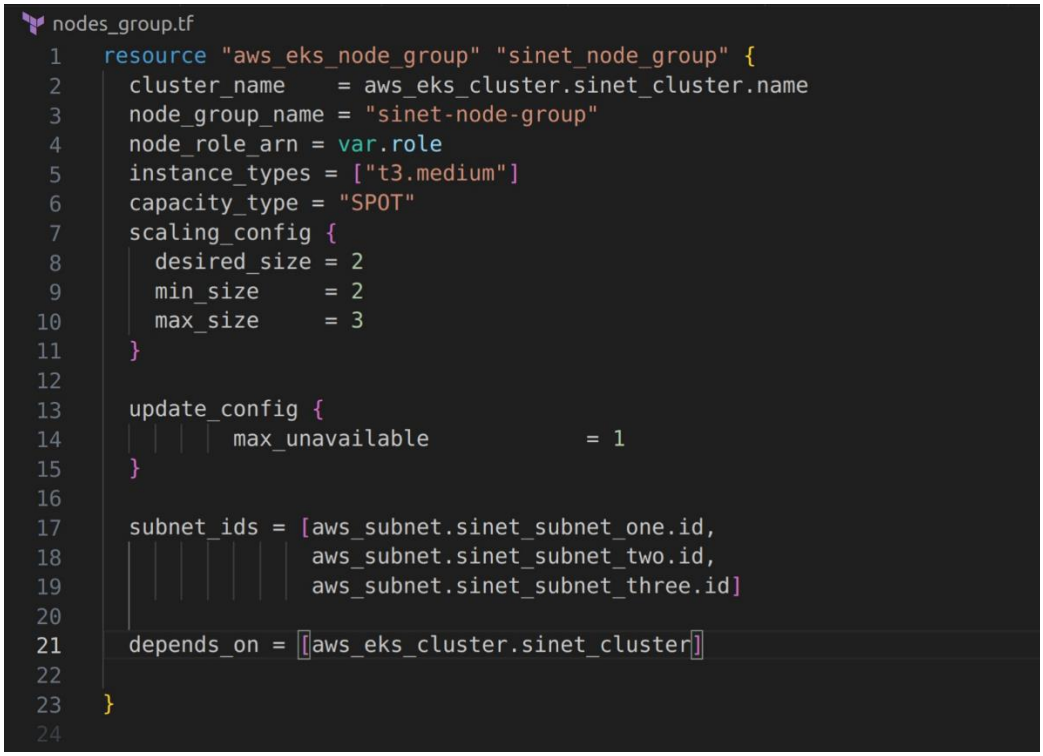


Fuente (Propia)

- nodes_group.tf: En este archivo se encuentra toda la definición para la creación de los nodos de Kubernetes, en él se define el tipo de instancia que trabajara como nodo del clúster, su capacidad y el número de instancias, de resaltar la bandera *capacity_type="SPOT";* en AWS es posible crear nodos del tipo "SPOT", los nodos de este tipo tienen como particularidad que el costo le ofrecen un 60% de descuento sobre el valor de una instancia EC2 normal, esto es debido a que estas se crean en servidores que en el momento no se encuentran asignados a un cliente, por tanto están disponibles para ser utilizados, en el caso de que estos servidores se vendan, AWS le notifica al clúster con dos minutos de anticipación antes de destruir el nodo, tiempo suficiente para que el clúster cree una nueva instancia en otro sitio disponible y el este siga funcionando sin problemas. a continuación, se puede observar el

contenido del archivo donde se observan las configuraciones necesarias para la creación de los nodos del clúster:

Figura 27 *Archivo nodes_group.tf AWS*



Fuente (Propia)

- rds.tf: Elegir un servicio SaaS (Software as a Service) para la base de datos presenta múltiples ventajas para empresas y desarrolladores. En primer lugar, estos servicios ofrecen una administración simplificada, además, al utilizar un servicio gestionado, se ahorra tiempo y recursos valiosos, ya que se evitan tareas como la instalación y el mantenimiento de la infraestructura de la base de datos, permitiendo a los equipos centrarse en el desarrollo de aplicaciones y en las actividades centrales del negocio, transfiriendo la responsabilidad de la operación del servicio al proveedor Cloud.

  En AWS, el servicio que se encarga de ofrecer bases de datos se conoce como RDS (Relational Database Service) y a continuación se relaciona las configuraciones para la creación de dicho servicio desde Terraform:

Figura 28 *Archivo rds.tf AWS*

rds.tf

```
resource "aws_db_subnet_group" "subnet_group_sinet" {
    name        = "grupo de subnets eks rds"
    subnet_ids  = [aws_subnet.sinet_subnet_one.id,
                   aws_subnet.sinet_subnet_two.id,
                   aws_subnet.sinet_subnet_three.id]
}

resource "aws_db_instance" "sinet2022" {
  identifier           = "postgresql-sinet2022"
  allocated_storage    = 10
  engine               = "postgres"
  engine_version       = "14.7"
  instance_class       = "db.t3.micro"
  db_name                 = "sinet2022"
  username             = "emaku"
  password             = var.db_password
  publicly_accessible = false
  db_subnet_group_name      = aws_db_subnet_group.subnet_group_sinet.id
  vpc_security_group_ids    = [aws_security_group.sinet_security.id]
  skip_final_snapshot       = true
  final_snapshot_identifier = "Ignore"

}
```

Fuente (Propia)

- networks.tf: AWS define toda la configuración de sus redes en una VPC (Virtual Private Cloud), en él se configura tanto un rango de direcciones ip para la VPC, el gateway para la salida de los servicios a internet, las tablas de enrutamiento y las subnet, donde generalmente estas se asocian a una zona de disponibilidad diferente para tener redundancia de conexión entre el clúster y los nodos asociados en caso de caída de alguna subnet. Como se mencionó al inicio del tema, esta es una de las grandes ventajas que tiene AWS, ya que en este tipo de servicios se definen parámetros que tienen alcance a toda la infraestructura. A continuación, podemos observar parte del archivo de configuración donde se parametriza lo anteriormente mencionado.

Figura 29 *Archivo networks.tf AWS*

networks.tf

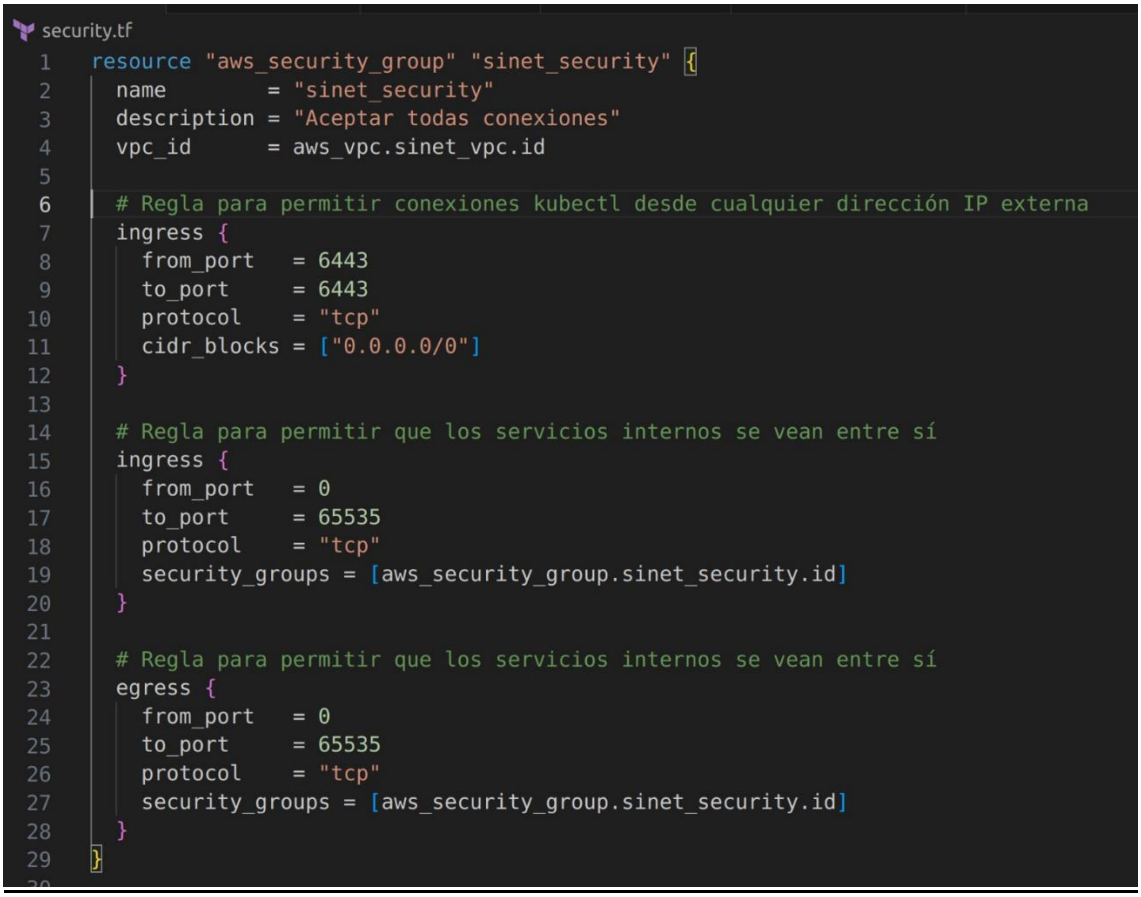
```
resource "aws_vpc" "sinet_vpc" {
 cidr_block           = "192.168.0.0/16"
 enable_dns_hostnames = true
 tags = {
  Name = "SINET VPC"
 }
}

resource "aws_internet_gateway" "gw" {
  vpc_id            = aws_vpc.sinet_vpc.id
  tags              = {
      Name = "Gateway"
  }
}

resource "aws_route_table" "sinet_table" {
  vpc_id        = "${aws_vpc.sinet_vpc.id}"
  route {
    cidr_block  = "0.0.0.0/0"
    gateway_id  = "${aws_internet_gateway.gw.id}"
  }
}

resource "aws_subnet" "sinet_subnet_one" {
  vpc_id                  = aws_vpc.sinet_vpc.id
  cidr_block              = "192.168.10.0/24"
  availability_zone       = "us-east-1a"
  map_public_ip_on_launch = true
  depends_on = [aws_internet_gateway.gw]
}

resource "aws_route_table_association" "subnet_one_public" {
  subnet_id      = aws_subnet.sinet_subnet_one.id
  route_table_id = aws_route_table.sinet_table.id
}

resource "aws_subnet" "sinet_subnet_two" {
  vpc_id                  = aws_vpc.sinet_vpc.id
  cidr_block              = "192.168.20.0/24"
  availability_zone       = "us-east-1b"
  map_public_ip_on_launch = true
  depends_on = [aws_internet_gateway.gw]
}

resource "aws_route_table_association" "subnet_two_public" {
  subnet_id      = aws_subnet.sinet_subnet_two.id
  route_table_id = aws_route_table.sinet_table.id
}
```

Fuente (Propia)

- Security.tf: Finalmente y no menos importante encontramos el archivo security.tf, en él se define las reglas de seguridad para todos los servicios relacionados en la infraestructura creada, tanto para ingreso como para salida, en el podemos encontrar reglas para permitir únicamente conexiones externas al clúster de Kubernetes, y así mismo también permitir conexiones internas entre los servicios desplegados en esta infraestructura, por ultimo podemos observar que se permite la conexión hacia afuera de cualquier servicio existente de la infraestructura

desplegada. A continuación, podemos observar la configuración anteriormente mencionada.

Figura 30 *Archivo security.tf AWS*



Fuente (Propia)

Aparte de los archivos de configuración anteriormente explicados encontramos dos más que corresponden al manejo de variables, el primero es variables.tf donde se definen las variables con una breve descripción de su contenido y un segundo llamado terrafom.tfvars que contiene la asignación de estas en un formato llave valor, como particularidad en esta configuración encontramos que a pesar de estar definida la variable db_password en el archivo de variables, este no se encuentra en terraform.tfvars, al configurar esta variable de esta forma, esta es solicitada en el momento de crear la infraestructura y se evita tener datos sensibles en los archivos de configuración de la infraestructura que se encuentra disponible en el repositorio de git.

Figura 31 *archivo variables.tf AWS*

variables.tf

```
variable "region" {
    description = "Region donde se establecera la conexion"
}


variable "access_key" {
    description = "Clave de acceso AWS"
}

variable "secret_key" {
    description = "Llave secreta AWS"
}

variable "role" {
    description = "Role EKS"
}

variable "cluster_name" {
    description = "nombre del cluster EKS"
}

variable "db_password" {
  description = "Contraseña para la base de datos"
  type        = string
  sensitive   = true
}

```

Fuente (Propia)

Figura 32 *Archivo terraform.tfvars AWS*

terraform.tfvars

```
region          = "us-east-1"
access_key      = "xxxxxxxxxxxxxxxxxxxx"
secret_key      = "xxxxxxxxxxxxxxxxxxxx"
role            = "arn:aws:iam::722266111517:role/LabRole"
cluster_name    = "sinet-cluster"
```

Fuente (Propia)

4.4.2.2. Azure

La creación de la infraestructura en Azure es muy similar a AWS y con costos similares, con la diferencia que en Azure no existen instancias similares a las SPOT de AWS con descuentos considerables, lo cual la convierte a esta en una de las opciones de mayor costo.

En cuanto a las diferencias en su configuración encontramos las siguientes:

- A diferencia de AWS y DigitalOcean donde las credenciales se sestean directamente en los archivos de parametrización de Terraform, en Azure, se debe autenticar el cliente de nube antes de iniciar la construcción de la infraestructura. Esto se hace con el cliente Azure para consola az. Se digita el comando *"az login"* y este automáticamente redirecciona a un browser donde pide las credenciales de la cuenta de Azure para finalmente desplegar una página donde dice que el cliente se ha logueado satisfactoriamente. Una vez logueado ya se puede desplegar la infraestructura de Terraform.
- La definición de seguridad para la base de datos no se gobierna desde un componente similar a las VPC de AWS, si no esta es especifica del servicio de base de datos y existe un recurso propios para definir los accesos de nombre azurem_postgresql_flexible_server_firewall_rule, y la particularidad de la configuración de este es que se da la dirección 0.0.0.0 tanto en la bandera start_ip_address como en la bandera end_ip_address, y esto habilita el acceso local desde cualquier recurso de Azure a la base de datos, pero limita el acceso externo al mismo servicio. En la imagen siguiente se puede observar la configuración del servicio de base de datos en Azure.

Figura 33 *archivo azurem_db.tf Azure*

azurem_db.tf

```
resource "azurerm_postgresql_flexible_server" "sinet_server_db" {
  name                = "sinet-server-db"
  location            = azurerm_resource_group.sinet_resource_group.location
  resource_group_name = azurerm_resource_group.sinet_resource_group.name
  sku_name            = "B_Standard_B2ms"
  storage_mb          = 32768
  version             = "14"
  zone                = "1"
  administrator_login    = var.user_login
  administrator_password = var.db_password
}

resource "azurerm_postgresql_flexible_server_database" "sinet_db" {
  name                = "sinet2022"
  server_id           = azurerm_postgresql_flexible_server.sinet_server_db.id
  charset             = "UTF8"
}

resource "azurerm_postgresql_flexible_server_firewall_rule" "postgresql-firewall" {
  name             = "postgresql-fw"
  server_id        = azurerm_postgresql_flexible_server.sinet_server_db.id
  start_ip_address = "0.0.0.0"
  end_ip_address   = "0.0.0.0"
}
```

Fuente (Propia)

Los archivos de configuración completos para la creación de la infraestructura en Azure los podemos encontrar en los anexos.

4.4.2.3. DigitalOcean

DigitalOcean es un servicio en la nube que ha experimentado un crecimiento significativo en popularidad durante los últimos años. Fundada en 2011, esta plataforma ha ganado reconocimiento por su enfoque en la simplicidad y facilidad de uso, lo que la convierte en una opción atractiva para desarrolladores y empresas de todos los tamaños.

Una de las principales razones de su creciente adopción es su enfoque en brindar una experiencia sencilla y orientada al desarrollador. DigitalOcean ofrece una interfaz de usuario intuitiva y una documentación clara que permite a los usuarios implementar y gestionar rápidamente sus recursos en la nube. Además, proporciona una amplia gama de opciones preconfiguradas, conocidas como "droplets", que son máquinas virtuales listas para usar con diferentes tamaños y configuraciones para adaptarse a diversas necesidades.

Otro factor importante a tener en cuenta de DigitalOcean son sus bajos precios en comparación con las principales nubes del mercado, un gran punto a su favor cuando se trata de desplegar infraestructura pequeña y de limitados presupuestos, pero con las garantías de una sólida administración de recursos por parte de esta.

Al igual que AWS y Azure, DigitalOcean también cuenta con modulo para despliegues en Terraform y su parametrización es muy similar a las anteriores.

De resaltar que en cuanto a configuración de credenciales estas al igual que AWS se configuran en los archivos de Terraform con autenticación basada en token, el cual se genera desde el portal de DigitalOcean. También cuenta con un cliente de consola.

Al igual que Azure, la seguridad del servicio de base de datos se administra directamente desde este, mediante un recurso llamado digitalocean_database_firewall y de resaltar es que la creación de los usuarios de la base de datos no es posible hacerlos desde Terraform, si no que se hacen directamente desde el cliente de consola doctl. A continuación, podemos observar cómo se realiza la creación de un servicio de base de datos con lo anteriormente mencionado.

