



Universidad Internacional de La Rioja
Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Desarrollo y Operaciones (DevOps)

**Actualización de un proyecto
implementando procedimientos DevOps
mediante Kubernetes y un entorno CI/CD**

Trabajo fin de estudio presentado por:	Jesús Barquero Cuadrado
Tipo de trabajo:	Trabajo Fin de Máster
Director/a:	David Fernández Gámez
Fecha:	18/09/2024

Resumen

Actualmente, las organizaciones de desarrollo de software deben adaptarse para brindar servicios de calidad a sus clientes en un entorno empresarial competitivo. Desde la aparición de las prácticas y metodologías DevOps, estas no han parado de ganar popularidad debido a su capacidad para fomentar la colaboración entre los equipos de Desarrollo (Dev) y Operaciones (Ops) garantizando el establecimiento de un ciclo de vida del software más rápido, confiable y continuamente mejorado.

Este trabajo de fin de máster (TFM) simulará una situación real dada en el mundo software empresarial en el que un ingeniero DevOps deberá mejorar un sistema de software existente de una empresa cliente. El sistema inicial es una aplicación web basada en microservicios y contenedores Docker administrados por el orquestador de AWS Elastic Container Service (ECS), que además carece de la automatización de su ciclo de vida. El objetivo de TFM será actualizar el orquestador hacia el servicio AWS Elastic Kubernetes Service (EKS) e implantar un entorno de integración y despliegue continuos (CI/CD) utilizando GitHub Actions para automatizar las fases del ciclo de vida del sistema y de la aplicación hacia la mejora continua.

Palabras clave: DevOps, ECS, EKS, CI/CD, GitHub

Abstract

Nowadays, software development organizations must adapt to provide quality services to their clients in a competitive business environment. Since the emergence of DevOps practices and methodologies, they have continuously gained popularity due to their ability to foster collaboration between Development (Dev) and Operations (Ops) teams, ensuring a faster, more reliable, and continuously improved software lifecycle.

This master's thesis (TFM) will simulate a real-world situation which is commonly encountered in the corporate software industry, where a DevOps engineer must improve an existing software system for a client company. The initial system is a web application based on microservices and Docker containers, which is managed by AWS Elastic Container Service (ECS), and lacks automation in its lifecycle. The main purpose of this TFM is to upgrade the orchestrator to AWS Elastic Kubernetes Service (EKS) and to implement a continuous integration and continuous deployment (CI/CD) environment by using GitHub Actions to automate the system and application lifecycle phases toward continuous improvement.

Keywords: DevOps, ECS, EKS, CI/CD, GitHub

Índice de contenidos

1.	Introducción	8
1.1.	Justificación del trabajo	9
1.2.	Planteamiento del problema	11
1.3.	Estructura de la memoria	12
1.3.1.	Introducción	12
1.3.2.	Contexto y estado del arte	12
1.3.3.	Objetivos y metodología de trabajo	12
1.3.4.	Desarrollo específico de la contribución	12
1.3.5.	Conclusiones y trabajo futuro	12
2.	Contexto y estado del arte	13
2.1.	Contextualización y antecedentes	13
2.1.1.	DevOps	13
2.1.2.	Contenedores y arquitectura de microservicios	17
2.1.3.	Tecnologías de orquestación de contenedores	19
2.1.4.	Importancia de los pipelines de integración y despliegue continuos (CI/CD) ..	28
2.1.5.	Herramientas CI/CD	31
2.2.	Trabajos relacionados	33
2.3.	Conclusiones del estado del arte	33
3.	Objetivos y metodología de trabajo	34
3.1.	Objetivo general	34
3.2.	Objetivos específicos	34
3.3.	Metodología del trabajo	34
4.	Desarrollo específico de la contribución	37
4.1.	Planificación / Análisis / Requisitos	37

4.1.1.	Requisitos del sistema a implantar.....	37
4.1.2.	Tecnologías y herramientas.....	38
4.2.	Descripción del sistema desarrollado / Implementación.....	42
4.2.1.	Presentación del sistema software actual.....	42
4.2.2.	Servicios que componen la aplicación.....	42
4.2.3.	Diagrama de arquitectura cloud en AWS	47
4.2.4.	Presentación del sistema final actualizado	50
4.2.5.	Proceso de desarrollo del sistema final.....	57
4.3.	Evaluación.....	78
4.3.1.	Medición de la mejora adquirida con el nuevo sistema	78
4.3.2.	Comprobación del correcto despliegue de los componentes del sistema	80
5.	Conclusiones y trabajo futuro	84
5.1.	Conclusiones	84
5.2.	Líneas de trabajo futuro	85
	Referencias bibliográficas.....	86

Índice de figuras

Figura 1.	<i>Ciclo DevOps</i>	14
Figura 2.	<i>Contenedores dentro de un sistema host.</i>	18
Figura 3.	<i>Funciones principales de los orquestadores de contenedores</i>	20
Figura 4.	<i>Arquitectura ECS.....</i>	23
Figura 5.	<i>Arquitectura EKS.....</i>	26
Figura 6.	<i>Estructura de pipeline CI/CD</i>	30
Figura 7.	<i>Estructura GitHub Actions.....</i>	41
Figura 8.	<i>Navegador Front-End.....</i>	43
Figura 9.	<i>Front parcelas</i>	44

Figura 10. <i>Parcela catastro</i>	44
--	----

Índice de tablas

No se encuentran elementos de tabla de ilustraciones.

Tabla 1. <i>Comparación herramientas CI/CD</i>	32
---	-----------

1. Introducción

Ante la necesidad que presentan las organizaciones dedicadas al desarrollo software por mantenerse relevantes en un entorno empresarial tan competitivo, la capacidad de poder entregar a sus clientes un servicio de calidad adaptado a sus necesidades en un mercado cambiante es vital. Por esta misma razón, cada vez más empresas demandan la adopción de DevOps, puesto que, gracias a que sus pilares de establecer una cultura de colaboración junto al uso de metodologías ágiles permiten que los equipos de desarrollo, operaciones y el resto de los departamentos estén alineados para cumplir sus objetivos de una manera holística junto a los de sus clientes, apoyados por un ciclo de vida automatizado, rápido, fiable y de mejora continua para la producción de software de calidad.

La adopción de DevOps puede producirse en mayor o menor medida debido a su amplio abanico de procesos, prácticas y metodologías, por lo que puede darse tanto en empresas grandes con un nivel de madurez elevado como en empresas pequeñas o startups. Es por esto que, en este Trabajo de Fin de Máster (TFM) se abordará un caso o situación común dada en el ámbito empresarial donde, ante un proyecto/sistema software inicial implantado en una empresa cliente, el ingeniero DevOps tendrá como objetivo aplicar procedimientos DevOps para mejorar su calidad implantando un nuevo sistema que cumpla las necesidades tecnológicas de la organización y permitiendo que este pueda adoptar mejoras de forma continua de una forma más rápida y a lo largo del tiempo.