Figura 34 archivo database_cluster.tf DigitalOcean

database_cluster.tf

```
resource "digitalocean_database_cluster" "sinet_db_cluster" {
  name       = "sinet-db-cluster"
  region     = var.region
  engine     = "pg"
  size               = "db-s-1vcpu-1gb"
  node_count         = 1
  version     = 14
  private_network_uuid  = digitalocean_vpc.sinet_vpc.id
}

resource "digitalocean_database_db" "sinet-db" {
  cluster_id = digitalocean_database_cluster.sinet_db_cluster.id
  name       = "sinet2022"
}

resource "digitalocean_database_firewall" "sinet-db-fw" {
  cluster_id = digitalocean_database_cluster.sinet_db_cluster.id

  rule {
    type  = "k8s"
    value = digitalocean_kubernetes_cluster.sinet_cluster.id
  }

}
```

Fuente (Propia)

Los archivos de configuración completos para la creación de la infraestructura en DigitalOcean los podemos encontrar en los anexos.

4.4.3. Kubernetes

Una vez desplegado la infraestructura en cualquiera de las anteriores nubes, se tiene listo un clúster en Kubernetes, el cual de cierta forma es un entorno de virtualización que también requiere se desplieguen ahora sus correspondientes pods, services, secretes, namespaces entre otros, los cuales son necesarios para poner en funcionamiento los aplicativos que se quieren orquestar desde el clúster, para poder realizar lo anterior es necesario actualizar las credenciales de conexión, para posteriormente poder acceder al clúster por el cliente de consola *"kubectl"*.

Dependiendo de la nube con la que se esté trabajando, eso se realiza directamente con el cliente de consola de cada nube, esto se sustentó a detalle en la fase de despliegue en la explicación del pipeline de GitHub Actions.

El despliegue en Kubernetes se configuro para este proyecto en dos fases que son:

4.4.3.1. Despliegues Iniciales

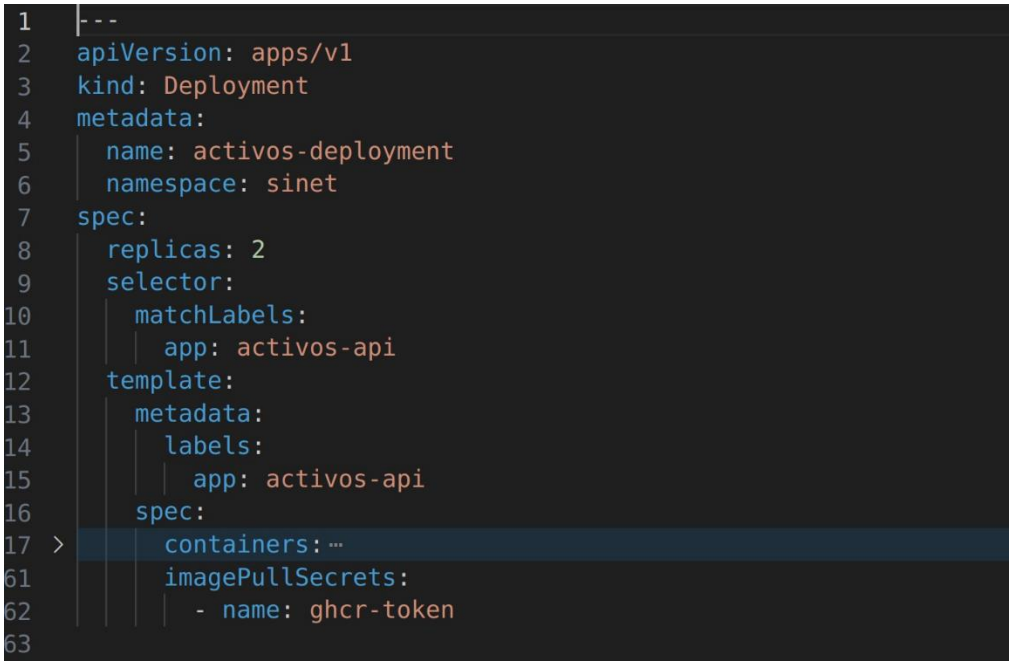
En los despliegues iniciales se crearon los objetos yaml y workflow necesarios para preparar el entorno de Kubernetes con objetos del tipo secrets donde se almacenan credenciales de acceso para la base de datos, y namespace para la separación lógica de los deployments que se crearan por medido de GitHub Actions en procesos de CI/CD, estos archivos de configuración se encuentran en el proyecto sinet_cloud en la carpeta init_k8s. Estos archivos de configuración no serán explicados ya que no tienen configuraciones particulares, sin embargo, los podemos encontrar en los anexos para su estudio.

4.4.3.2. Deployments de microservicios

Todos los microservicios para poder ser desplegados en Kubernetes, necesitan de un archivo deployment, donde se configuran las instrucciones necesarias para su despliegue, en este caso en el mismo archivo se configuro tanto el deployment como el services.

A continuación, se ira explicando la configuración del archivo del servicio sinet_admin_activos.

Figura 35 *archivo deployment Kubernetes*



```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: activos-deployment
  namespace: sinet
spec:
  replicas: 2
  selector:
    matchLabels:
      app: activos-api
  template:
    metadata:
      labels:
        app: activos-api
    spec:
      containers:…
      imagePullSecrets:
        - name: ghcr-token
```

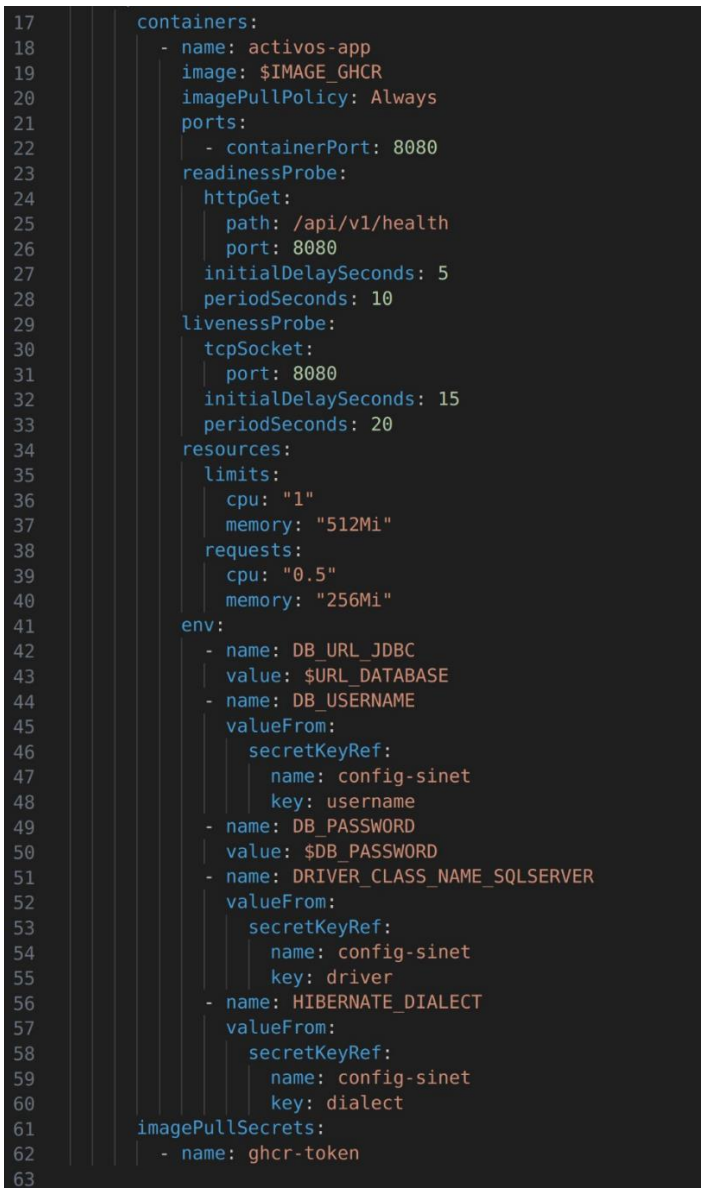
Fuente (Propia)

Este archivo es un manifiesto de Kubernetes que crea un recurso de tipo "Deployment" para desplegar una aplicación llamada "activos-api" en un namespace llamado "sinet". El archivo contiene las siguientes secciones a resaltar:

- apiVersion: apps/v1: Indica la versión de la API de Kubernetes que se utilizará, en este caso, la versión 1 del grupo de recursos "apps".
- kind: Deployment: Define el tipo de recurso que se va a crear, en este caso, un "Deployment". El Deployment es un controlador que administra la creación y actualización de un conjunto de réplicas de pods.
- metadata: Contiene metadatos para el recurso. Aquí se especifica el nombre del Deployment como "activos-deployment" y se indica que pertenece al namespace "sinet".
- spec: Define las especificaciones del Deployment.
- replicas: 2: Indica que se deben crear 2 réplicas del pod que contiene la aplicación.
- selector: Define el selector de etiquetas que se utilizará para identificar los pods administrados por este Deployment. En este caso, los pods tendrán la etiqueta "app: activos-api".
- template: Define la plantilla para crear los pods.

- metadata: Contiene metadatos para los pods. Aquí se asigna la etiqueta "app: activos-api" a los pods.
- spec: Especifica las especificaciones del pod.
- imagePullSecrets: Es una lista de secretos que se utilizan para autenticarse en el registro de contenedores. En este caso, se utiliza un secreto llamado "ghcr-token", que se utiliza para acceder a las imágenes del registro GitHub Container Registry (ghcr.io).
- Posteriormente encontramos las especificaciones del pod como se puede observar en la siguiente imagen:

Figura 36 *especificación del pod a desplegar en Kubernetes*



```
17      containers:
18        - name: activos-app
19          image: $IMAGE_GHCR
20          imagePullPolicy: Always
21          ports:
22            - containerPort: 8080
23          readinessProbe:
24            httpGet:
25              path: /api/v1/health
26              port: 8080
27            initialDelaySeconds: 5
28            periodSeconds: 10
29          livenessProbe:
30            tcpSocket:
31              port: 8080
32            initialDelaySeconds: 15
33            periodSeconds: 20
34          resources:
35            limits:
36              cpu: "1"
37              memory: "512Mi"
38            requests:
39              cpu: "0.5"
40              memory: "256Mi"
41          env:
42            - name: DB_URL_JDBC
43              value: $URL_DATABASE
44            - name: DB_USERNAME
45              valueFrom:
46                secretKeyRef:
47                  name: config-sinet
48                  key: username
49            - name: DB_PASSWORD
50              value: $DB_PASSWORD
51            - name: DRIVER_CLASS_NAME_SQLSERVER
52              valueFrom:
53                secretKeyRef:
54                  name: config-sinet
55                  key: driver
56            - name: HIBERNATE_DIALECT
57              valueFrom:
58                secretKeyRef:
59                  name: config-sinet
60                  key: dialect
61      imagePullSecrets:
62        - name: ghcr-token
63
```

Fuente (Propia)

- containers: Es una lista de contenedores que se ejecutarán en el pod.
- name: activos-app: Es el nombre del contenedor que se creará en el pod.
- image: $IMAGE_GHCR: Es la imagen del contenedor y su valor es reemplazado en el momento que se genera el workflow de GitHub Actions.
- imagePullPolicy: Always: Indica que el pod debe descargar siempre la imagen más reciente.
- ports: Es una lista de puertos que el contenedor expone. En este caso, el contenedor expone el puerto 8080.
- readinessProbe: Es la sonda de preparación que se utiliza para determinar si el contenedor está listo para recibir tráfico. En este caso, se realiza una solicitud HTTP a la ruta /api/v1/health en el puerto 8080 cada 10 segundos después de una espera inicial de 5 segundos.
- livenessProbe: Es la sonda de estado de vida que se utiliza para determinar si el contenedor está en un estado saludable y en funcionamiento. En este caso, se intenta realizar una conexión de socket TCP al puerto 8080 cada 20 segundos después de una espera inicial de 15 segundos.
- resources: Permite establecer límites de recursos (CPU y memoria) para el contenedor. En este caso, el contenedor tiene un límite máximo de 1 CPU y 512 MB de memoria, y una solicitud mínima de 0.5 CPU y 256 MB de memoria.
- env: Es una lista de variables de entorno que requiere el contenedor para poder ejecutarse.

Por último, encontramos la definición del service como se observa en la siguiente imagen, y explicaremos a continuación:

Figura 37 *Definición del despliegue del service Kubernetes*

```
64  ---
65  apiVersion: v1
66  kind: Service
67  metadata:
68    name: activos-service
69  spec:
70    selector:
71      app: activos-api
72    ports:
73      - name: http
74        protocol: TCP
75        port: 8030
76        targetPort: 8080
77
```

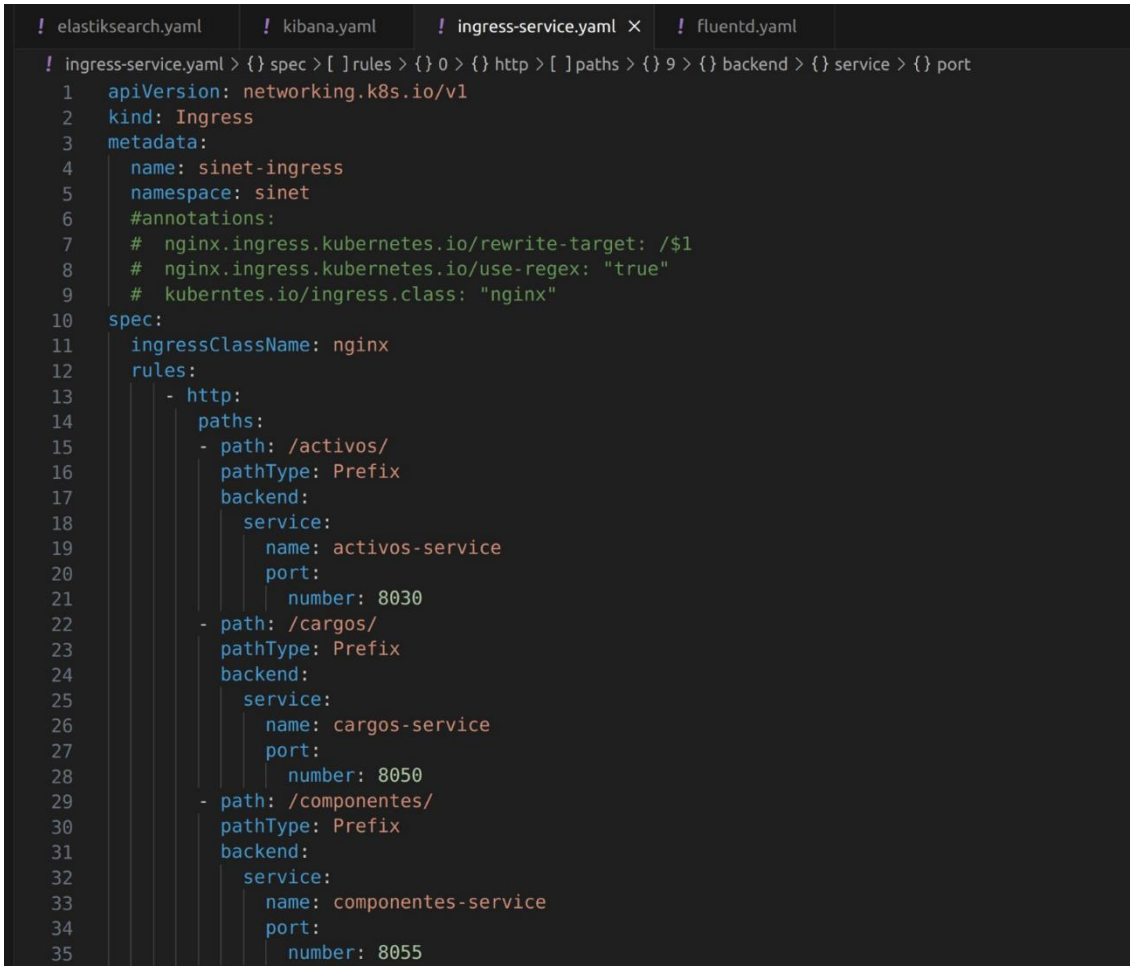
Fuente (Propia)

- apiVersion: v1: Indica que estamos usando la API de Kubernetes versión 1.
- kind: Service: Define que estamos creando un recurso de tipo Service.
- metadata: Contiene los metadatos del Service, como el nombre.
- name: activos-service: Es el nombre del Service que estamos creando.
- spec: Especifica las características del Service.
- selector: Es un conjunto de etiquetas que se utilizan para seleccionar los pods a los que el Service debe dirigir el tráfico. En este caso, se seleccionarán los pods con la etiqueta app: activos-api.
- ports: Es una lista de puertos que se exponen en el Service.
- name: http: Es el nombre asignado al puerto. En este caso, se llama "http".
- protocol: TCP: Es el protocolo que se utiliza para el puerto. En este caso, es TCP.
- port: 8030: Es el puerto que se expone externamente en el Service. Cuando los clientes se conecten a este puerto en el Service, el tráfico se dirigirá al puerto targetPort de los pods seleccionados.
- targetPort: 8080: Es el puerto de los pods al que se dirigirá el tráfico que llega al puerto port del Service. En este caso, el tráfico será dirigido al puerto 8080 de los pods que tengan la etiqueta app: activos-api.

Así concluimos la explicación del archivo deployment y de la misma forma la fase de Operación del ciclo DevOps. La totalidad de los archivos de despliegue se encuentran pueden encontrar anexos.

Una vez desplegada la totalidad de la infraestructura, se verifica que los microservicios están correctamente operativos y funcionando, para ello todos fueron enrutados por medio de un recurso ingress propio de nginx configurado como based path routing, esto quiere decir que, dependiendo de la primera palabra de la ruta, redirecciona al microservicio correspondiente. A continuación, podemos observar la configuración del recurso ingress.

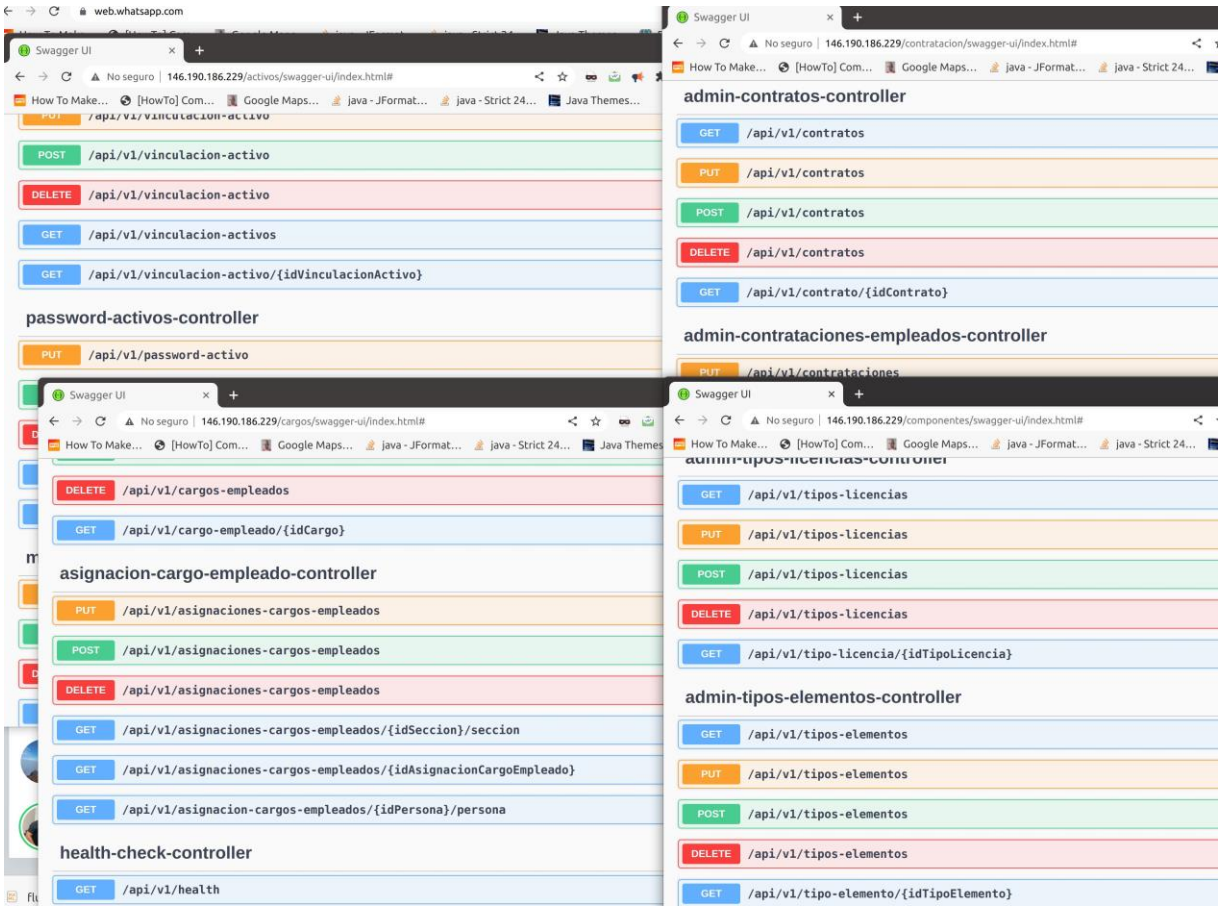
Figura 38 *Configuración recurso ingress con enrutamiento hacia los service de los pods de Sinet*



Fuente (Propia)

Finalmente se realiza la verificación de que se tenga acceso a la página de swagger de algunos de los microservicios para comprobar que el recurso ingress este funcionado correctamente, como se observa en la siguiente imagen.

Figura 39 *Evidencia del despliegue de los microservicios enrutados por el recurso ingress nginx*



Fuente (Propia)

4.5. Monitorización

En la fase de monitorización, se trabajó con la pila EKF (ElastikSearch Kibana y Fluentd) la cual esta explicada a detalle en el apartado 2.1.3.7.

Toda la configuración de su despliegue se encuentra en el proyecto sinet_cloud y se realizara junto con los parámetros de inicialización del clúster.

En el se encuentra un archivo llamado ekf_namespace.yaml que como su nombre lo dice creara un namespace en el clúster donde se desplegaran todos los pods necesarios para poner en funcionamiento la pila de monitorización.

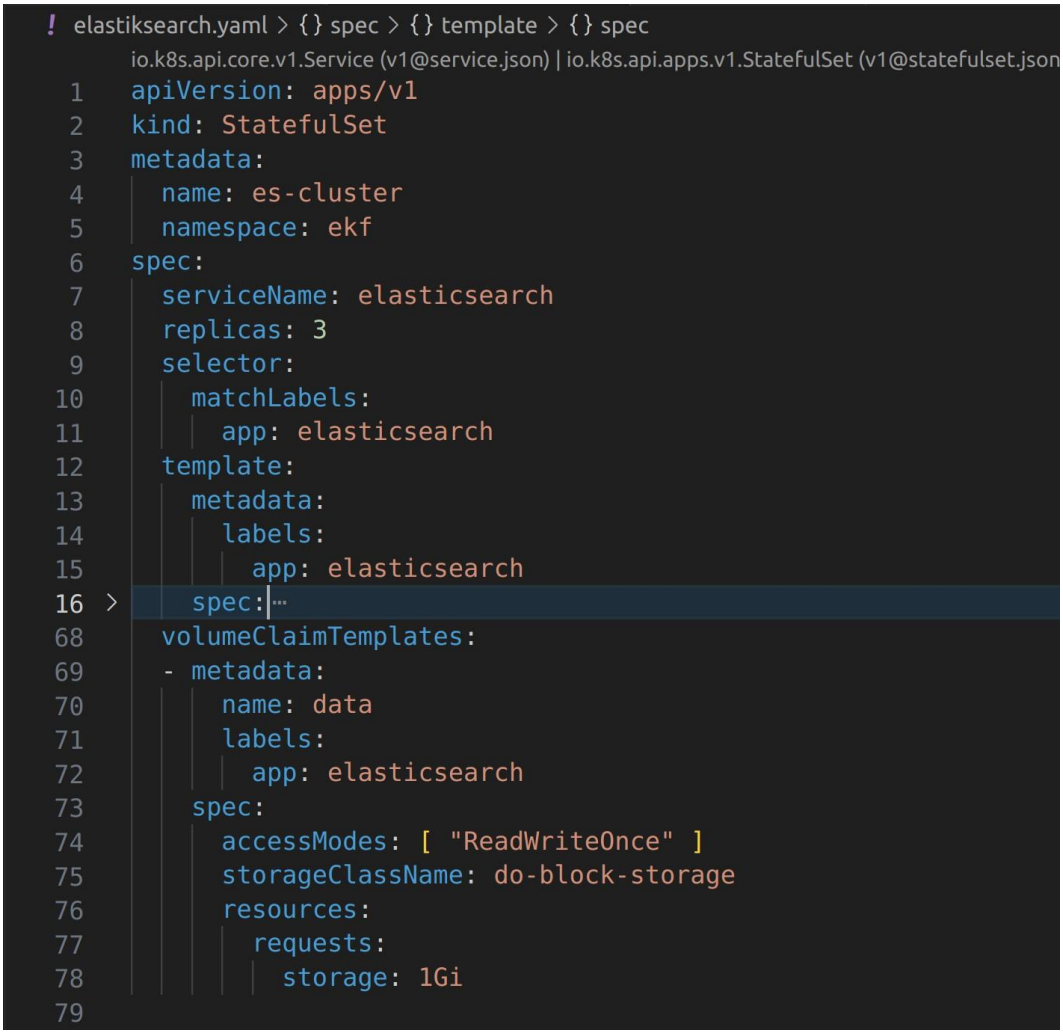
Una vez creado el namespace se procedió a la configuración de la pila con cada una de sus herramientas y servicios como se explica a continuación.

4.5.1. ElasticSearch

La implementación de Elasticsearch como pod de Kubernetes se realiza mediante un controlador de tipo StateFulSet, es de este tipo ya que esta aplicación requiere ser desplegada con una única identidad y almacenamiento persistente, ya que como se sabe, Elasticsearch requiere almacenamiento de la información suministrada por Fluentd.

A continuación, se explica el archivo de despliegue el cual se presenta en la siguiente imagen:

Figura 40 *Archivo elasticsearch.yaml Kubernetes*



Fuente (Propia)

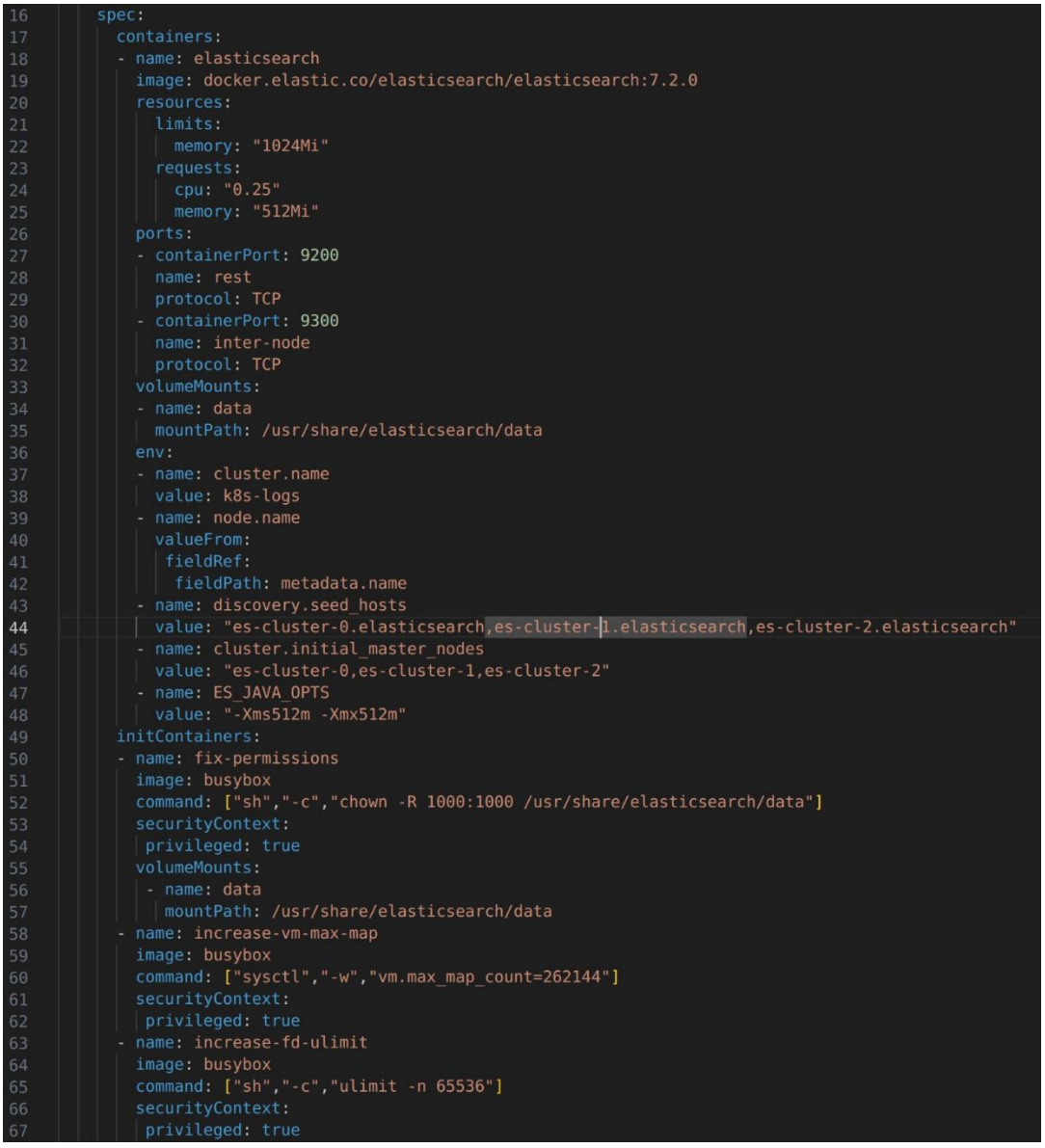
En la configuración general del controlador podemos observar que se define un nombre para el mismo con el nombre elasticsearch el cual se creara dentro del namespace “ekf“ y se define que se debe desplegar en 3 réplicas, también podemos observar que se define un volumeClaimTemplate para el almacenamiento de la data con acceso de lectura y escritura

con un tamaño máximo de 1gb de almacenamiento. De resaltar se puede observar la línea storageClassName: $STORAGE_CLASSNAME. Esta línea se define de esta manera ya que dependiendo de la nube en la que se despliegue su valor asignado cambia de la siguiente manera:

- AWS el valor de la variable es gp2
- Azure el valor es managed-premium
- DigitalOcean el valor es do-block-storage

Continuando con el archivo de configuración, en la siguiente imagen se puede observar la configuración correspondiente al contenedor:

Figura 41 *Elasticsearch.yaml - definición del contenedor*



```
16    spec:
17      containers:
18      - name: elasticsearch
19        image: docker.elastic.co/elasticsearch/elasticsearch:7.2.0
20        resources:
21          limits:
22            memory: "1024Mi"
23          requests:
24            cpu: "0.25"
25            memory: "512Mi"
26        ports:
27        - containerPort: 9200
28          name: rest
29          protocol: TCP
30        - containerPort: 9300
31          name: inter-node
32          protocol: TCP
33        volumeMounts:
34        - name: data
35          mountPath: /usr/share/elasticsearch/data
36        env:
37        - name: cluster.name
38          value: k8s-logs
39        - name: node.name
40          valueFrom:
41           fieldRef:
42            fieldPath: metadata.name
43        - name: discovery.seed_hosts
44          value: "es-cluster-0.elasticsearch,es-cluster-1.elasticsearch,es-cluster-2.elasticsearch"
45        - name: cluster.initial_master_nodes
46          value: "es-cluster-0,es-cluster-1,es-cluster-2"
47        - name: ES_JAVA_OPTS
48          value: "-Xms512m -Xmx512m"
49      initContainers:
50      - name: fix-permissions
51        image: busybox
52        command: ["sh","-c","chown -R 1000:1000 /usr/share/elasticsearch/data"]
53        securityContext:
54          privileged: true
55        volumeMounts:
56          - name: data
57            mountPath: /usr/share/elasticsearch/data
58      - name: increase-vm-max-map
59        image: busybox
60        command: ["sysctl","-w","vm.max_map_count=262144"]
61        securityContext:
62          privileged: true
63      - name: increase-fd-ulimit
64        image: busybox
65        command: ["sh","-c","ulimit -n 65536"]
66        securityContext:
67          privileged: true
```

Fuente (Propia)

Se puede observar en la anterior imagen, como se parametriza el contenedor según las especificaciones requeridas por una imagen descargada desde el sitio de docker.elastic.co para el docker elasticsearch de la versión 7.2.0.

De resaltar encontramos que se definen dos puertos que son el 9200, el cual es el puerto que utiliza el api rest de Elasticsearch y pueda ser accedida a partir de solicitudes HTTP, y por otro lado encontramos el puerto 9300, este es el puerto de comunicación entre los nodos dentro del clúster de Elasticsearch, este segundo le permite al api comunicarse entre los clústeres desplegados para compartir datos y estados entre sí en un clúster distribuido. También se encuentra la sección initContainers, en ella se define un contenedor de inicialización, esta característica es propia de Kubernetes y permite ejecutar uno o más contenedores antes de que los contenedores principales de un pod se inicialicen. En este caso específico, está cambiando los permisos de todos los archivos y directorios el volumen de data el cual se montará posteriormente también ajusta parámetros del kernel necesarios para que el pod funcione correctamente.