El sistema inicial del que se parte es una aplicación web basada en microservicios desplegados a través de contenedores Docker gestionados por el servicio Elastic Container Service (ECS) dentro del proveedor cloud de AWS, cuya infraestructura ha sido creada a través de Terraform. El código de esta aplicación se encuentra contenido en un repositorio de GitHub sin ningún control del ciclo de vida de este. La aplicación o sistema final que se implementará seguirá utilizando la infraestructura de Amazon Web Services y buscará la actualización de la tecnología utilizada para la gestión de los contenedores de ECS hacia el orquestador de Kubernetes de AWS Elastic Kubernetes Service (EKS), mucho más potente, y la implantación de un entorno o pipeline CI/CD mediante GitHub Actions, antes inexistente, que automatice la integración y despliegue de código del proyecto junto a la entrega de nuevas características a los usuarios finales.

1.1. Justificación del trabajo

Este proyecto abordará un caso práctico donde se actualizará un proyecto software desarrollado previamente, donde se buscará que el proyecto inicial evolucione y aumente su calidad mediante la aplicación de procedimientos y tecnologías DevOps. Con la resolución de esta problemática se consolidarán los conocimientos adquiridos en este máster, y se aplicarán para resolver un caso práctico basado en la actualidad empresarial como DevOps donde se pide la transformación de un proyecto software hacia la mejora continua cumpliendo los requerimientos de una empresa cliente.

La primera actualización que se producirá será sobre el sistema de orquestación de contenedores ECS (Elastic Container Service) hacia Kubernetes, más concretamente, utilizando el servicio EKS (Elastic Kubernetes Service) de Amazon Web Services.

Actualmente, el servicio ECS es el orquestador que gestiona el clúster donde se despliegan los microservicios contenedores con las funcionalidades principales de la aplicación dentro del proveedor de Amazon Web Services. Al tratarse de una aplicación pequeña con pocas dependencias entre sus componentes, el uso de ECS es una elección muy adecuada para este sistema por varias razones, entre ellas: su integración total con los demás servicios de AWS, está completamente administrada por Amazon, permite una gestión simplificada de los contenedores a través de la interfaz web de AWS, sin costes añadidos más allá de los recursos consumidos por los nodos del clúster y cuenta con un escalado automático de manera confiable (Casero, 2023).

Sin embargo, debido al rápido avance del mercado tecnológico actual, la popularidad de Kubernetes dentro de las organizaciones no ha ido más que aumentando con los años, convirtiéndose en el estándar de facto actual para desplegar, gestionar y escalar aplicaciones contenerizadas en entornos nativos de nube basadas en microservicios (Mohindroo, 2023). Es por esta razón que se ha investigado sobre los beneficios y costes de actualizar hacia Kubernetes la tecnología de orquestación de contenedores del sistema actual. Por un lado, Kubernetes destaca por su arquitectura y su amplio abanico de componentes que permiten un mayor control sobre el clúster, haciéndolo muy flexible en cuanto a su configuración para satisfacer la lógica y requerimientos de aplicaciones multicontenedor por muy complejas que sean. Además, al ser una plataforma open-source con un robusto ecosistema de herramientas

y una comunidad activa de desarrolladores, esta proporciona una mayor flexibilidad reduciendo el riesgo de dependencia con el proveedor conocido como “vendor lock-in”. Asimismo, el propio proveedor de AWS posee su servicio de Kubernetes conocido como EKS administrado por Amazon, lo que le proporciona escalabilidad y una alta disponibilidad de servicio, además de estar integrado con los demás servicios de la nube (nOps, 2024).

Todas estas ventajas son razones muy positivas por las que se quiere actualizar el orquestador de la aplicación hacia EKS, no obstante, es necesario conocer los costes asociados a este cambio. En primer lugar, la curva de aprendizaje de Kubernetes es algo elevada y requiere de formación previa para explotar al máximo las capacidades del orquestador. También, al delegar la gestión del plano de control del clúster a AWS, supone un coste adicional asociado a su mantenimiento. Por otra parte, al tratarse de una aplicación pequeña con pocas dependencias entre sus componentes, la utilización de ECS dentro del sistema actual es una opción bastante adecuada al tamaño de la aplicación, y aunque EKS se adapta a cualquier tipo de tamaño de aplicación, destaca en la gestión de aplicaciones algo más complejas de mayor tamaño. Con todo esto presente, actualizar el sistema hacia EKS supondría una mejora en cuanto a calidad al utilizar la herramienta de orquestación más eficiente actualmente en el mercado, pudiendo aprovechar todo su potencial si la aplicación crece en tamaño y complejidad en un futuro (como podría ser en el ámbito empresarial de un cliente real). Por último, es de interés propio el aprender a utilizar esta herramienta como ingeniero DevOps.

La segunda actualización del sistema inicial será la instalación de un entorno de Integración y Despliegue Continuo (CI/CD) mediante GitHub Actions que gestione el ciclo de vida de la aplicación.

El hecho de no disponer de un entorno CI/CD en un proyecto software implica una serie de desventajas en una organización hasta el punto de marcar la diferencia entre el éxito o el fracaso del propio proyecto, lo que provocará la pérdida de valor del producto, de su calidad, la incapacidad de incorporar cambios de forma adaptativa a las necesidades del proyecto y la insatisfacción de los clientes. Por un lado, sin un pipeline CI/CD que automatice tareas como la compilación, pruebas y despliegue incrementará el riesgo de errores humanos en el código al realizarse estas de forma manual, reflejados después en producción. Además, los equipos de desarrollo y operaciones no tendrían una estructura donde integrar el código de manera frecuente, por lo que la colaboración dentro de los equipos sería ineficiente y sin una visión

clara del estado del proyecto. También, se verían afectados los tiempos de entrega de nuevas funcionalidades, la capacidad para corregir errores de forma rápida y la pérdida de productividad al tener que prestar más atención a las tareas repetitivas o resolución de errores en vez de enfocarse en las que aporten un mayor valor (GitLab, 2020).

El sistema actual no posee ningún tipo de control del ciclo de vida de su aplicación, por lo que, para evitar los problemas mencionados anteriormente, se implementará un pipeline a través de GitHub Actions que mejorará su calidad sustancialmente y agilizará tanto su fase de desarrollo como el despliegue. Como DevOps, esta es una tarea fundamental, ya que sin la incorporación de prácticas CI/CD a través de este pipeline, no será posible la aplicación del ciclo DevOps hacia la mejora continua de la aplicación y del proyecto.

1.2. Planteamiento del problema

Una vez justificado el problema de estudio y la motivación detrás de este trabajo, a continuación, se explicará la propuesta de solución para este caso práctico de actualización de un sistema software hacia un nuevo sistema para su mejora continua y tecnológica.