Finalmente encontramos la definición del servicio la cual se puede observar en la siguiente imagen:

Figura 42 *definición del service de Elasticsearch*

```
81  apiVersion: v1
82  kind: Service
83  metadata:
84    name: elasticsearch
85    namespace: ekf
86    labels:
87      app: elasticsearch
88  spec:
89    selector:
90      app: elasticsearch
91    clusterIP: None
92    ports:
93      - port: 9200
94        name: rest
95      - port: 9300
96        name: inter-node
97
```

Fuente (Propia)

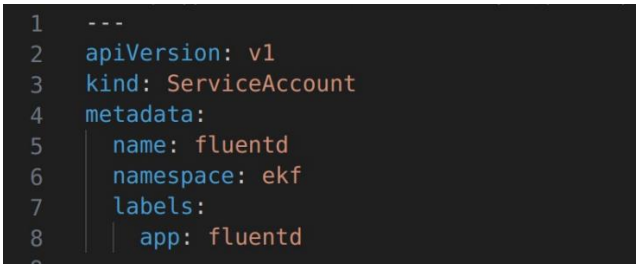
De resaltar en la definición del servicio que es de tipo clusterIP: None, lo que quiere decir que es de tipo “Service headless”, lo que significa que no tiene una IP de clúster asignada. Este solo se utiliza para exponer los puertos del pod seleccionado sin intermediación de un balanceador de carga y únicamente puede ser accedido solo desde los nodos de Kubernetes.

4.5.2. Fluentd

Para la instalación de fluend es necesario definir varias configuraciones necesarias para poder desplegar el DaemonSet que recolectara los registros de cada uno de los pods de los nodos desplegados, entre las secciones de configuración encontramos las siguientes:

- ServiceAccount: Aquí se define una cuenta de servicio llamanda “fluentd” dentro del namespace ekf. Esta cuenta es utilizada por los pods para autenticarse y acceder a los recursos de kubernetes. En este caso el pod de Fluentd utilizara esta cuenta de servicio para ello. En la imagen podemos observar la configuración anteriormente expuesta.

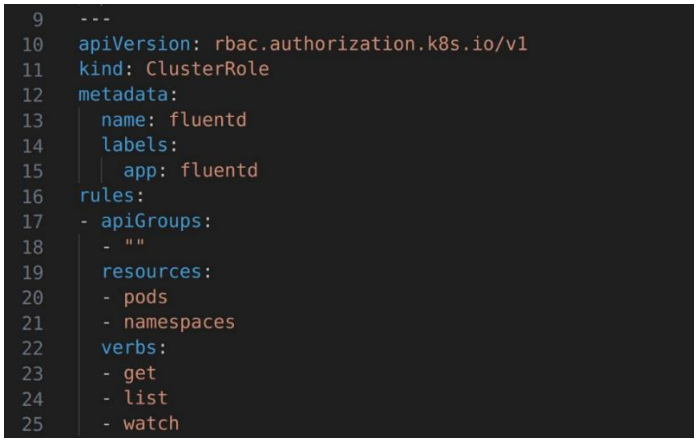
Figura 43 *configuración ServiceAccount Fluentd*



Fuente (Propia)

- ClusterRole: Aquí se define un rol de clúster llamado “fluentd” que contiene reglas de autorización para acceder a recursos específicos en el clúster. En este caso, el rol “fluend” tiene permisos de acceso a recursos de tipo “pods” y “namespaces” con los verbos “get”, “list” y “watch”. Estos permisos hacen que fluentd recopile información de los pods. En la siguiente imagen podemos observar la configuración mencionada.

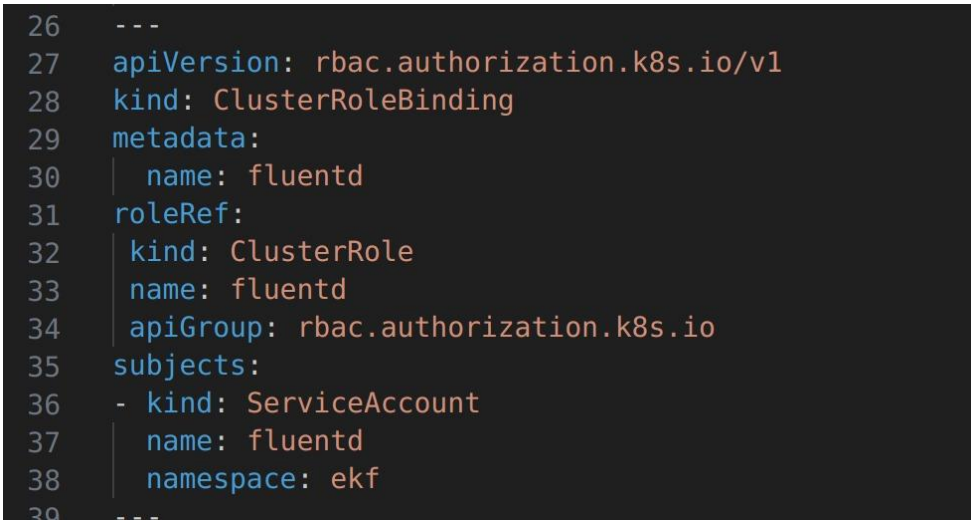
Figura 44 *Configuración ClusterRole Fluentd*



Fuente (Propia)

- ClusterRoleBinding: Esta configuración es propia de fluentd y se utiliza para crear un enlace entre el rol “fluentd” y la cuenta de servicio “fluentd” en el namespace “ekf”, esto quiere decir que la cuenta de servicio “fluentd” tendrá los permisos definidos en el rol “fluend” del clúster. En la siguiente imagen se puede observar la configuración expuesta.

Figura 45 *Configuración ClusterRoleBinding Fluentd*



Fuente (Propia)

- ConfigMap: Dentro este recurso, se definen los archivos de configuración de fluentd, los cuales posteriormente serán montados para que este servicio cargue con la configuración establida, a continuación, se explican brevemente:

  fluent-bit.conf: Este es el archivo de configuración principal de Fluentd. Define configuraciones globales y detalles de entrada, filtrado y salida. Aquí se especifican detalles como la dirección del servidor de salida, la ruta de los archivos de registro, el formato de registro, entre otros.

  input-kubernetes.conf: Este archivo de configuración define la entrada de registros provenientes de clústeres de Kubernetes. Configura Fluentd para que obtenga registros de los contenedores y pods en un clúster Kubernetes y los envíe al siguiente paso de procesamiento.

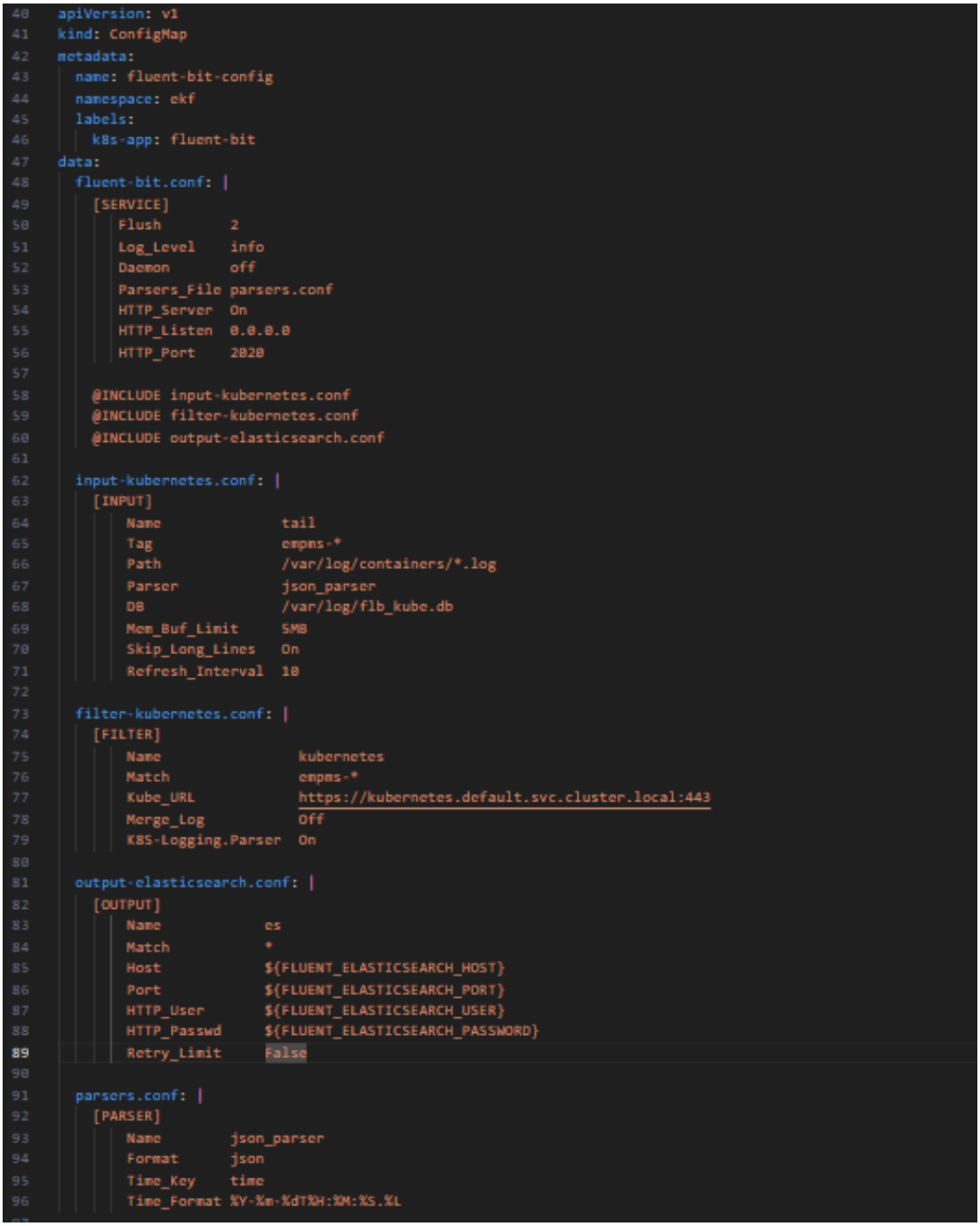
  filter-kubernetes.conf: En este archivo, se establecen las reglas de filtrado para los registros de Kubernetes. Puedes configurar qué registros se aceptan o rechazan

según ciertos criterios, como nombres de pods o etiquetas específicas. También puedes enriquecer los registros con información adicional.

output-elasticsearch.conf: Aquí se configura la salida de los registros hacia una instancia de Elasticsearch. Puedes especificar la dirección del servidor de Elasticsearch, el índice en el que se almacenarán los registros y otras opciones de configuración relacionadas con la conexión y el formato de los datos enviados.

parser.conf**:** En este archivo, puedes definir reglas para analizar y estructurar registros en diferentes formatos. Si los registros están en formatos personalizados o no estructurados, un parser puede ayudar a Fluentd a interpretarlos adecuadamente para su posterior procesamiento y envío.

Figura 46 *Configmap con archivos de configuración fluentd*



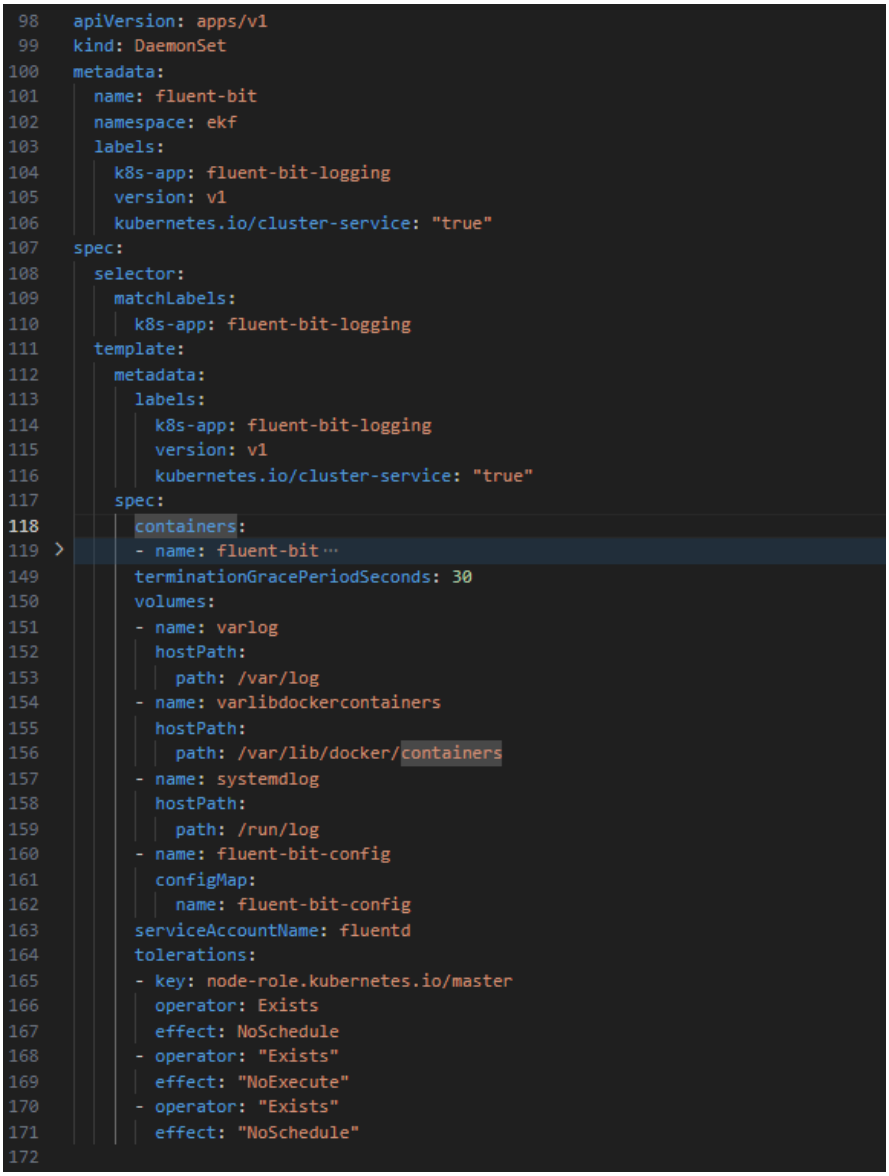
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
  namespace: ekf
  labels:
    k8s-app: fluent-bit
data:
  fluent-bit.conf: |
    [SERVICE]
        Flush        2
        Log_Level    info
        Daemon       off
        Parsers_File parsers.conf
        HTTP_Server  On
        HTTP_Listen  0.0.0.0
        HTTP_Port    2020

    @INCLUDE input-kubernetes.conf
    @INCLUDE filter-kubernetes.conf
    @INCLUDE output-elasticsearch.conf

  input-kubernetes.conf: |
    [INPUT]
        Name              tail
        Tag               empms-*
        Path              /var/log/containers/*.log
        Parser            json_parser
        DB                /var/log/flb_kube.db
        Mem_Buf_Limit     5MB
        Skip_Long_Lines   On
        Refresh_Interval  10

  filter-kubernetes.conf: |
    [FILTER]
        Name                kubernetes
        Match               empms-*
        Kube_URL            https://kubernetes.default.svc.cluster.local:443
        Merge_Log           Off
        K8S-Logging.Parser  On

  output-elasticsearch.conf: |
    [OUTPUT]
        Name            es
        Match           *
        Host            ${FLUENT_ELASTICSEARCH_HOST}
        Port            ${FLUENT_ELASTICSEARCH_PORT}
        HTTP_User       ${FLUENT_ELASTICSEARCH_USER}
        HTTP_Passwd     ${FLUENT_ELASTICSEARCH_PASSWORD}
        Retry_Limit     False

  parsers.conf: |
    [PARSER]
        Name        json_parser
        Format      json
        Time_Key    time
        Time_Format %Y-%m-%dT%H:%M:%S.%L
```

Fuente (Propia)

- DaemonSet: Finalmente encontramos la definición del recurso DaemonSet que es la encargada de desplegar un pod por cada nodo existente en el clúster. Se destaca en esta configuración la definición de dos puntos de montaje de volúmenes volátiles, necesarios para almacenar temporalmente la información recolectada antes de que sea transmitida a Elasticsearch. Estos volúmenes proporcionan un lugar en el pod donde se pueden guardar los datos temporalmente antes de procesarlos y enviarlos

al destino final. En la siguiente imagen podemos observar la configuración completa del recurso DaemonSet.

Figura 47 *Configuración DaemonSet Fluentd*



```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluent-bit
  namespace: ekf
  labels:
    k8s-app: fluent-bit-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: fluent-bit-logging
  template:
    metadata:
      labels:
        k8s-app: fluent-bit-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: fluent-bit ···
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
      - name: systemdlog
        hostPath:
          path: /run/log
      - name: fluent-bit-config
        configMap:
          name: fluent-bit-config
      serviceAccountName: fluentd
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      - operator: "Exists"
        effect: "NoExecute"
      - operator: "Exists"
        effect: "NoSchedule"
```

Fuente (Propia)

- Container: Dentro de la configuración del DaemonSet, encontramos la sección contianers, donde se especifica una imagen del recolector de información fluend, en esta es importante la configuración de las variables de entorno donde se especifica tanto los pods de Elasticsearch como el puerto por el que está escuchando y los puntos de montaje de los volúmenes de información temporal, también se puede encontrar la configuración de la limitación de recursos asignados al pod para su ejecución. A continuación, podemos observar esta configuración.

Figura 48 *configuración container - DaemonSet Fluentd*

```
118       containers:
119       - name: fluent-bit
120         image: fluent/fluent-bit:1.5
121         imagePullPolicy: Always
122         ports:
123           - containerPort: 2020
124         env:
125         - name: FLUENT_ELASTICSEARCH_HOST
126           value: "elasticsearch.ekf.svc.cluster.local"
127         - name: FLUENT_ELASTICSEARCH_PORT
128           value: "9200"
129         - name: FLUENT_ELASTICSEARCH_USER
130           value: "elastic"
131         - name: FLUENT_ELASTICSEARCH_PASSWORD
132           value: "elastic"
133         resources:
134           limits:
135             memory: "512Mi"
136           requests:
137             cpu: "0.25"
138             memory: "512Mi"
139         volumeMounts:
140         - name: varlog
141           mountPath: /var/log
142         - name: varlibdockercontainers
143           mountPath: /var/lib/docker/containers
144           readOnly: true
145         - name: systemdlog
146           mountPath: /run/log
147         - name: fluent-bit-config
148           mountPath: /fluent-bit/etc/
```

Fuente (Propia)

4.5.3. Kibana

Una vez realizados y desplegados los pods tanto para ElasticSearch como para Fluend, se puede proceder con la configuración de Kibana, para así poder visualizar por medio de su aplicación web toda la información recolectada y almacenada por los anteriores servicios entre muchas de las funcionalidades que nos brinda Kibana.

Para ello se configura un recurso de tipo Deployment, que para este trabajo se le define una única replica tomando la imagen correspondiente a Kibana directamente desde el sitio de elastic.co, y especificando en sus variables de entorno, como la URL del service de ElasticSearch y el puerto correspondiente para que pueda acceder a la información recolectada. A continuación, podemos observar esta configuración.

Figura 49 *Configuración Deployment Kibana*

init_k8s > ! kibana.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  namespace: ekf
  labels:
    app: kibana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
      - name: kibana
        image: docker.elastic.co/kibana/kibana:7.2.0
        resources:
          limits:
            memory: "512Mi"
          requests:
            cpu: "0.25"
            memory: "512Mi"
        env:
          - name: ELASTICSEARCH_URL
            value: http://elasticsearch:9200
        ports:
        - containerPort: 5601

```

Fuente (Propia)

Para finalizar, es necesario definir un servicio para poder acceder al pod de Kibana, para ello se realiza la configuración correspondiente del servicio de tipo LoadBalancer para que este pueda ser accedido desde fuera del clúster, el cual se puede acceder desde el puerto TCP 80. A continuación se puede observar la configuración del Service.