En primer lugar, puesto que la aplicación web se encuentra desplegada dentro de la infraestructura AWS a través de la herramienta IaC de Terraform con un grado elevado de “vendor-lock in”, el sistema final seguirá manteniendo este proveedor y utilizará el diverso ecosistema de servicios que provee Amazon Web Services. Por lo tanto, la primera actualización que se realizará será el cambio de orquestador de ECS hacia EKS. Esto aprovechará la infraestructura inicial y reducirá enormemente la complejidad de gestión del plano de control del clúster de Kubernetes al estar administrado por AWS. También, el nuevo sistema permitirá en un futuro que la aplicación de Kubernetes se pueda desplegar en entornos multinube si fuera necesaria su migración.

Una vez que la aplicación funcione correctamente con el nuevo orquestador, se pasará la a implementación de un pipeline CI/CD mediante GitHub Actions para la gestión del ciclo de vida de la aplicación y la aceleración de su despliegue en pocos minutos.

Finalmente, se comparará el sistema inicial con el nuevo para comprobar la mejora de calidad adquirida mediante diversas métricas como: tiempo requerido para desplegar la aplicación, tiempo de despliegue del sistema completo, automatización, reproducibilidad, escalabilidad y compatibilidad/portabilidad multicloud.

1.3. Estructura de la memoria

El documento se estructura de la siguiente manera:

1.3.1. Introducción

En este capítulo se presenta la necesidad de los procesos DevOps en las empresas software y se introduce la temática del proyecto sobre la actualización de un sistema software inicial y sus tecnologías hacia un sistema final donde se implantará un orquestador de contenedores EKS y un pipeline CI/CD utilizando procedimientos DevOps.

1.3.2. Contexto y estado del arte

El objetivo del capítulo es proporcionar un contexto adecuado introductorio hacia el desarrollo del proyecto a través de un estado del arte donde se aprenderá sobre DevOps, contenedores, tecnologías de orquestación y la importancia de los entornos CI/CD.

1.3.3. Objetivos y metodología de trabajo

El capítulo establece el objetivo a conseguir con este proyecto, los objetivos específicos en los que se dividen para cumplir con el objetivo principal y una metodología para el desarrollo del proyecto.

1.3.4. Desarrollo específico de la contribución

Este capítulo comienza con la presentación de los requisitos, herramientas y tecnologías que se utilizarán durante el desarrollo. Seguidamente, se explicarán todos los detalles del sistema inicial que se va a actualizar y el proceso de desarrollo seguido para la implantación del sistema final en forma de diversos diagramas y detalles de implementación. Finalmente, se evaluarán los resultados obtenidos y se expondrán las mejoras conseguidas en el nuevo sistema.

1.3.5. Conclusiones y trabajo futuro

En este apartado resumirá el trabajo realizado y se relacionará con los objetivos establecidos para determinar si se han cumplido o no. También irá acompañado con unas posibles líneas de trabajo futuro.

2. Contexto y estado del arte

2.1. Contextualización y antecedentes

2.1.1. DevOps

2.1.1.1. Qué es DevOps

El entorno cambiante y altamente competitivo de la industria del software al que se enfrentan las empresas dedicadas al desarrollo de software exige a estas operar con agilidad y flexibilidad para adaptarse rápidamente a las demandas de los clientes, de forma que les puedan entregar servicios de forma continua en tiempos cada vez más reducidos. No solo deben adaptarse a los clientes, sino también a las condiciones del mercado, cambios en el ámbito legal y el resto de la competencia en el ámbito empresarial. Para lograr dicha tarea, los departamentos de Desarrollo (Dev) y Operaciones (Ops) necesitan estar perfectamente coordinados, una tarea difícil y que muchas empresas no logran conseguir por sí solas (Sanjurjo Royo, 2022).

En este mismo contexto, aparece el paradigma DevOps tomando cada vez más fuerza y relevancia en las empresas con el paso de los años. Tiene sus bases en los principios “Lean” y ágiles. El pensamiento o metodología Lean primero busca optimizar los procesos de gestión y productivos de la empresa, para conseguir con menos recursos una mayor eficiencia al maximizar todo lo posible la reducción de la inversión, el tiempo y esfuerzo (Redacción APD, 2023). Los enfoques ágiles buscan siempre entregar productos que busquen la excelencia y valor a sus clientes, asegurando el rápido retorno de inversión (ROI), y que estos productos se adapten a los cambios de las necesidades del cliente a lo largo del tiempo junto a los cambios permanentes de los medios gracias a un desarrollo de proyectos incremental e iterativo (Guerrero, Certuche Díaz, Zúñiga, & Pardo, 2019). Otras de las prácticas más comunes en los equipos DevOps es la automatización de todos los procesos posibles del ciclo de vida de un proyecto software, la Integración Continua (CI) y el Despliegue Continuo (CD).

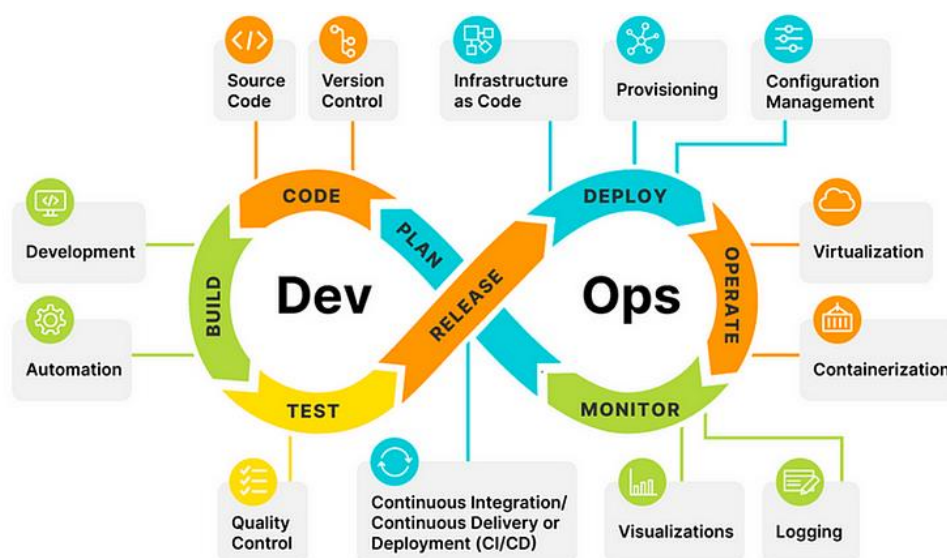
DevOps tiene como objetivo principal la unificación de los procesos realizados por los departamentos de Desarrollo (Dev) y Operaciones (Ops), en un solo proceso integrado y continuo, además de derribar los silos entre los departamentos Dev y Ops implantando una cultura de cooperación y comunicación. Desde el enfoque DevOps, se busca aprovechar la

experiencia y conocimiento de las personas, los procesos y la tecnología para estimular la colaboración y la innovación en todo el proceso de desarrollo y lanzamiento del software de manera rápida, iterativa, frecuente y confiable para aprovechar las oportunidades de mercado más temprano, sin sacrificar ni la calidad ni el valor entregado a los clientes. (Guerrero et al., 2019)

2.1.1.2. Ciclo DevOps

El ciclo DevOps es un conjunto de fases al que se somete el software hacia la mejora continua del mismo mediante la repetición iterativa del ciclo completo. Es una visión muy distinta a la metodología de cascada tradicional para la gestión de proyectos, ya que diferentes tareas y procesos pueden estar comprendidos en diferentes fases de forma superpuesta y natural, siempre buscando la mejora y el aumentar el valor del producto. Como puede verse en la “Figura 1”, este ciclo comienza con el equipo de Desarrollo (Dev) abordando las fases de Planificación (Plan), Codificación (Code), Compilación (Build) y Pruebas (Test). Seguidamente, el equipo de Operaciones (Ops) se encargará de las fases de Liberación (Release), Despliegue (Deploy), puesta en Funcionamiento (Operate) de la nueva versión y la Monitorización (Monitor) de esta. Una vez finalizada la iteración o ciclo, comenzará la siguiente en la fase de Planificación (Redondo & Cárdenas, 2022).