Figura 50 *Configuración del Service Kibana*

```
---
apiVersion: apps/v1
kind: Service
metadata:
  name: kibana
  namespace: EKF
  labels:
    app: kibana
spec:
  selector:
    app: kibana
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 5601
    protocol: TCP
```

Fuente (Propia)

El despliegue de la pila EKF se realiza posteriormente a la creación de la infraestructura y esta enlazado al proyecto sinet_cloud, es decir posteriormente a crear las configuraciones iniciales de kubernetes se realiza el despliegue de los tres servicios como se muestra en la siguiente imagen.

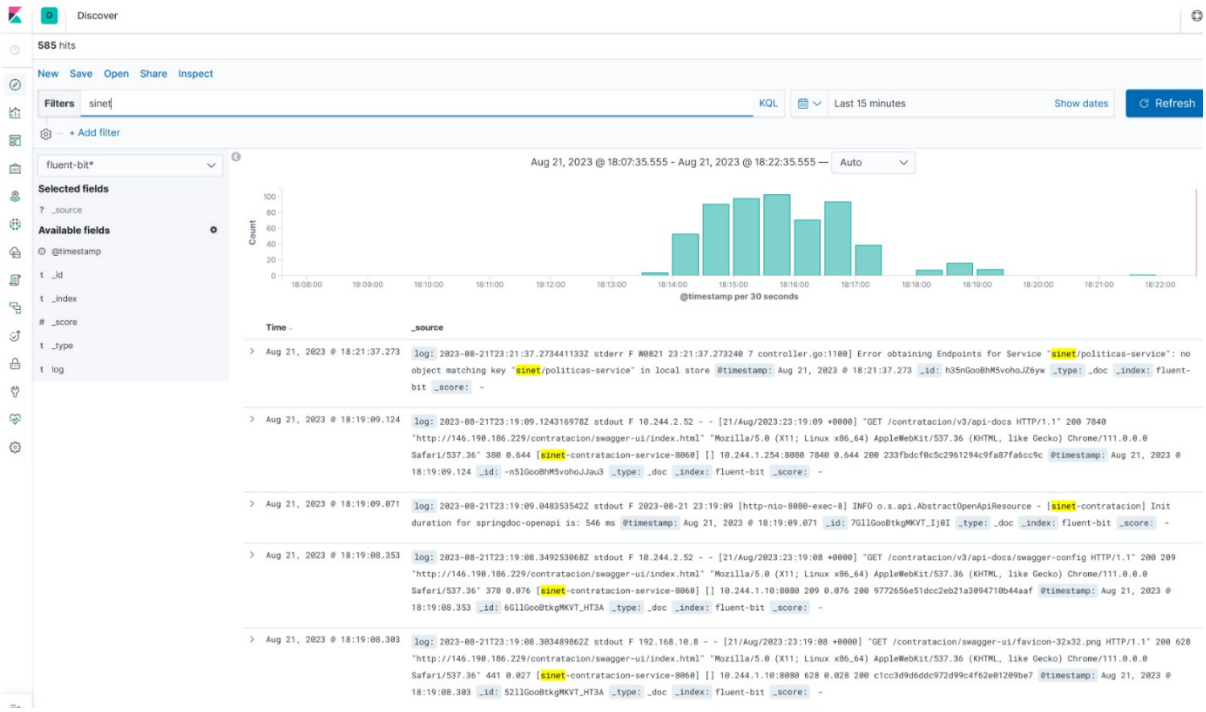
Figura 51 *Pipeline de despliegue de la pila EKF*

```
    - name: Creando namespace para pods de monitorizacion
      run: |
        if kubectl get namespace ekf -o name >/dev/null 2>&1; then
            echo "El namespace ya existe, no se aplicará el archivo YAML."
        else
            kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ekf_namespace.yaml
        fi
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/elastiksearch.yaml
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/fluentd.yaml
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/kibana.yaml
```

Fuente (Propia)

Una vez desplegada la pila EKF, se procede a realizar el despliegue de los microservicios y automáticamente los agentes de fluentd empiezan a recolectar información la cual se registra en Elasticsearch y así mismo esta puede visualizarse en Kibana como se puede observar en la siguiente imagen.

Figura 52 *Kibana desplegando logs capturados por fluentd de los logs de los microservicios desplegados*



Fuente (Propia)

4.5.4. Kube Ops View

También se miró la necesidad de conocer información gráfica sobre el clúster de kubernetes, para ello se instaló esta herramienta visual muy interesante que en tiempo real está indicando el estado de los nodos y los pods, para ello se realizó la descarga de la misma desde su sitio de GitHub y se procedió de desplegar los recursos descargados igualmente en el pipeline de inicialización de kubernetes del proyecto sinet_cloud, como se observa en la imagen a continuación.

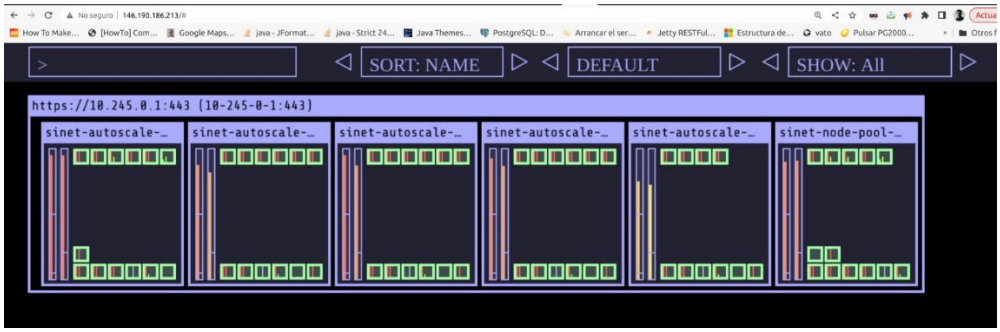
Figura 53 *Pipeline de despliegue de kube-ops-view*

```
73
74      - name: Desplegando monitor de kubernetes kube-ops-view
75        run: |
76          kubectl apply -k ~/work/sinet_cloud/sinet_cloud/init_k8s/deploy
77
```

Fuente (Propia)

De todos los objetos desplegados existe un service el cual se modificó su tipo a loadBalancer para poder ser accedido desde fuera del clúster, a continuación, podemos observar una imagen del monitor de kube-ops-view en funcionamiento con la totalidad de servicios desplegados.

Figura 54 *Kube-ops-view monitoreando el clúster de kubernetes*



Fuente (Propia)

4.6. Retroalimentación

Si bien esta no es una fase operativa que requiera la configuración de herramientas para optimizar su proceso, es de gran importancia mencionarla en este trabajo ya que, finalizados las fases anteriores, quedan muchos aprendizajes de cómo es posible hacer las cosas de mejor manera. A continuación, mencionamos varios puntos que se deberían evaluar para mejorar la eficiencia de los procesos trabajados.

- No necesariamente el ahorrar costos es una buena elección, y esto se pudo evidenciar en GitHub al trabajar una organización gratuita, ya que tenía la limitante de no poder utilizar secretos y variables globales por organización, esto implicaba el tener que replicar las mismas variables por cada uno de los proyectos, en este caso solo eran once, y si bien era un trabajo engorroso de hacer era manejable, pero no es recomendable manejarlo así con una cantidad grande de proyectos, ya que por ejemplo las credenciales cloud tienen vencimiento y esta tarea debería replicarse cada vez que estas caduquen para cada uno de los proyectos de la organización, lo cual representa tiempo en tareas repetitivas.
- Dependiendo del tamaño del proyecto y a quien va dirigido también es fundamental la elección del proveedor de cloud. Ahora existen alternativas más cómodas en costo y pueden brindar los recursos que se necesiten para poder tener un proyecto en nube a costos mucho más accesible.
- Dependiendo de las necesitades y políticas de pasos a producción puede ser no tan necesario que estas sean obligatoriamente condicionadas a un pull-request a la rama correspondiente, sino más bien pasos programados en ciertos días de la semana,

siendo así, estos pasos se pueden desligar la acción especifica, y cuando son microservicios tan similares como en este caso, se podría manejar un mismo pipeline y un mismo archivo deployment, donde por variables de proyecto se podría especificar la particularidad de los mismos, como por ejemplo el nombre del microservicio a desplegar, con esto se conseguiría centralizar la configuración de despliegues solamente en dos archivos, minimizando el riesgo de error y el mantenimiento de los mismos, ya que todo seria centralizado en estos únicos archivos de despliegue.

4.7. Comparativa sobre el despliegue multi-nube

4.7.1. Análisis Detallado de las Plataformas Cloud Abordadas

Finalizadas las fases propuestas en este TFM para cada una de las nubes (AWS, Azure y DigitalOcean) se presenta a continuación los pro y contras encontrados para cada una de ellas los cuales se presentan a continuación.

4.7.1.1. Azure:

Si quien la va a trabajar está familiarizado con el ecosistema de Microsoft, azure desde su Interfaz de usuario le va a parecer muy amigable y intuitiva, todo lo contrario, para los que no, sobre todo para el manejo de credenciales ya que estas van directamente relacionados con conocimiento relacionado a Active Directory, algo que en lo personal fue particularmente difícil de comprender e implementar para lograr autenticar Terraform con la nube de azure.

En cuanto a los costos son muy similares a los de AWS, con la diferencia que no se encontraron descuentas del tipo instancias SPOT como si los hay en AWS, lo que la convierte en la nube más costosa entre las tres utilizadas.

De destacar en Azure podría decir que tiene un soporte excelente, no se puede decir nada de las otras dos nubes ya que no fue requerido, pero en Azure si, y fue para la creación de la cuenta, en la cual cuando se solicitó, el personal me guio con un soporte por chat, hasta conseguir la activación de la cuenta.

4.7.1.2. AWS:

Para nadie es un secreto que AWS es el servicio Cloud por excelencia, para este trabajo fue supremamente sencillo implementar los servicios requeridos, ya que está muy bien diseñada, de resaltar el manejo de las VPC y los Security Groups, y se resaltan ya que da gran facilidad que estos gobiernen la totalidad de los servicios a diferencia de Azure y DigitalOcean donde no funcionan de esta manera y por poner un ejemplo, la administración se seguridad del servicio de base de datos se configura en esta, mientras en AWS no, y esto hace parte de las VPC y sus respectivos Security Groups.

Por otra parte, la creación de credenciales y los permisos otorgados, también al igual que las VPC, todo es administrado por el servicio IAM lo que la vuelve intuitiva y sin conocer demasiado de esta se puede asumir sin temor a equivocarse que ese tipo de necesidades se van a encontrar en el servicio de IAM.

En cuanto a los costos, si bien es cierto que están a la par con los de Azure, esta tiene una diferencia grande que puede reducir sustancialmente el costo de la infraestructura desplegada en Clúster de Kubernetes, y son el tipo de instancias elegidas para el despliegue de los pods.

En AWS se puede elegir que los nodos sean de tipo SPOT, es decir son instancias que se crean sobre infraestructura no utilizada por AWS y otorgan un descuento del 60% sobre el valor normal de una instancia, con la salvedad de que, si dicha infraestructura es requerida, solo se tendrá 2 minutos para migrar el nodo a otra instancia del mismo tipo. El paso del nodo a otro sitio es automático y administrado por el clúster y el tiempo otorgado es suficiente para poder crear un nuevo nodo sin problemas. Este tipo de instancias fueron creadas por AWS para tipos de servicios específicos entre ellos EKS.

Lo anterior hace de AWS muy atractiva en relación costo beneficio y si los servicios a desplegar son para proyectos a gran escala seria la opción ganadora sobre las otras dos.

4.7.1.3. DigitalOcean:

Esta nube tiene un enfoque simplificado y fácil de usar, ideal para quienes desean una experiencia más liviana y directa; es ideal para la pequeña o mediana empresa que quiera incursionar de servidores On Premise a servicios Cloud.

Si bien no es un servicio cloud muy conocido, este viene operando desde el año 2011 y ya tiene presencia global.

Una opción interesante que tiene DigitalOcean es la posibilidad de crear nodos por demanda, lo cual genera un gran ahorro ya que estos se crearan y destruyen según el consumo requerido por sus pods.

En cuanto a su interfaz web, es simple y clara, ofrece máquinas virtuales preconfiguradas (Droplets) que agilizan el proceso de implementación y tiene soporte para despliegues con herramientas de terceros como es el caso de Terraform lo que permitió utilizarla para el desarrollo en este TFM.

En cuanto a precios, es la más competitiva de las 3 con precios muy por debajo de los que maneja Azure y DigitalOcean; puede carecer de la misma gama de servicios avanzados que ofrecen Azure y AWS, Menos opciones de personalización y configuración en comparación con las otras plataformas sin embargo es ideal para el despliegue de infraestructuras pequeñas que no requieran gran complejidad en su infraestructura.

4.7.2. Comparativa de costos

Para realizar una correcta estimación de los costos de infraestructura, primero se debe realizar el cálculo de los recursos que requiere cada uno de los componentes que se van a desplegar. En la tabla que se muestra a continuación se puede ver como se realizó la estimación de recursos.

Tabla 1 *Recursos necesarios para el despliegue en kubernetes*

Recursos Necesarios para el despliegue en Kubernetes					
TYPE	**PODS**	**MEM (Mb)**	**CPU**	**TOTAL MEM**	**TOTAL CPU**
Sinet Spring Boot Deployment	20	320	0,25	6400	5
Fluentd Daemon Set	4	512	0,25	2048	1
ElasticSearch StateFulSet	3	1024	0,25	3072	0,75
Kibana Deployment	2	512	0,25	1024	0,5
TOTAL RECURSOS	**29**	**2368**	**1**	**12544**	**7,25**

Fuente (Propia)

Según la estimación anterior, es necesario contar con 11GB de memoria y 7 cpus, para ello, se cotizará una configuración de 1 nodos de 2cpu y 4gb de memoria cada uno más un pull de 5 nodos adicionales con despliegue de 1 y auto escalamiento máximo a 5, para un total de

24GB de memoria y 12 cpus, valor por encima de los requeridos según la estimación realizada.

Para los pods de Sinet se estiman 20 ya que son 10 microservicios desplegados con dos replicas para un total de 20 pods, para el despliegue de Fluentd al ser de tipo DaemonSet, esto significa que siempre se desplegara 1 por cada nodo para un total de 4 pods. ElasticSearch es de tipo StateFulSet y se despliega con 3 réplicas y Kibana on una. Para un total de 28 pods desplegados.

Del mismo modo también es necesario estimar la base de datos, ya que dependiendo de las conexiones requeridas se define la capacidad de esta.

Teniendo en cuenta que cada pod de Sinet requiere 5 conexiones configuradas en el pool de conexiones a la base de datos y son un total de 20 pods contando sus réplicas es necesario la adquisición de un servicio de base de datos que tenga la capacidad de otorgar un mínimo de 100 conexiones.