Figura 1. *Ciclo DevOps*



Fuente: (Group, 2023)

A continuación, se explicarán en qué consisten cada una de las fases:

1. Planificación

Esta etapa se enfoca en identificar y comprender los requisitos del cliente, buscando siempre establecer un canal de comunicación continuo con estos. Como resultado se obtiene una hoja de ruta del producto para orientar el desarrollo a lo largo del tiempo, donde se planificarán que funcionalidades y objetivos han de cumplirse a corto, medio y largo.

2. Codificación

Comienza el diseño del software y la creación del código, estableciendo las herramientas y tecnologías que faciliten el proceso de desarrollo, buscando mantener un estilo de código coherente, evitar fallos de seguridad comunes y anti-patrones de código.

3. Compilación

Es en este punto donde entra en acción DevOps poniendo en práctica la Integración Continua (CI). Cada desarrollador cuando finaliza una tarea envía su código al repositorio compartido con el resto de los desarrolladores mediante una solicitud de extracción “pull request” para fusionarlo con la rama de código común en la que esté trabajando. Otro desarrollador revisa y aprueba los cambios de la solicitud, evitando así problemas de manera temprana ya sean por errores o incompatibilidades entre las versiones del desarrollador y la rama.

Paralelamente, la solicitud de extracción activa el proceso automatizado de pruebas unitarias, de integración y de extremo a extremo para detectar cualquier regresión. Si la compilación o cualquiera de las pruebas falla, la solicitud de extracción se rechaza y se notificará al desarrollador para que este resuelva el error. Como se están continuamente verificando los cambios de código del repositorio común, ejecutando compilaciones y pruebas de forma automatizada, se minimizan los problemas de integración y se detectan los errores al principio del ciclo de vida del desarrollo.

4. Pruebas

Al finalizar la compilación, de forma automática se despliega en un entorno de pruebas para llevar a cabo pruebas más exhaustivas que en la fase anterior, tanto manuales como automatizadas. Entre las pruebas manuales se encuentran las pruebas de aceptación del

usuario (UAT), donde los usuarios pueden probar la versión de una forma más directa para detectar problemas o mejoras antes de la implementación en producción. Simultáneamente, las pruebas automatizadas pueden ejecutar análisis de seguridad en la aplicación comprobando también si se cumplen las mejores prácticas, verificar cambios, evaluar el rendimiento de la aplicación o realizar pruebas de carga. Las pruebas realizadas durante esta fase pueden variar según la organización y la aplicación, las cuales se ejecutan simultáneamente sin interrumpir el flujo de trabajo de los desarrolladores o afectar al entorno de producción, pudiendo incorporar nuevas pruebas según se necesiten.

5. Liberación

Comienza cuando la compilación está lista para su despliegue en el entorno de producción. Como el código ha pasado por una serie de pruebas manuales y automatizadas, los problemas críticos y las regresiones son poco probables, por lo que el equipo de operaciones puede trabajar con mucha más seguridad. En este punto, las organizaciones pueden incluso implementar otras compilaciones en conjunto que llegan a esta fase del pipeline, con el objetivo de conseguir múltiples lanzamientos de sus productos cuando el nivel de madurez adecuado al implementar DevOps.

6. Despliegue

Lanzamiento a producción, pudiendo automatizarse desde la fase anterior para asegurar el Despliegue Continuo (CD). Como para el entorno de producción se utiliza una infraestructura como código idéntico al entorno de prueba, cualquier problema que se produzca con la nueva versión puede solucionarse mientras que las solicitudes se redirigen a este entorno o a uno de preproducción.

7. Funcionamiento

Con la nueva versión en funcionamiento, el equipo de Operaciones trabaja de manera para asegurar que todo funcione sin problemas, prestando especial atención a cómo reciben el producto los clientes puesto que es vital obtener su retroalimentación para orientar el desarrollo futuro del producto.

8. Monitorización

Se recopilan datos, se analiza el comportamiento de la aplicación en cuanto a rendimiento y los errores que experimentan los clientes al usar la aplicación. Toda esta información es

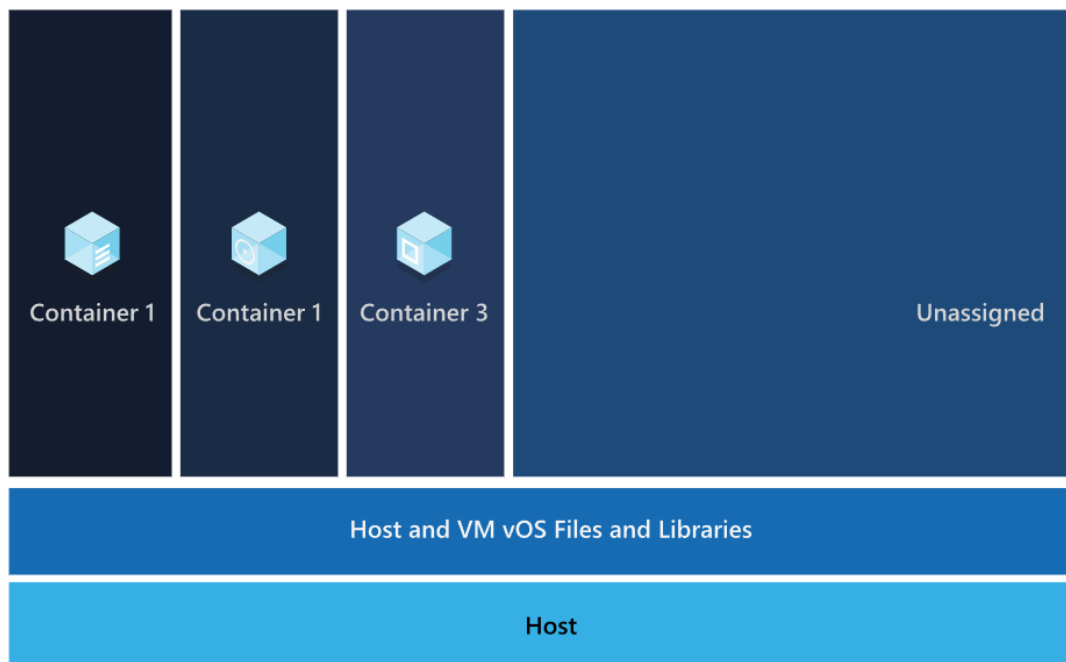
enviada al administrador del producto y al equipo de desarrollo, dando final al ciclo del proceso de esta iteración.