Con la información anterior, se realiza una cotización de los servicios requeridos en las tres nubes con el supuesto de que se realice un consumo máximo, sin embargo, estos precios pueden variar dependiendo del consumo de los pods. La siguiente comparación se realiza con costos del año 2023 en dólares americanos y se realiza la respectiva comparación. Los cuales se presentan en la siguiente tabla

Tabla 2 *Comparativa de Costos de despliegue*

Servicio	AWS	Azure	DigitalOcean
1 Clúster Kubernetes	$ 73,00	$ 73,00	$ 12,00
6 Nodos 2 cpu 4gb Ram	$ 114,00	$ 396,00	$ 144,00
2 Balanceadores de Carga	$ 32,86	$ 36,00	$ 24,00
1 Base de datos 8gb 2 core	$ 288,68	$ 590,00	$ 120,00
Totales	**$ 508,54**	**$ 1.095,00**	**$ 300,00**

Fuente (Propia)

Los precios de la tabla anterior se encuentran en dólares americanos y todas sus configuraciones son similares entre todas las nubes, a excepción de la base de datos que en Azure solo se contaba con una configuración de 9gb de memoria y 2 cores en CPU.

Como nota aclaratoria, el servicio de base de datos en Azure es supremamente elevado, sin embargo, también da la posibilidad de utilizar este servicio bajo el modelo de ServerLess y cobro por GB transmitido.

Igualmente podemos observar que los precios de DigitalOcean son los más competitivos en relación a las otras opciones y sería el más adecuado para trabajar este tipo de proyecto, ya que es un proyecto pequeño y es posible manejarlo en esta nube, la cual presta todos los servicios necesarios para funcionar, además todo el respaldo y soporte que brinda este proveedor, sin embargo, en caso de requerir una gran cantidad de nodos a desplegar, la opción más atractiva sería la de AWS, ya que esta por el modelo de nodos con instancias SPOT, sería mucho más rentable que DigitalOcean.

5. Conclusiones y trabajo futuro

5.1. Conclusiones

El implementar las herramientas necesarias para la automatización de las fases de desarrollo, integración, despliegue y monitorización del ciclo DevOps es posible y en este trabajo se describió el paso a paso para poder alcanzar dicho objetivo, sin embargo, es fundamental tener el conocimiento y experiencia necesaria para hacerlo y así lograr una implementación exitosa de los objetivos propuestos en este TFM.

En la fase de desarrollo, logramos adaptar nuestros microservicios de manera eficaz para que fueran compatibles con un orquestador como Kubernetes. Esta adaptación, que incluyó ajustes esenciales para garantizar la idoneidad de los microservicios, ha establecido una base sólida para futuras escalabilidades y despliegues continuos.

En la fase de integración, la configuración de GitHub para la integración continua de nuestros microservicios ha impulsado un enfoque de desarrollo colaborativo y una mayor calidad en la gestión de nuestro código fuente. Los flujos de trabajo definidos para el manejo de repositorios han fomentado la coherencia y eficiencia en nuestro proceso de integración, asegurando que las últimas versiones de los microservicios sean evaluadas y verificadas constantemente.

En la fase de despliegue, la implementación de herramientas clave como GitHub Actions y la configuración de flujos de trabajo para diversas nubes han facilitado despliegues automatizados y confiables. El hecho de que hayamos logrado desplegar microservicios en múltiples entornos, demostrando su funcionalidad, valida el éxito de nuestras iniciativas de despliegue continuo.

La fase de operación ha sido esencial para optimizar la infraestructura subyacente y los procesos de despliegue. La configuración y optimización del Dockerfile han mejorado los tiempos de creación y despliegue de imágenes, permitiendo una mayor agilidad en la entrega de nuevas versiones. La implementación de Terraform y la adaptación a las particularidades de las nubes de AWS, Azure y DigitalOcean han allanado el camino para despliegues consistentes y reproducibles.

Una conclusión importante a la que se llegó en esta fase es que la elección del lenguaje de programación y frameworks involucrados en el desarrollo de los microservicios es de vital importancia, ya que estos en si definen la carga tanto en CPU como en la cantidad de memoria que se requiere para su funcionamiento, en el caso particular Java + Spring Boot no es una combinación recomendable debido a la considerable cantidad de recursos requerida, lo cual se convierte en costos elevados en el despliegue de la infraestructura, ya que por la cantidad de recursos que consumen son menos la cantidad de pods que se pueden desplegar por nodo requiriendo más nodos para la totalidad de recursos que se despliegan.

Finalmente, la fase de monitorización ha resultado en una mayor visibilidad y control sobre el rendimiento de nuestros microservicios. La implementación exitosa de la pila EKF ha proporcionado una solución robusta para la gestión de registros y el análisis de datos en tiempo real. Esta suite ha mejorado significativamente la capacidad de detección temprana de problemas y la toma de decisiones informadas, contribuyendo en última instancia a la estabilidad y confiabilidad de nuestro entorno de desarrollo y producción.

En conjunto, la implementación exitosa de estas fases propuestas en este TFM, han aumentado la eficiencia, la calidad y la confiabilidad en todos los aspectos del proceso en el ciclo de vida DevOps para el aplicativo SINET. Estos logros son testimonio de nuestro compromiso con la mejora continua y la adopción de prácticas modernas de desarrollo, lo que nos posiciona de manera sólida para enfrentar los desafíos y oportunidades en constante evolución en el mundo del desarrollo de software.

5.2. Líneas de trabajo futuro

Terminado el alcance del proyecto se encontraron grandes vacíos que deberían tratarse en trabajos futuros los cuales se detallan a continuación:

- La implementación de secretos en Kubernetes mediante codificación en Base64, aunque comúnmente utilizada, no es considerada una práctica segura para proteger información sensible. La codificación en Base64 no es una forma de cifrado ni de encriptación, es simplemente una forma de representar datos binarios en formato de caracteres ASCII, lo que significa que cualquier persona con acceso al secreto en Kubernetes podría fácilmente decodificarlo y acceder a la información en texto claro.

Esto es especialmente preocupante si los archivos de creación de Kubernetes que contienen secretos, están almacenados en un repositorio público o en un entorno compartido. Por tanto, sería interesante desarrollar un trabajo que trate el tema de cómo es la implementación de seguridad en los secretos de secretos de Kubernetes.

- En el aprendizaje de kubernetes, encontramos grandes virtudes que este ofrece como la escalabilidad horizontal dependiendo de políticas definidas que se encargan de verificar la carga de los pods para decidir la creación automática de nuevos pods y suplir la demanda de un servicio específico, sin embargo, todo esto es interno y depende de una configuración constante de los nodos del clúster, más el clúster no tienen la capacidad de crear nodos por demanda, por tanto para suplir este inconveniente generalmente se definen nodos de reserva los cuales pueden estar inoperantes o la capacidad del clúster estar operando muy por debajo de su máximo, lo cual representa una elevación en los costos de la infraestructura que a la larga genera pérdidas por reservas inutilizadas, o también se puede dar el caso contrario, que la demanda suba esporádicamente y se llegue al límite de la capacidad de los nodos y se empiece a bajar el rendimiento o caída de los servicios al superar la capacidad máxima de los mismos. Para suplir este inconveniente existe herramientas externas al clúster que se encarga de verificar la carga, pero ahora de los nodos y tienen la capacidad de crear nuevos nodos para luego enlazarlos al clúster, por mencionar algunas herramientas de este tipo tenemos Kube Scale y Clúster Autoescale. Esta sería una línea de trabajo interesante a tratar ya que aparte de la funcionalidad que brindan estas herramientas, también sería muy interesante saber el ahorro en costos que una empresa puede tener al implementarlas ya que con estas herramientas ya no existiera la necesidad de crear nodos de reserva y se puede tener operando los nodos a una capacidad superior al 80% con la garantía de un escalado a nivel de nodos en caso de necesitarlo.
- Los costos de operación en una infraestructura cloud son de vital importancia y juega un papel fundamental los recursos que se van a desplegar en esta para así lograr minimizarlos al máximo sin afectar el rendimiento de los sistemas desplegados, para ello los lenguajes de programación en los que se desarrollan en este caso los microservicios son cruciales y pueden mostrar diferencias muy significativas.

Una investigación muy interesante sería el realizar una comparativa de consumo de recursos de un mismo microservicio desarrollado en diferentes lenguajes de programación, teniendo en cuenta variables adicionales como el tiempo de desarrollo, y el mantenimiento de los mismos, para así tener bases sólidas y realizar una correcta elección de los lenguajes de programación para este tipo de desarrollos.

- Por otra parte, y a medida que se iba desarrollando el proyecto, empezó a surgir la siguiente inquietud. Se tiene una infraestructura desarrollada como código, y se tienen creados los pipelines y los archivos deployment de kubernetes para el despliegue automático de los pods a medida que se realice pull request a cada una de las ramas productivas involucradas, pero ¿qué pasa si se llegara a perder el clúster?, si bien los procesos están automatizados, el despliegue de cada uno de los pods en este momento debería irse realizando de forma manual por cada uno de los proyecto de git, lo cual sería manejable porque solo son 10 proyectos, pero si fuera una cantidad mayor, que mecanismos se deben implementar para poder realizar una recuperación total de la infraestructura? Investigando un poco sobre el tema se encontró que para ello se debe implementar un plan de recuperación de desastres o más conocido como DRP.

  "Disaster Recovery Plan" es un conjunto de procedimientos y estrategias diseñadas para restaurar sistemas y datos críticos en caso de un desastre o interrupción grave en una organización. El objetivo principal es minimizar el tiempo de inactividad y asegurar la continuidad del negocio. El DRP aborda cómo recuperar sistemas, datos y aplicaciones esenciales después de eventos como fallas de hardware, ataques cibernéticos, desastres naturales u otras situaciones que puedan afectar la operación normal de una empresa.

  Me parece que es indispensable la existencia de un DRP de debe ser parte de un entregable cuando se está implementando un ciclo DevOps y al igual que el punto anterior creo que es un tema que debería tratarse a profundidad en un trabajo futuro.

6. BIBLIOGRAFÍA

Alexander, C. M. (20 de 10 de 2021). Obtenido de http://repositorio.espe.edu.ec/bitstream/21000/29374/1/T-ESPE-052322.pdf

Amazon. (2023). *http://amazon.com.* Obtenido de https://docs.aws.amazon.com/es_es/

Andrés Martínez Villegas, P. N. (01 de 07 de 2022). *unisimon.edu.co.* Obtenido de https://revistas.unisimon.edu.co/index.php/innovacioning/article/view/6142

DigitalOcean. (s.f.). *https://digitalocean.com.* Obtenido de https://docs.digitalocean.com/

Durández, E. A. (2020). *https://docs.google.com/viewerng/viewer?url=https://e-archivo.uc3m.es/bitstream/handle/10016/32686/TFG_Esther_Anton_Durandez.pdf.*

ElasticSearch. (2023). *https://www.elastic.co/.* Obtenido de https://www.elastic.co/guide/en/enterprise-search/current/index.html

Fluentbit. (2023). *fluentbit.io.* Obtenido de https://docs.fluentbit.io/manual

GitHub. (2023). *https://github.com.* Obtenido de https://docs.github.com/en

Hernández-Alonso, J. J. (13 de 07 de 2022). Obtenido de https://reunir.unir.net/handle/123456789/13548

Kubernetes. (s.f.). *https://kubernetes.io.* Obtenido de https://kubernetes.io/docs/home/

Microsoft, A. (s.f.). *https://azure.microsoft.com/.* Obtenido de https://learn.microsoft.com/en-us/azure/?product=popular

netapp. (2023). *https://www.netapp.com.* Obtenido de https://www.netapp.com/es/devops-solutions/what-is-devops/

Netfilx. (2023). *https://netflixtechblog.com.* Obtenido de https://netflixtechblog.com/full-cycle-developers-at-netflix-a08c31f83249

Quintero, J. R. (2020). *Pontificia Universidad Catolica de Ecuador.* Obtenido de https://repositorio.pucese.edu.ec/bitstream/123456789/2602/1/Huerlo%20Quintero%20Jurgen%20Ronaldo.pdf

Suntaxi-Cocanguilla, P. S. (22 de 07 de 2020). *https://reunir.unir.ne.* Obtenido de https://reunir.unir.net/handle/123456789/14022

Terraform, H. (2023). *http://terraform.io.* Obtenido de https://developer.hashicorp.com/terraform?product_intent=terraform

Velásquez Santos, A. V. (2022). Obtenido de https://renati.sunedu.gob.pe/handle/sunedu/3372689

Velásquez Santos, A. V. (11 de 2022). *https://repositorio.utp.edu.pe/handle/20.500.12867/6753.*

Yeren, B. F. (19 de 01 de 2020). *https://repositorio.ucv.edu.pe.* Obtenido de https://repositorio.ucv.edu.pe/bitstream/handle/20.500.12692/41593/Casas_YFE.pdf?sequence=1

Anexo A. Ajustes en código fuente para configurar readlinesProbe de Kubernetes

Clase HealthCheckController

```
import co.com.sinet.services.HealthCheckService;


@RestController
@RequestMapping("api/v1")
public class HealthCheckController {
   @Autowired
   private HealthCheckService healthCheckService;
   @GetMapping("/health")
   public ResponseEntity<String> healthCheck() {
         boolean isDatabaseHealthy = healthCheckService.isDatabaseHealthy();
         if (isDatabaseHealthy) {
            return ResponseEntity.ok("Servicio saludable");
         } else {
            return     ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error en la base de datos");
         }
   }
}
```

Clase HealtCheckService

```
@Service
public class HealthCheckService {
   @Autowired
   private EntityManager entityManager;
   public boolean isDatabaseHealthy() {
         try {
            Query query = entityManager.createNativeQuery("SELECT 1");
            Object result = query.getSingleResult();
            // Verifica que se haya obtenido algún resultado
            return result != null;
         } catch (Exception e) {
            // Ocurrió un error al conectar o consultar la base de datos
            e.printStackTrace();
            return false;
         }
   }
}
```

Anexo B. Pipeline de despliegue servicio sinet_admin_activos para DigitalOcean

```
name: Kubernetes Deploy DigitalOcean

on:
  pull_request:
    branches: [ produccion_do ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:

   - name: Cancel Previous Runs
     uses: styfle/cancel-workflow-action@0.8.0
     with:
       access_token: ${{ github.token }}

   - name: Checkout repository
     uses: actions/checkout@v2

   - name: Install Java y Maven
     uses: actions/setup-java@v2
```

```
  with:
    java-version: '11'
    distribution: 'adopt'
    cache: maven

- name: Compile Project
  run: mvn clean compile

- name: Test Proyect
  run: mvn test

- name: Package Project
  run: mvn package

- name: Configure Docker
  uses: docker/setup-buildx-action@v1

- name: Login to GitHub Container Registry
  uses: docker/login-action@v1
  with:
     registry: ghcr.io
     username: ${{ vars.USERNAME }}
     password: ${{ secrets.CR_PAT }}

- name: Create Image Tag
```

```
      run: |
        TIMESTAMP=`date +%Y%m%d%H%M%S`
        SHORT_SHA=$(echo "${GITHUB_SHA}" | cut -c1-6)
        SNAPSHOT_TAG="${TIMESTAMP}${SHORT_SHA}"
        echo "SNAPSHOT_TAG=$SNAPSHOT_TAG" >> $GITHUB_ENV

    - name: Build and push
      run: |
        export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker build . --tag ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker push $IMAGE_GHCR

    - name: Downloading doctl DigitalOcean
      run: |
        wget https://github.com/digitalocean/doctl/releases/download/v1.94.0/doctl-1.94.0-linux-amd64.tar.gz
        tar xvfz doctl-1.94.0-linux-amd64.tar.gz
        sudo mv doctl /usr/local/bin

    - name: Authenticating DigitalOcean
      run: |
        doctl auth init --access-token ${{ secrets.DO_ACCESS_TOKEN }}

    - name: Update kubectl DigitalOcean
```

```
    run: |
      doctl k8s cluster kubeconfig save ${{ vars.CLUSTER_NAME }}

  - name: Kubernetes Configuration Generation
    run: |
      export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
      echo $IMAGE_GHCR
      export ID_DATABASE=`doctl db list --format ID | awk 'NR>1'`
      export URL_CLUSTER=`doctl db connection $ID_DATABASE --format Host | awk 'NR>1'`
      export DB_PORT=`doctl db connection $ID_DATABASE --format Port | awk 'NR>1'`
      export URL_DATABASE=jdbc:postgresql://$URL_CLUSTER:$DB_PORT/sinet2022
      export DB_PASSWORD=`doctl db user get $ID_DATABASE ${{ secrets.DB_USER }} --format Password | awk 'NR>1'`
      envsubst < ~/./work/sinet_admin_activos/sinet_admin_activos/deployments/activos-deployment.yaml > ~/activos-deployment-ok.yaml

  - name: Kubernetes deployment
    run: |
      cat ~/activos-deployment-ok.yaml
      kubectl apply -f ~/activos-deployment-ok.yaml
```