Una vez terminado el ciclo, comienza el siguiente. Es un proceso de evolución continua del producto a lo largo de su vida útil, que termina únicamente cuando se pasa a otro proyecto o cuando ya no se necesitan nuevas versiones.

2.1.2. Contenedores y arquitectura de microservicios

La aparición de nuevos patrones arquitectónicos de software, como los microservicios, han mejorado el modularidad de las aplicaciones junto a su desarrollo, prueba, escalado y reemplazo de componentes. Esto ha provocado la aparición de nuevas prácticas como las metodologías y herramientas de DevOps, que promueven una mejor cooperación entre los equipos de desarrollo de software y operaciones y respaldan la automatización y el monitoreo en todo el ciclo de vida de construcción de software. Esto ha afectado positivamente a las empresas del sector TI, y las tecnologías basadas en contenedores han desempeñado un papel crucial al permitir el despliegue rápido de microservicios y su escalabilidad con bajo costo adicional (al Jawarneh et al., 2019).

Los contenedores son una de las formas de virtualización más utilizadas para el despliegue de aplicaciones ya que, al contrario que las máquinas virtuales, estos no reservan una porción de recursos hardware del equipo donde se encuentran ejecutándose, sino que todos los recursos se encuentran compartidos por todos los contenedores del sistema, como puede observarse en la “Figura 2”, pudiendo ser tratados como procesos individuales, ligeros y más rápidos en su ejecución.

Figura 2. *Contenedores dentro de un sistema host.*

Fuente: (Microsoft, 2024).

Un contenedor es una unidad de software estandarizada que empaqueta el código de una aplicación junto a todas sus bibliotecas, archivos de configuración y dependencias que se necesitan para permitir su ejecución de una forma consistente, rápida e independiente de donde se ejecute. Esto permite a los desarrolladores implementar, probar y desplegar aplicaciones en cualquier entorno informático. Tienen la característica de ser livianos además de proporcionar una infraestructura que no cambia al empaquetar completamente el software de una aplicación. Esta aplicación o servicio se transforma en una imagen de contenedor, con la que después puede probarse como una unidad de software individual mediante una instancia de esa imagen de contenedor en un sistema operativo host, pudiendo además compartir estas imágenes con otros desarrolladores mediante su subida a repositorios (Microsoft, 2024).

Los contenedores son ampliamente utilizados en las arquitecturas software basadas en microservicios. Este tipo de arquitecturas permiten la construcción de aplicaciones distribuidas a través de un amplio conjunto de servicios desacoplados, desplegados con independencia unos de otros y que interactúan entre sí de manera coordinada. Una de las características más importantes que aportan los microservicios es la escalabilidad puesto que, al igual que se desarrollan de manera independiente unos de otros, también pueden ser

escalados independientemente permitiendo trabajar con cargas de trabajo elevadas cuando se necesite. Los contenedores proporcionan un medio ideal para la implementación de los microservicios debido a su capacidad de empaquetar las aplicaciones y sus dependencias que, junto a su coste ligero de recursos hardware y su rapidez de implementación, potencia la escalabilidad horizontal de manera natural y eficiente (al Jawarneh et al., 2019; Murazzo et al., 2019).

2.1.3. Tecnologías de orquestación de contenedores

Uno de los principales problemas es la gestión de contenedores de aplicaciones, sobre todo a gran escala, en las que multitud de ellos se encuentran desplegados al mismo tiempo con interdependencias complejas. Además, los contenedores no solo pueden estar ejecutándose en un solo host, lo más normal es que distribuyan en varias instancias o nodos de distinto tipo y capacidad de manera descentralizada, los cuales escalan también en número según aumenta el tamaño de la aplicación. Esto hace que la gestión manual de los contenedores se complique hasta el punto de volverse inviable.

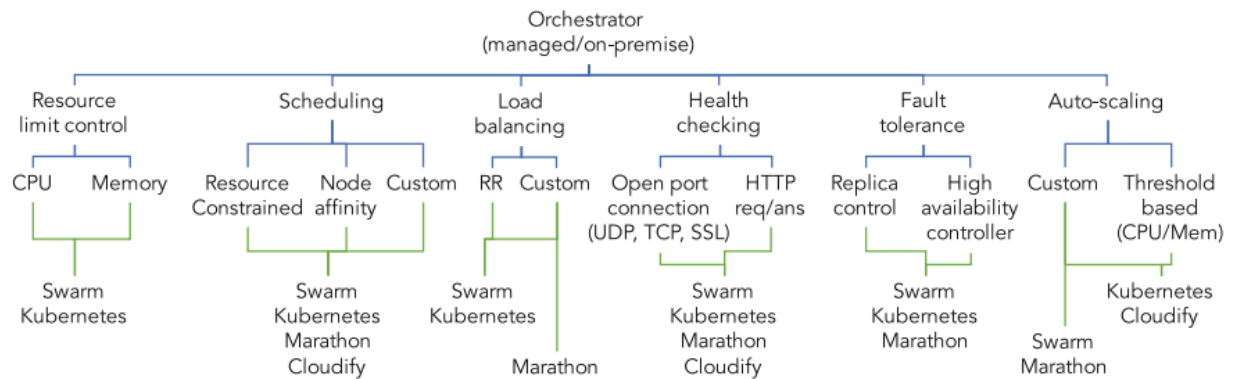
Al aumentar el número de nodos de cómputo de contenedores, los equipos de desarrollo y operaciones necesitan soluciones para convertir los nodos independientes en clústeres para gestionar sus contenedores, la cantidad de recursos que consumen y la capacidad de recursos que se necesita de los nodos de forma eficiente. Buscan una solución robusta proporcione una gestión completa de sus cargas de trabajo de contenedores en esos clústeres. Es aquí donde surgen las tecnologías de orquestación de contenedores (Amazon Web Services, 2020c).

2.1.3.1. Orquestación de contenedores

Los orquestadores de contenedores permiten a los proveedores de servicios en la nube y de aplicaciones la capacidad de seleccionar, desplegar, monitorizar y gestionar de forma dinámica aplicaciones multicontenedor. Aísla al usuario de los detalles de implementación y automatiza los procesos del plano de control y operaciones de los nodos, reduciendo la intervención manual y permitiendo que los desarrolladores puedan trabajar con contenedores

de manera más eficiente. En la “Figura 3” pueden observarse las principales características y funciones que ofrecen los orquestadores según (Casalicchio, 2019):

Figura 3. Funciones principales de los orquestadores de contenedores



Fuente: (Casalicchio, 2019)

Control del límite de recursos: permite reservar una cantidad específica de CPU y memoria para un contenedor. Estas restricciones son útiles durante la toma de decisiones para planificación de procesos y reducir posibles interferencias en el rendimiento entre contenedores. Aunque un contenedor puede utilizar todos los recursos disponibles en el sistema host subyacente, los orquestadores cuentan con mecanismos propios para regular el control de estos y proporcionan una API de gestión de los contenedores para limitar la cantidad de memoria y CPU utilizada.