Anexo C. Pipeline de despliegue servicio sinet_admin_activos para Azure

```
name: Kubernetes Deploy Azure

on:
  pull_request:
    branches: [ produccion_az ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:

   - name: Cancel Previous Runs
     uses: styfle/cancel-workflow-action@0.8.0
     with:
       access_token: ${{ github.token }}

   - name: Checkout repository
     uses: actions/checkout@v2

   - name: Install Java y Maven
     uses: actions/setup-java@v2
```

```
  with:
    java-version: '11'
    distribution: 'adopt'
    cache: maven

- name: Compile Project
  run: mvn clean compile

- name: Test Proyect
  run: mvn test

- name: Package Project
  run: mvn package

- name: Configure Docker
  uses: docker/setup-buildx-action@v1

- name: Login to GitHub Container Registry
  uses: docker/login-action@v1
  with:
     registry: ghcr.io
     username: ${{ vars.USERNAME }}
     password: ${{ secrets.CR_PAT }}

- name: Create Image Tag
```

```
      run: |
        TIMESTAMP=`date +%Y%m%d%H%M%S`
        SHORT_SHA=$(echo "${GITHUB_SHA}" | cut -c1-6)
        SNAPSHOT_TAG="${TIMESTAMP}${SHORT_SHA}"
        echo "SNAPSHOT_TAG=$SNAPSHOT_TAG" >> $GITHUB_ENV

    - name: Build and push
      run: |
        export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker build . --tag ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker push $IMAGE_GHCR

    - name: Login a Azure
      uses: azure/login@v1
      with:
        creds: ${{ secrets.AZURE_CREDENTIALS }}

    - name: Updating kubectl Azure
      run: |
        az aks get-credentials --resource-group ${{ secrets.AZ_RESOURCE_GROUP }} --name ${{ vars.CLUSTER_NAME }}

    - name: Kubernetes Configuration Generation
      run: |
```

```
        export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}

        echo $IMAGE_GHCR

        export URL_DB=`az postgres flexible-server show --name sinet-server-db --resource-group ${{ secrets.AZ_RESOURCE_GROUP }} --output tsv --query "fullyQualifiedDomainName"`

        export URL_DATABASE=jdbc:postgresql://$URL_DB/sinet2022

        export DB_PASSWORD=${{ secrets.DB_PASSWORD }}

        envsubst < ~/./work/sinet_admin_activos/sinet_admin_activos/deployments/activos-deployment.yaml > ~/activos-deployment-ok.yaml


    - name: Kubernetes deployment

      run: |

        cat ~/activos-deployment-ok.yaml

        kubectl apply -f ~/activos-deployment-ok.yaml
```

Anexo D. Pipeline de despliegue servicio sinet_admin_activos para AWS

```
name: Kubernetes Deploy AWS

on:
  pull_request:
    branches: [ produccion_aws ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:

   - name: Cancel Previous Runs
     uses: styfle/cancel-workflow-action@0.8.0
     with:
       access_token: ${{ github.token }}

   - name: Checkout repository
     uses: actions/checkout@v2

   - name: Install Java y Maven
     uses: actions/setup-java@v2
```

```
  with:
    java-version: '11'
    distribution: 'adopt'
    cache: maven

- name: Compile Project
  run: mvn clean compile

- name: Test Proyect
  run: mvn test

- name: Package Project
  run: mvn package

- name: Configure Docker
  uses: docker/setup-buildx-action@v1

- name: Login to GitHub Container Registry
  uses: docker/login-action@v1
  with:
     registry: ghcr.io
     username: ${{ vars.USERNAME }}
     password: ${{ secrets.CR_PAT }}

- name: Create Image Tag
```

```
      run: |
        TIMESTAMP=`date +%Y%m%d%H%M%S`
        SHORT_SHA=$(echo "${GITHUB_SHA}" | cut -c1-6)
        SNAPSHOT_TAG="${TIMESTAMP}${SHORT_SHA}"
        echo "SNAPSHOT_TAG=$SNAPSHOT_TAG" >> $GITHUB_ENV

    - name: Build and push
      run: |
        export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker build . --tag ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        docker push $IMAGE_GHCR

    - name: Config credentials AWS
      run: |
          mkdir -p ~/.aws
          touch ~/.aws/credentials
          echo "${{ secrets.AWS_CREDENTIAL }}" > ~/.aws/credentials
          echo "region=${{ vars.AWS_REGION }}" >> ~/.aws/credentials

    - name: Update kubectl AWS
      run: |
          aws eks update-kubeconfig --name ${{ vars.CLUSTER_NAME }} --region ${{ vars.AWS_REGION }}
```

```
    - name: Kubernetes Configuration Generation
      run: |
        export IMAGE_GHCR=ghcr.io/${{ vars.ORGANIZACION }}/activos-api:${{ env.SNAPSHOT_TAG }}
        echo $IMAGE_GHCR
        export URL_RDS=$(aws rds describe-db-instances --db-instance-identifier postgresql-sinet2022 --query 'DBInstances[0].Endpoint.Address' --output text)
        export URL_DATABASE=jdbc:postgresql://$URL_RDS/sinet2022
        export DB_PASSWORD=${{ secrets.DB_PASSWORD }}
        envsubst < ~/./work/sinet_admin_activos/sinet_admin_activos/deployments/activos-deployment.yaml > ~/activos-deployment-ok.yaml

    - name: Kubernetes deployment
      run: |
        cat ~/activos-deployment-ok.yaml
        kubectl apply -f ~/activos-deployment-ok.yaml
```

Anexo E. Archivo Deployment del servicio sinet_admin_activos

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: activos-deployment
  namespace: sinet
spec:
  replicas: 2
  selector:
    matchLabels:
      app: activos-api
  template:
    metadata:
      labels:
        app: activos-api
    spec:
      containers:
        - name: activos-app
          image: $IMAGE_GHCR
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          readinessProbe:
```

```
    httpGet:
      path: /activos/api/v1/health
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 10
  livenessProbe:
    tcpSocket:
      port: 8080
    initialDelaySeconds: 15
    periodSeconds: 20
  resources:
    limits:
      memory: "300Mi"
    requests:
      cpu: "0.2"
      memory: "300Mi"
  env:
    - name: DB_URL_JDBC
      value: $URL_DATABASE
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: config-sinet
          key: username
    - name: DB_PASSWORD
```

```
          value: $DB_PASSWORD
        - name: DRIVER_CLASS_NAME_SQLSERVER
          valueFrom:
            secretKeyRef:
              name: config-sinet
              key: driver
        - name: HIBERNATE_DIALECT
          valueFrom:
            secretKeyRef:
              name: config-sinet
              key: dialect
    imagePullSecrets:
      - name: ghcr-token
---
apiVersion: v1
kind: Service
metadata:
  name: activos-service
  namespace: sinet
spec:
  selector:
    app: activos-api
  ports:
    - name: http
      protocol: TCP
```

port: 8030

targetPort: 8080

Anexo F. Archivos de creación de infraestructura en Terraform para DigitalOcean

main.tf

```
terraform {
  required_providers {
    digitalocean = {
      source = "digitalocean/digitalocean"
      version = "~> 2.0"
    }
  }
}

provider "digitalocean" {
  token = var.token
}

resource "digitalocean_kubernetes_cluster" "sinet_cluster" {
  name  = "sinet-cluster"
  region = var.region
  version = "1.27.4-do.0"
  vpc_uuid   = digitalocean_vpc.sinet_vpc.id
  node_pool {
        name            = "sinet-node-pool"
        node_count      = 1
```

```
    size       = "s-2vcpu-4gb"
  }

}

resource "digitalocean_kubernetes_node_pool" "sinet_autoscale_pool" {
  cluster_id = digitalocean_kubernetes_cluster.sinet_cluster.id
  name       = "sinet-autoscale-pool"
  size       = "s-2vcpu-4gb"
  auto_scale = true
  min_nodes  = 1
  max_nodes  = 5
}
```

network.tf

```
resource "digitalocean_vpc" "sinet_vpc" {
  name = "sinet-vpc"
  region = var.region
  ip_range = "192.168.10.0/24"
}
```

database.tf

```
resource "digitalocean_database_cluster" "sinet_db_cluster" {
  name                 = "sinet-db-cluster"
  region               = var.region
  engine               = "pg"
  size          = "db-s-2vcpu-4gb"
```

```
  node_count           = 1
  version              = 14
  private_network_uuid  = digitalocean_vpc.sinet_vpc.id
}

resource "digitalocean_database_db" "sinet-db" {
  cluster_id = digitalocean_database_cluster.sinet_db_cluster.id
  name       = "sinet2022"
}

resource "digitalocean_database_firewall" "sinet-db-fw" {
  cluster_id = digitalocean_database_cluster.sinet_db_cluster.id

  rule {
    type  = "k8s"
    value = digitalocean_kubernetes_cluster.sinet_cluster.id
  }
  depends_on = [digitalocean_kubernetes_cluster.sinet_cluster]
}
```

Anexo G. Archivos de creación de infraestructura en Terraform para Azure

main.tf

```
provider "azurerm" {
  features {}
}


resource "azurerm_resource_group" "sinet_resource_group" {
  name     = "sinet-resource-group"
  location = var.region
}


resource "azurerm_kubernetes_cluster" "sinet_aks_cluster" {
  name                = "sinet-cluster"
  location            = var.region
  resource_group_name = azurerm_resource_group.sinet_resource_group.name
  dns_prefix          = "sinet-aks-cluster"

  default_node_pool {
    name           = "default"
    node_count     = 2
    vm_size        = "Standard_DS2_v2"
    os_disk_size_gb = 30
  }
```

```
  network_profile {
    network_plugin   = "azure"
    load_balancer_sku = "standard"
  }

  identity {
    type = "SystemAssigned"
  }

  depends_on = [
    azurerm_resource_group.sinet_resource_group
  ]
}
```

azure_db.tf

```
resource "azurerm_postgresql_flexible_server" "sinet_server_db" {
  name                = "sinet-server-db"
  location            = azurerm_resource_group.sinet_resource_group.location
  resource_group_name = azurerm_resource_group.sinet_resource_group.name
  sku_name            = "B_Standard_B2ms"
  storage_mb          = 32768
  version             = "14"
  zone                = "1"
  administrator_login    = var.user_login
  administrator_password = var.db_password
```

```
}


resource "azurerm_postgresql_flexible_server_database" "sinet_db" {
  name             = "sinet2022"
  server_id         = azurerm_postgresql_flexible_server.sinet_server_db.id
  charset          = "UTF8"
}


resource "azurerm_postgresql_flexible_server_firewall_rule" "postgresql-firewall" {
  name           = "postgresql-fw"
  server_id       = azurerm_postgresql_flexible_server.sinet_server_db.id
  start_ip_address = "0.0.0.0"
  end_ip_address   = "0.0.0.0"
}
```

nodes_group.tf

```
resource "azurerm_kubernetes_cluster_node_pool" "sinet_node_group" {
  name                 = "sinetnodes"
  kubernetes_cluster_id   = azurerm_kubernetes_cluster.sinet_aks_cluster.id
  vm_size               = "Standard_D2s_v3"  # Debes seleccionar el tamaño de máquina virtual adecuado
  node_count            = 2  # Debes configurar el número de nodos según tus necesidades
}
```

Anexo H. Archivos de creación de infraestructura en Terraform para AWS

main.tf

```
provider "aws" {
 region     = var.region
 shared_credentials_files = ["$HOME/.aws/credentials"]
}



resource "aws_eks_cluster" "sinet_cluster" {
 name     = var.cluster_name
 role_arn = var.role
 vpc_config {
  subnet_ids = [aws_subnet.sinet_subnet_one.id,
                aws_subnet.sinet_subnet_two.id,
                aws_subnet.sinet_subnet_three.id]
 }

 depends_on = [aws_vpc.sinet_vpc]
}
```

networks.tf

```
resource "aws_vpc" "sinet_vpc" {
 cidr_block          = "192.168.0.0/16"
 enable_dns_hostnames = true
```

```
 tags = {
  Name = "SINET VPC"
 }
}

resource "aws_internet_gateway" "gw" {
 vpc_id          = aws_vpc.sinet_vpc.id
 tags            = {
    Name = "Gateway"
 }
}

resource "aws_route_table" "sinet_table" {
 vpc_id       = "${aws_vpc.sinet_vpc.id}"
 route {
  cidr_block  = "0.0.0.0/0"
  gateway_id  = "${aws_internet_gateway.gw.id}"
 }
}

resource "aws_subnet" "sinet_subnet_one" {
 vpc_id               = aws_vpc.sinet_vpc.id
 cidr_block           = "192.168.10.0/24"
 availability_zone      = "us-east-1a"
 map_public_ip_on_launch = true
```

```
  depends_on = [aws_internet_gateway.gw]
}

resource "aws_route_table_association" "subnet_one_public" {
  subnet_id      = aws_subnet.sinet_subnet_one.id
  route_table_id = aws_route_table.sinet_table.id
}

resource "aws_subnet" "sinet_subnet_two" {
  vpc_id                  = aws_vpc.sinet_vpc.id
  cidr_block              = "192.168.20.0/24"
  availability_zone       = "us-east-1b"
  map_public_ip_on_launch = true
  depends_on = [aws_internet_gateway.gw]
}

resource "aws_route_table_association" "subnet_two_public" {
  subnet_id      = aws_subnet.sinet_subnet_two.id
  route_table_id = aws_route_table.sinet_table.id
}

resource "aws_subnet" "sinet_subnet_three" {
  vpc_id                  = aws_vpc.sinet_vpc.id
  cidr_block              = "192.168.30.0/24"
  availability_zone       = "us-east-1c"
```

```
  map_public_ip_on_launch = true

  depends_on = [aws_internet_gateway.gw]

}


resource "aws_route_table_association" "subnet_three_public" {

  subnet_id      = aws_subnet.sinet_subnet_three.id

  route_table_id = aws_route_table.sinet_table.id

}
```

security.tf

```
resource "aws_security_group" "sinet_security" {

  name        = "sinet_security"

  description = "Aceptar todas conexiones"

  vpc_id      = aws_vpc.sinet_vpc.id


  # Regla para permitir conexiones kubectl desde cualquier dirección IP externa

  ingress {

    from_port   = 6443

    to_port     = 6443

    protocol    = "tcp"

    cidr_blocks = ["0.0.0.0/0"]

  }


  # Regla para permitir que los servicios internos se vean entre sí

  ingress {

    from_port   = 0
```

```
  to_port     = 65535
  protocol    = "tcp"
  security_groups = [aws_security_group.sinet_security.id]
 }

 # Regla para permitir que los servicios internos se vean entre sí
 egress {
  from_port   = 0
  to_port     = 65535
  protocol    = "tcp"
  security_groups = [aws_security_group.sinet_security.id]
 }
}
```

nodes_group.tf

```
resource "aws_eks_node_group" "sinet_node_group" {
 cluster_name    = aws_eks_cluster.sinet_cluster.name
 node_group_name = "sinet-node-group"
 node_role_arn = var.role
 instance_types = ["t3.medium"]
 capacity_type = "SPOT"
 scaling_config {
  desired_size = 2
  min_size     = 2
  max_size     = 3
 }
```

```
  update_config {
        max_unavailable          = 1
  }

  subnet_ids = [aws_subnet.sinet_subnet_one.id,
              aws_subnet.sinet_subnet_two.id,
              aws_subnet.sinet_subnet_three.id]

  depends_on = [aws_eks_cluster.sinet_cluster]

}
```

rds.tf

```
resource "aws_db_subnet_group" "subnet_group_sinet" {
   name       = "grupo de subnets eks rds"
   subnet_ids  = [aws_subnet.sinet_subnet_one.id,
                aws_subnet.sinet_subnet_two.id,
                aws_subnet.sinet_subnet_three.id]
}
resource "aws_db_instance" "sinet2022" {
  identifier          = "postgresql-sinet2022"
  allocated_storage    = 10
  engine              = "postgres"
  engine_version      = "14.7"
  instance_class      = "db.t3.micro"
```

```
  db_name              = "sinet2022"

  username             = "emaku"

  password             = var.db_password

  publicly_accessible = false

  db_subnet_group_name     = aws_db_subnet_group.subnet_group_sinet.id

  vpc_security_group_ids    = [aws_security_group.sinet_security.id]

  skip_final_snapshot      = true

  final_snapshot_identifier = "Ignore"


}
```

Anexo I. Archivos yaml de inicialización del clúster de Kubernetes

ekf_namespaces.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: ekf
```

sinet_namespaces.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: sinet
```

sinet_secrets.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: config-sinet
  namespace: sinet
data:
  url: **********
  username: **********
  password: **********
  driver: **********
  dialect: **********
```

Anexo J. Archivos yaml creación StateFulSet Elasticsearch

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: es-cluster
  namespace: ekf
spec:
  serviceName: elasticsearch
  replicas: 3
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
    spec:
      containers:
      - name: elasticsearch
        image: docker.elastic.co/elasticsearch/elasticsearch:7.2.0
        resources:
          limits:
            memory: "1024Mi"
          requests:
```

```
      cpu: "0.25"
      memory: "512Mi"
    ports:
    - containerPort: 9200
      name: rest
      protocol: TCP
    - containerPort: 9300
      name: inter-node
      protocol: TCP
    volumeMounts:
    - name: data
      mountPath: /usr/share/elasticsearch/data
    env:
    - name: cluster.name
      value: k8s-logs
    - name: node.name
      valueFrom:
       fieldRef:
        fieldPath: metadata.name
    - name: discovery.seed_hosts
      value: "es-cluster-0.elasticsearch,es-cluster-1.elasticsearch,es-cluster-2.elasticsearch"
    - name: cluster.initial_master_nodes
      value: "es-cluster-0,es-cluster-1,es-cluster-2"
    - name: ES_JAVA_OPTS
      value: "-Xms512m -Xmx512m"
```