Planificación de las políticas que deciden en un instante de tiempo dado qué cantidad de contenedores debe ser desplegada y en qué nodos debe hacerlo. Esta tarea se realiza a través de una entidad planificadora dentro del orquestador, pudiendo ser utilizado incluso como un componente externo integrado al clúster en función de su complejidad. El tipo de políticas pueden ser en función de restricciones de recursos que pueden consumir, la afinidad del nodo para un trabajo, ambas u otra regla personalizada.

Los balanceadores de carga distribuyen el tráfico o carga de la aplicación entre múltiples instancias de contenedores. Round-robin es la política implementada por defecto, pero se pueden establecer políticas más complejas y personalizadas mediante balanceadores de carga externos, como por ejemplo los proporcionados por los proveedores en la nube.

La tolerancia a fallos es gestionada mediante un sistema de control de réplicas y/o mediante un controlador de alta disponibilidad. El control de réplicas permite especificar y mantener el número deseado de contenedores en funcionamiento a través de los nodos del clúster, asegurándose de que la aplicación siga en funcionamiento, aunque algún contenedor o nodo falle.

La comprobación de salud o “Health check” se consigue mediante el envío de peticiones a los contenedores, las cuales proporcionarán información de su estado en las respuestas. Los tipos de peticiones incluyen la comprobación de establecimiento de conexiones en puertos TCP/UDP/SSH y la comprobación de recepción y respuesta de solicitudes HTTP.

El autoescalado permite agregar y eliminar contenedores automáticamente. Las políticas implementadas se suelen basar en umbrales de límite de recursos, las cuales se pueden personalizar. Igual que con otras funciones mencionadas anteriormente, es posible integrar un autoescalador más sofisticado y administrado completamente por Amazon, Azure, Google... y otros proveedores cloud.

2.1.3.2. Amazon Elastic Container Service (ECS)

Amazon ECS (Amazon Web Services, 2023a) es un servicio ofrecido por el proveedor de AWS para la orquestación de contenedores Docker. Está totalmente administrado por Amazon, por lo que el usuario no tendrá la responsabilidad de administrar el plano de control del clúster, además de facilitar las tareas de implementación y escalado de aplicaciones contenerizadas. Con ECS, las cargas de trabajo pueden ejecutarse en instancias EC2 “Amazon Elastic Compute Cloud” (Amazon Web Services, 2020b) a través del servicio AWS Fargate (Amazon Web Services, 2024b) para la ejecución de contenedores sin servidor o “serverless”.

Son numerosos beneficios de utilizar ECS como orquestador de contenedores (Novotný, 2023), entre ellos:

- **Facilidad de uso:** servicio diseñado desde la simplicidad para que el usuario no se preocupe en exceso por cuestiones relacionadas a la computación, red o configuración de seguridad, al mismo tiempo que AWS facilita su gestión.
- **Integración total con los demás servicios de AWS:** forma parte del ecosistema de servicios de AWS, por lo que se integra con ellos perfectamente aprovechando todas

sus ventajas, como por ejemplo Elastic Load Balancer (Amazon Web Services, 2023b) para redirigir el tráfico de la aplicación, entre otros.

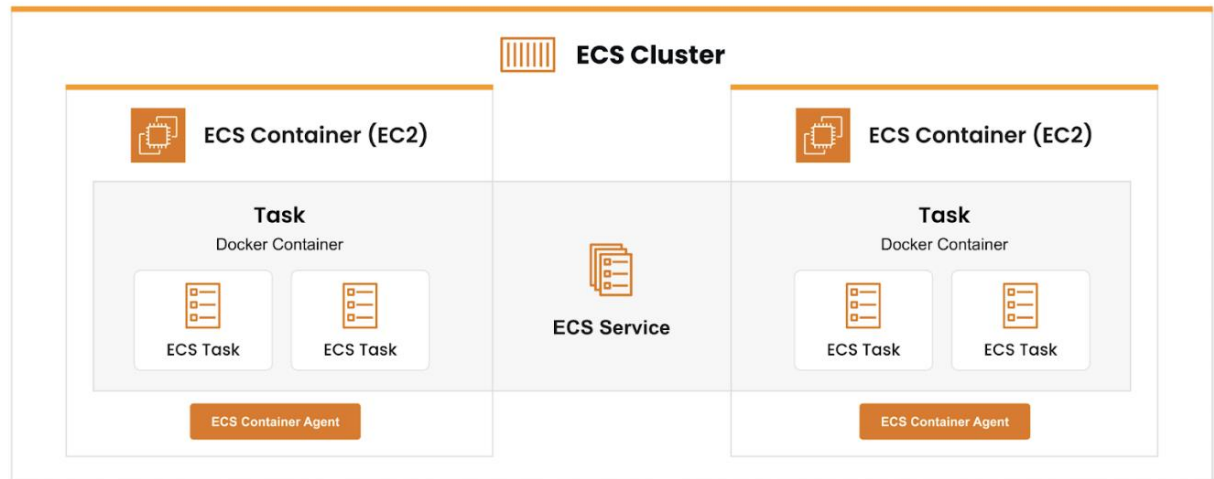
- **Escalado flexible de las aplicaciones:** capacidad de escalado de las aplicaciones tanto manual como de forma autoescalada en función de la demanda a través de políticas definidas sobre el clúster con reglas personalizadas de escalado y capacidad.
- **Coste reducido:** no existe un coste adicional por la utilización de ECS, solo se paga por los recursos consumidos por los demás servicios utilizados por este como EC2 o Fargate.

Al mismo tiempo, algunas de las desventajas al utilizar ECS son las siguientes:

- **Vendor lock-in:** el servicio de ECS solo funciona dentro del proveedor de AWS, lo que dificulta la migración de las aplicaciones hacia otras infraestructuras y no se podrán implementar en un entorno multinube.
- **Menor control sobre el clúster:** como contrapunto de ser un servicio gestionado por AWS, limita el control que tiene el usuario para realizar ciertas acciones o tareas de administración del clúster. Esta limitación puede ser una preocupación si se necesita una mayor personalización en la configuración sobre los contenedores o satisfacer requisitos específicos de infraestructura.

En cuanto a los componentes que forman la arquitectura de ECS, tal y como puede observarse en la “Figura 4”, son los siguientes:

Figura 4. *Arquitectura ECS*



Fuente: (nOps, 2024)

Las **tareas** son instancias creadas a partir de una **definición de tarea** dentro del clúster. Estas definiciones de tareas son archivos JSON donde se describen los contenedores, almacenados previamente en algún repositorio como GitHub (GitHub, 2024) o ECR “Amazon Elastic Container Registry” (Amazon Web Services, 2020a), que forman la aplicación junto a los parámetros de lanzamiento como la especificación de la imagen a utilizar, abrir puertos, límite de recursos que consumirá, volúmenes de datos que utilizarán los contenedores...

Estas tareas pueden ejecutarse de forma independiente o ser también ejecutadas como un **servicio**, los cuales mantendrán en funcionamiento una o varias tareas. En caso de que alguna tarea falle o se pare, el planificador del clúster lanzará otra instancia de esa tarea. Lo reemplaza y mantendrá el número deseado de tareas definidas para ese servicio.

Dentro de cada nodo del clúster, también llamado instancia de contenedor, se ejecuta un **agente de contenedor**. El agente envía información al plano de control de ECS acerca de la utilización de recursos y las tareas que se encuentran en funcionamiento. Recibe las solicitudes del servicio ECS para ejecutar y detener las tareas dentro de las instancias.

2.1.3.3. Kubernetes y AWS Elastic Kubernetes Service (EKS)

Amazon EKS (Amazon Web Services, 2017) es el servicio ofrecido por AWS como “Kubernetes-as-a-Service” en la nube para desplegar, ejecutar y gestionar aplicaciones multicontenedor que utilizan Kubernetes. EKS administra de forma automática la escalabilidad y disponibilidad de los nodos del plano de control del clúster encargados de la planificación de los contenedores, gestión de la disponibilidad de las aplicaciones y almacenar los datos de configuración del clúster, entre otras. Además, libera al usuario de la realización de tareas complejas asociadas al levantamiento y mantenimiento de un clúster como actualizaciones, aprovisionamiento de los nodos... También, al igual que ECS, permite la ejecución de los contenedores dentro de instancias EC2 o a través del servicio AWS Fargate.

A su vez, Kubernetes (Kubernetes, 2022) es la plataforma de código abierto para la orquestación de contenedores más utilizado actualmente en el mercado para la implementación, escalado y administración de aplicaciones, además de facilitar la automatización gran parte de las tareas asociadas a estos procesos. Destaca por su portabilidad entre proveedores de infraestructuras, capacidad de configuración declarativa y extensible gracias a su amplio ecosistema de herramientas y servicios que ofrece. Orquesta la infraestructura de cómputo, redes y almacenamiento de las cargas de trabajo para que los usuarios no tengan que hacerlo, y como su diseño mezcla características de Plataforma como Servicio (PaaS) e Infraestructura como Servicio (IaaS), puede satisfacer una amplia variedad de requisitos de aplicación.

Algunos de los beneficios de la utilización de EKS son los siguientes (Novotný, 2023):

- **Ecosistema de Kubernetes:** EKS aprovecha el ecosistema de herramientas, plugins e integraciones entre estas que aumentan la potencia operacional y personalización del clúster.
- **Integración con los demás servicios de AWS:** aprovecha todas las ventajas de la nube de AWS para integrarse con los demás servicios de esta, que combinado con el ecosistema de Kubernetes, permite adaptarse a las necesidades de las aplicaciones por muy complejas que sean.
- **Alta disponibilidad y escalabilidad:** la administración del plano de control del clúster dentro de AWS garantiza también la alta disponibilidad de este y facilita el

autoescalado de las aplicaciones alrededor de las múltiples zonas de disponibilidad, asegurando la alta disponibilidad del servicio.

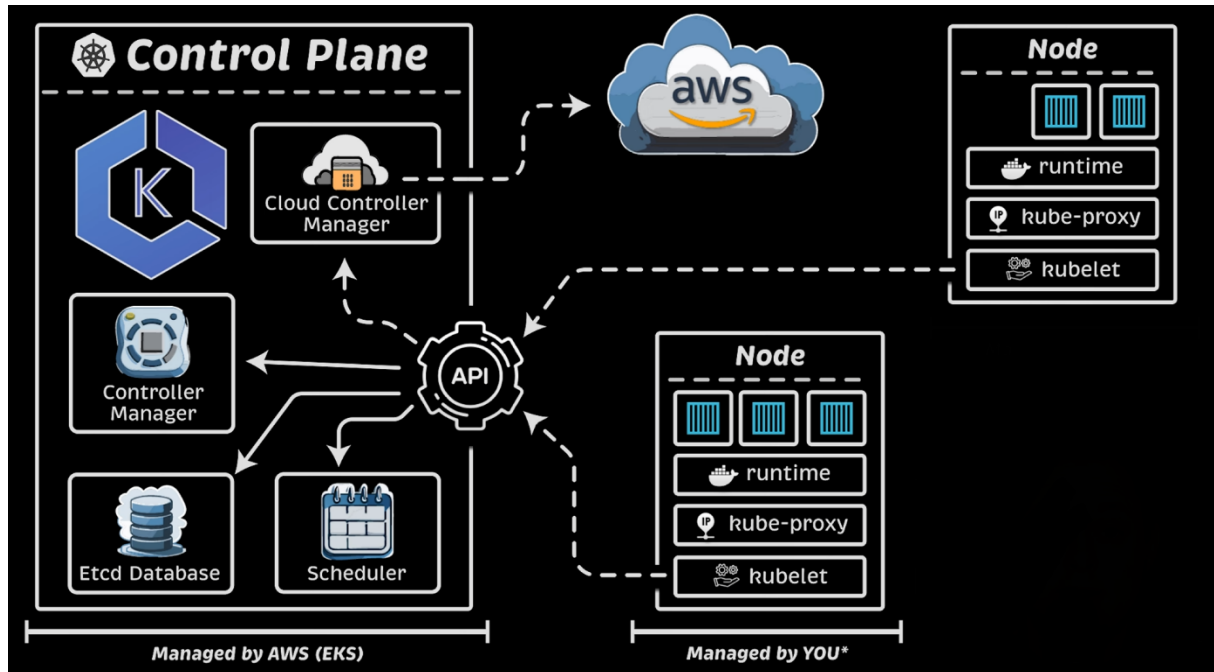
- **Mayor control sobre el clúster:** flexibilidad en la configuración del clúster que ofrece mayor control sobre cómo y dónde se despliegan los contenedores a través de “pods” en diferentes “namespaces” o espacios de nombres, los cuales separan lógicamente los recursos del clúster para organizarlos según el tipo de entorno o relaciones entre estos. También permite la configuración de endpoints para los servicios, creación de certificados, gestión de usuarios y permisos basados en roles y el uso de etiquetas para gestionar y organizar los recursos de manera granular.
- **Menor “vendor lock-in”:** aunque EKS es un servicio dentro de la nube de AWS, las aplicaciones orquestadas por Kubernetes permiten el despliegue híbrido, tanto multinube como en centros de datos locales.