```
      initContainers:
      - name: fix-permissions
        image: busybox
        command: ["sh","-c","chown -R 1000:1000 /usr/share/elasticsearch/data"]
        securityContext:
          privileged: true
        volumeMounts:
        - name: data
          mountPath: /usr/share/elasticsearch/data
      - name: increase-vm-max-map
        image: busybox
        command: ["sysctl","-w","vm.max_map_count=262144"]
        securityContext:
          privileged: true
      - name: increase-fd-ulimit
        image: busybox
        command: ["sh","-c","ulimit -n 65536"]
        securityContext:
          privileged: true
  volumeClaimTemplates:
  - metadata:
      name: data
      labels:
        app: elasticsearch
    spec:
```

```
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: do-block-storage
    resources:
     requests:
       storage: 1Gi


---
apiVersion: v1
kind: Service
metadata:
 name: elasticsearch
 namespace: ekf
 labels:
  app: elasticsearch
spec:
 selector:
  app: elasticsearch
 clusterIP: None
 ports:
  - port: 9200
    name: rest
  - port: 9300
    name: inter-node
```

Anexo K. Archivos yaml creación DaemonSet Fluentd

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd
  namespace: ekf
  labels:
    app: fluentd
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: fluentd
  labels:
    app: fluentd
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - namespaces
  verbs:
  - get
  - list
```

```
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fluentd
roleRef:
 kind: ClusterRole
 name: fluentd
 apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: fluentd
  namespace: ekf
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
  namespace: ekf
  labels:
    k8s-app: fluent-bit
data:
  fluent-bit.conf: |
    [SERVICE]
```

```
    Flush        2
    Log_Level    info
    Daemon       off
    Parsers_File parsers.conf
    HTTP_Server  On
    HTTP_Listen  0.0.0.0
    HTTP_Port    2020

  @INCLUDE input-kubernetes.conf
  @INCLUDE filter-kubernetes.conf
  @INCLUDE output-elasticsearch.conf

input-kubernetes.conf: |
  [INPUT]
    Name             tail
    Tag              empms-*
    Path             /var/log/containers/*.log
    Parser           json_parser
    DB               /var/log/flb_kube.db
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

filter-kubernetes.conf: |
  [FILTER]
```

```
    Name              kubernetes
    Match             empms-*
    Kube_URL          https://kubernetes.default.svc.cluster.local:443
    Merge_Log         Off
    K8S-Logging.Parser  On


 output-elasticsearch.conf: |
  [OUTPUT]
    Name          es
    Match         *
    Host          ${FLUENT_ELASTICSEARCH_HOST}
    Port          ${FLUENT_ELASTICSEARCH_PORT}
    HTTP_User     ${FLUENT_ELASTICSEARCH_USER}
    HTTP_Passwd   ${FLUENT_ELASTICSEARCH_PASSWORD}
    Retry_Limit   False


 parsers.conf: |
  [PARSER]
    Name       json_parser
    Format     json
    Time_Key   time
    Time_Format %Y-%m-%dT%H:%M:%S.%L
---
apiVersion: apps/v1
kind: DaemonSet
```

```
metadata:
  name: fluent-bit
  namespace: ekf
  labels:
    k8s-app: fluent-bit-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: fluent-bit-logging
  template:
    metadata:
      labels:
        k8s-app: fluent-bit-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: fluent-bit
        image: fluent/fluent-bit:1.5
        imagePullPolicy: Always
        ports:
          - containerPort: 2020
        env:
```

```
  - name: FLUENT_ELASTICSEARCH_HOST
    value: "elasticsearch.ekf.svc.cluster.local"
  - name: FLUENT_ELASTICSEARCH_PORT
    value: "9200"
  - name: FLUENT_ELASTICSEARCH_USER
    value: "elastic"
  - name: FLUENT_ELASTICSEARCH_PASSWORD
    value: "elastic"
  resources:
    limits:
      memory: "512Mi"
    requests:
      cpu: "0.25"
      memory: "512Mi"
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: varlibdockercontainers
    mountPath: /var/lib/docker/containers
    readOnly: true
  - name: systemdlog
    mountPath: /run/log
  - name: fluent-bit-config
    mountPath: /fluent-bit/etc/
terminationGracePeriodSeconds: 30
```

```
volumes:
- name: varlog
  hostPath:
    path: /var/log
- name: varlibdockercontainers
  hostPath:
    path: /var/lib/docker/containers
- name: systemdlog
  hostPath:
    path: /run/log
- name: fluent-bit-config
  configMap:
    name: fluent-bit-config
serviceAccountName: fluentd
tolerations:
- key: node-role.kubernetes.io/master
  operator: Exists
  effect: NoSchedule
- operator: "Exists"
  effect: "NoExecute"
- operator: "Exists"
  effect: "NoSchedule"
```

Anexo L. Archivos yaml creación Deployment Kibana

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  namespace: ekf
  labels:
    app: kibana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
      - name: kibana
        image: docker.elastic.co/kibana/kibana:7.2.0
        resources:
          limits:
            memory: "512Mi"
          requests:
```

```
        cpu: "0.25"
        memory: "512Mi"
      env:
       - name: ELASTICSEARCH_URL
        value: http://elasticsearch:9200
      ports:
      - containerPort: 5601


---
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: ekf
  labels:
    app: kibana
spec:
  selector:
   app: kibana
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 5601
    protocol: TCP
```

Anexo M. Archivos yaml creación recurso ingress nginx

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sinet-ingress
  namespace: sinet
spec:
  ingressClassName: nginx
  rules:
    - http:
      paths:
      - path: /activos/
        pathType: Prefix
        backend:
          service:
            name: activos-service
            port:
              number: 8030
      - path: /cargos/
        pathType: Prefix
        backend:
          service:
            name: cargos-service
            port:
              number: 8050
```

```
  - path: /componentes/
    pathType: Prefix
    backend:
      service:
        name: componentes-service
        port:
          number: 8055
  - path: /contratacion/
    pathType: Prefix
    backend:
      service:
        name: contratacion-service
        port:
          number: 8060
  - path: /empleados/
    pathType: Prefix
    backend:
      service:
        name: empleados-service
        port:
          number: 8070
  - path: /personas/
    pathType: Prefix
    backend:
      service:
```

```
      name: personas-service
      port:
        number: 8065
  - path: /politicas/
    pathType: Prefix
    backend:
      service:
        name: politicas-service
        port:
          number: 8040
  - path: /redes/
    pathType: Prefix
    backend:
      service:
        name: redes-service
        port:
          number: 8035
  - path: /soportes/
    pathType: Prefix
    backend:
      service:
        name: soportes-service
        port:
          number: 8075
  - path: /usuarios/
```

```
    pathType: Prefix
    backend:
      service:
        name: usuarios-service
        port:
          number: 8045
```

Anexo N. Pipeline de GitHub actions para despliegue en DigitalOcean

```
name: Kubernetes initialization DigitalOcean

on:
  pull_request:
    branches: [ produccion_do ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:
   - name: Cancel Previous Runs
     uses: styfle/cancel-workflow-action@0.8.0
     with:
       access_token: ${{ github.token }}

   - name: Checkout of repository
     uses: actions/checkout@v2

   - name: Configure Docker
     uses: docker/setup-buildx-action@v1
```

```
    - name: Downloading doctl DigitalOcean
      run: |
        wget  https://github.com/digitalocean/doctl/releases/download/v1.94.0/doctl-1.94.0-linux-amd64.tar.gz
        tar xvfz doctl-1.94.0-linux-amd64.tar.gz
        sudo mv doctl /usr/local/bin

    - name: Authenticating DigitalOcean
      run: |
        doctl auth init --access-token ${{ secrets.DO_ACCESS_TOKEN }}

    - name: Update kubectl DigitalOcean
      run: |
        doctl k8s cluster kubeconfig save ${{ vars.CLUSTER_NAME }}

    - name: Verification and creation of namespace sinet
      run: |
        if kubectl get namespace sinet -o name >/dev/null 2>&1; then
            echo "El namespace ya existe, no se aplicará el archivo YAML."
        else
            kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_namespace.yaml
        fi

    - name: Verification and creation of secrets
      run: |
```

```
      if kubectl get secret config-sinet -n sinet -o name >/dev/null 2>&1; then
        echo "El secreto config-sinet ya existe, no se aplicará el archivo YAML."
      else
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_secrets.yaml -n sinet
      fi
      if kubectl get secret ghcr-token -n sinet -o name >/dev/null 2>&1; then
       echo "El secreto ghcr-token, por tanto no se correra el comando de creacion"
      else
       kubectl create secret docker-registry ghcr-token \
       --docker-server=${{ vars.GHCR_URL }} \
       --docker-username=${{ vars.GHCR_USER }} \
       --docker-password=${{ secrets.CR_PAT }} \
       --docker-email=${{ vars.GHCR_MAIL }} \
       -n sinet
      fi

  - name: Creando namespace para pods de monitorizacion
    run: |
      if kubectl get namespace ekf -o name >/dev/null 2>&1; then
        echo "El namespace ya existe, no se aplicará el archivo YAML."
      else
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ekf_namespace.yaml
      fi
      kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/elastiksearch.yaml
```

```
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/fluentd.yaml

    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/kibana.yaml


- name: Desplegando monitor de kubernetes kube-ops-view

  run: |

    kubectl apply -k ~/work/sinet_cloud/sinet_cloud/init_k8s/deploy


- name: Desplegando servicio ingress como API Manager

  run: |

    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ingress-service.yaml
```

Anexo O. Pipeline de GitHub actions para despliegue en Azure

```
name: Kubernetes initialization Azures

on:

  pull_request:

    branches: [ produccion_az ]


jobs:

 build:

  runs-on: ubuntu-latest


  steps:

   - name: Cancel Previous Runs

     uses: styfle/cancel-workflow-action@0.8.0

     with:

      access_token: ${{ github.token }}


   - name: Checkout del repositorio

     uses: actions/checkout@v2


   - name: Configure Docker

     uses: docker/setup-buildx-action@v1


   - name: Login a Azure

     uses: azure/login@v1

     with:
```

```
        creds: ${{ secrets.AZURE_CREDENTIALS }}


    - name: Update kubectl Azure
      run: |
        az aks get-credentials --resource-group ${{ secrets.AZ_RESOURCE_GROUP }} --name ${{ vars.CLUSTER_NAME }}


    - name: Verification and creation of namespace sinet
      run: |
        if kubectl get namespace sinet -o name >/dev/null 2>&1; then
          echo "El namespace ya existe, no se aplicará el archivo YAML."
        else
          kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_namespace.yaml
        fi


    - name: Verificación y creación de secretos
      run: |
        if kubectl get secret config-sinet -n sinet -o name >/dev/null 2>&1; then
          echo "El secreto config-sinet ya existe, no se aplicará el archivo YAML."
        else
          kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_secrets.yaml -n sinet
        fi
        if kubectl get secret ghcr-token -n sinet -o name >/dev/null 2>&1; then
         echo "El secreto ghcr-token, por tanto no se correra el comando de creacion"
```

```
    else
      kubectl create secret docker-registry ghcr-token \
      --docker-server=${{ vars.GHCR_URL }} \
      --docker-username=${{ vars.GHCR_USER }} \
      --docker-password=${{ secrets.CR_PAT }} \
      --docker-email=${{ vars.GHCR_MAIL }} \
      -n sinet
    fi

- name: Creando namespace para pods de monitorizacion
  run: |
    if kubectl get namespace ekf -o name >/dev/null 2>&1; then
       echo "El namespace ya existe, no se aplicará el archivo YAML."
    else
       kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ekf_namespace.yaml
    fi
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/elastiksearch.yaml
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/fluentd.yaml
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/kibana.yaml

- name: Desplegando monitor de kubernetes kube-ops-view
  run: |
    kubectl apply -k ~/work/sinet_cloud/sinet_cloud/init_k8s/deploy

- name: Desplegando servicio ingress como API Manager
```

```
run: |

  kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ingress-service.yaml
```

Anexo P. Pipeline de GitHub actions para despliegue en AWS

```
name: Kubernetes initialization AWS

on:
  pull_request:
    branches: [ produccion_aws ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:
   - name: Cancel Previous Runs
     uses: styfle/cancel-workflow-action@0.8.0
     with:
       access_token: ${{ github.token }}

   - name: Checkout of reposity
     uses: actions/checkout@v2

   - name: Configure Docker
     uses: docker/setup-buildx-action@v1

   - name: Config credentials AWS
     run: |
```

```
        mkdir -p ~/.aws
        touch ~/.aws/credentials
        echo "${{ secrets.AWS_CREDENTIAL }}" > ~/.aws/credentials
        echo "region=${{ vars.AWS_REGION }}" >> ~/.aws/credentials

    - name: Updating kubectl
      run: |
        aws eks update-kubeconfig --name ${{ vars.CLUSTER_NAME }} --region ${{ vars.AWS_REGION }}

    - name: Verification and creation of namespace sinet
      run: |
        if kubectl get namespace sinet -o name >/dev/null 2>&1; then
          echo "El namespace ya existe, no se aplicará el archivo YAML."
        else
          kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_namespace.yaml
        fi

    - name: Verification and creation of secrets
      run: |
        if kubectl get secret config-sinet -n sinet -o name >/dev/null 2>&1; then
          echo "El secreto config-sinet ya existe, no se aplicará el archivo YAML."
        else
          kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/sinet_secrets.yaml -n sinet
```

```
    fi
    if kubectl get secret ghcr-token -n sinet -o name >/dev/null 2>&1; then
      echo "El secreto ghcr-token, por tanto no se correra el comando de creacion"
    else
      kubectl create secret docker-registry ghcr-token \
      --docker-server=${{ vars.GHCR_URL }} \
      --docker-username=${{ vars.GHCR_USER }} \
      --docker-password=${{ secrets.CR_PAT }} \
      --docker-email=${{ vars.GHCR_MAIL }} \
      -n sinet
    fi

- name: Creando namespace para pods de monitorizacion
  run: |
    if kubectl get namespace ekf -o name >/dev/null 2>&1; then
        echo "El namespace ya existe, no se aplicará el archivo YAML."
    else
        kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ekf_namespace.yaml
    fi
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/elastiksearch.yaml
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/fluentd.yaml
    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/kibana.yaml

- name: Desplegando monitor de kubernetes kube-ops-view
  run: |
```

```
    kubectl apply -k ~/work/sinet_cloud/sinet_cloud/init_k8s/deploy


- name: Desplegando servicio ingress como API Manager

  run: |

    kubectl apply -f ~/work/sinet_cloud/sinet_cloud/init_k8s/ingress-service.yaml
```

Anexo Q. Screenshot de evidencia del despliegue de la totalidad de los microservicios

Despliegue de servicios

```
felipe@ldap:~/unir-tfm/sinet_cloud/digital_ocean$ kubectl get services -n sinet
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
activos-service             ClusterIP   10.245.9.168     <none>        8030/TCP   58m
cargos-service              ClusterIP   10.245.80.227    <none>        8050/TCP   58m
componentes-service         ClusterIP   10.245.94.58     <none>        8055/TCP   46m
contratacion-service        ClusterIP   10.245.170.43    <none>        8060/TCP   45m
empleados-service           ClusterIP   10.245.126.137   <none>        8070/TCP   44m
personas-service            ClusterIP   10.245.71.38     <none>        8065/TCP   43m
politicas-acceso-service    ClusterIP   10.245.120.25    <none>        8040/TCP   43m
redes-service               ClusterIP   10.245.112.52    <none>        8035/TCP   42m
soportes-service            ClusterIP   10.245.120.45    <none>        8075/TCP   41m
felipe@ldap:~/unir-tfm/sinet_cloud/digital_ocean$
```

Despliegue de deployments

```
felipe@ldap:~/unir-tfm/sinet_cloud/digital_ocean$ kubectl get pods -n sinet
NAME                                        READY   STATUS    RESTARTS   AGE
activos-deployment-7477c64d9-94r5q          1/1     Running   0          25m
activos-deployment-7477c64d9-nk969          1/1     Running   0          25m
cargos-deployment-76f974589-422g2           1/1     Running   0          19m
cargos-deployment-76f974589-78tp2           1/1     Running   0          19m
componentes-deployment-6b8ff5b967-6gmdn     1/1     Running   0          19m
componentes-deployment-6b8ff5b967-b9jw9     1/1     Running   0          19m
contratacion-deployment-7bdd488f6-9bccb     1/1     Running   0          18m
contratacion-deployment-7bdd488f6-mn5sp     1/1     Running   0          18m
empleados-deployment-f95b4477d-js2tn        1/1     Running   0          18m
empleados-deployment-f95b4477d-sczdd        1/1     Running   0          18m
personas-deployment-74bbff8755-kts4x        1/1     Running   0          18m
personas-deployment-74bbff8755-mvcc4        1/1     Running   0          18m
politicas-acceso-deployment-6cb8b46fdb-2dz7z   1/1   Running   0          13m
politicas-acceso-deployment-6cb8b46fdb-f4574   1/1   Running   0          13m
redes-deployment-6f9db7445-fqdg5            1/1     Running   0          13m
redes-deployment-6f9db7445-vxvr9            1/1     Running   0          13m
soportes-deployment-5f6bbb476d-4zv9g        1/1     Running   0          3m34s
soportes-deployment-5f6bbb476d-5cdgf        1/1     Running   0          3m34s
usuarios-deployment-78c7dd56fd-lcqxv        1/1     Running   0          3m15s
usuarios-deployment-78c7dd56fd-tvkdf        1/1     Running   0          3m15s
felipe@ldap:~/unir-tfm/sinet_cloud/digital_ocean$
```