Por otra parte, también existe una serie de desventajas que se deben tener en cuenta:

- **Complejidad de uso:** configurar y gestionar un clúster EKS no es tarea sencilla para los usuarios que no están familiarizados con el uso de Kubernetes. Su uso requiere de conocimientos previos sobre sus componentes, uso de las redes y con qué servicios de AWS se integra.
- **Sobrecarga operativa:** aunque EKS abstrae parte de la gestión de la infraestructura del plano de control, el usuario es responsable de las operaciones y mantenimiento de los nodos trabajadores.
- **Costo adicional:** el mantenimiento del clúster por parte de AWS incluye un coste por hora asociado.

Dentro de la arquitectura de un clúster EKS se encuentran los siguientes componentes (Kubernetes, 2020b):

Figura 5. Arquitectura EKS



Fuente: (Anton Putra, 2024)

Como puede observarse en la “Figura 5”, entre los componentes se distinguen los componentes de plano de control y los pertenecientes a los nodos trabajadores:

Componentes del plano de control

Dentro del plano de control se encuentran los nodos maestros, y sus componentes toman decisiones que afectan al clúster de manera global, por ejemplo, la planificación de los trabajos dentro de los nodos trabajadores o detectar y responder a eventos del clúster como mantener el número de réplicas esperadas de un “pod”, unidad mínima que puede contener uno o más contenedores. Estos componentes pueden ejecutarse en cualquier nodo del clúster, pero se separan las responsabilidades mediante esta distinción de nodo maestro para no ejecutar contenedores de usuarios dentro de ellos. Así mismo, se garantiza la alta disponibilidad del clúster mediante la ejecución de varios nodos maestros al mismo tiempo.

1. **APIserver:** expone la API de Kubernetes como un punto de entrada para la gestión de las solicitudes y operaciones de administración del clúster. Sirve para interactuar con el clúster, tanto desde fuera a través del cliente “kubectl” (Kubernetes, 2024) como desde los demás componentes internos.

2. **Etcd**: almacén de tipo clave-valor persistente, distribuido y consistente que guarda toda la información del clúster.
3. **Scheduler**: componente que actúa como planificador que asigna los pods a los nodos para ejecutarse en función de factores como requisitos de recursos, afinidad, políticas... entre otros.
4. **Controller Manager**: ejecuta los controladores de Kubernetes, cada uno a través de un proceso independiente, para mantener el estado deseado del clúster a través de llamadas al APIserver para obtener el estado actual y qué cambios realizar:
 - a. **Controlador de Nodo**: monitoriza los nodos para detectar si alguno deja de funcionar y responder para desplegar los pods en el resto de los nodos.
 - b. **Controlador de Replicación**: mantiene el número de pods deseado y eliminar los que han fallado.
 - c. **Controlador de Endpoint**: proporciona las rutas formadas a partir de la conexión entre los pods y los servicios los exponen.
 - d. **Controlador de Servicio**: creación de cuentas y tokens de acceso a la API para los nuevos “namespaces” o espacios lógicos dentro del clúster.
5. **Cloud Controller Manager**: ejecuta los controladores que interactúan con los proveedores en la nube. Permite que el código de Kubernetes evolucione de manera independiente al del proveedor en la nube a través de un binario separado. De esta manera, cada funcionalidad específica de cada proveedor deberá ser mantenido por este y enlazado con el binario al lanzar Kubernetes. Los controladores afectados que dependen del proveedor en la nube son los siguientes:
 - a. **Controlador de Nodos**: detecta y actúa ante la caída de algún nodo.
 - b. **Controlador de Rutas**: configuración de las rutas con destino a la infraestructura de nube específica.
 - c. **Controlador de Servicios**: gestiona los balanceadores de carga en la nube utilizados por el clúster.
 - d. **Controlador de Volúmenes**: interactúa con el proveedor para orquestar la creación, montaje y conexión de volúmenes.

Componentes de nodo:

Fuera del plano de control, el resto de los nodos del clúster se conocen como nodos trabajadores y son las unidades de procesamiento de Kubernetes. Dentro de ellos se ejecutan las cargas de trabajo gestionadas por los nodos maestros y sus componentes se encargan de las siguientes tareas:

1. **Kubelet:** agente ejecutado en todos los nodos del clúster. A partir de un conjunto de especificaciones de pod o “PodSpecs” proporcionada por los nodos maestros, garantiza que los contenedores descritos dentro de ellos se encuentren en buen estado y funcionando.
2. **Kube-proxy:** encargado de la gestión de las redes. Establece las reglas de red en el anfitrión, realiza el reenvío entre los nodos del clúster y los mantiene interconectados.
3. **Runtime de contenedores:** es el entorno de ejecución para la ejecución de los contenedores dentro de los nodos, entre ellos, Kubernetes soporta Docker, containerd, rktlet o cualquier implementación de la interfaz de runtime propia de Kubernetes (Kubernetes CRI).

2.1.4. Importancia de los pipelines de integración y despliegue continuos (CI/CD)

Las empresas se encuentran constantemente dedicando sus esfuerzos y recursos en desarrollar y entregar software de alta calidad a un ritmo acelerado debido al aumento en la competencia existente en el mercado software. Por esta razón, prácticas como la Integración Continua (CI), Despliegue Continuo (CD) y/o Entrega Continua (CDE), en su conjunto conocido como la implementación de un pipeline de CI/CD, se utilizan para ayudar a las organizaciones acelerando el desarrollo y entrega de nuevas características software sin comprometer la calidad del mismo. Estas prácticas proporcionan beneficios como los que se expone en (Shahin, Ali Babar, & Zhu, 2017):

1. Rápida obtención de feedback o retroalimentación procedente tanto del proceso de desarrollo como de los propios clientes que proporcionan de información valiosa utilizable en las siguientes iteraciones del producto.
2. Lanzamientos o iteraciones frecuentes y confiables del producto software que aumentan la satisfacción del cliente, produciendo además una mejora continua del mismo producto.

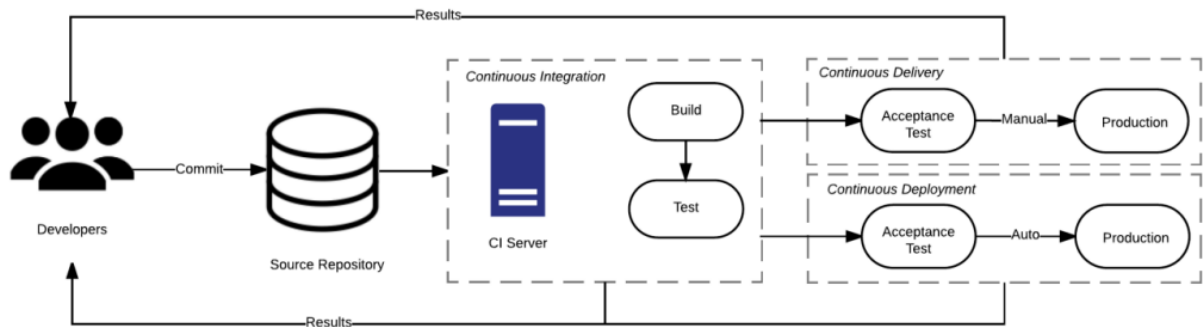
3. Fortalecimiento de la conexión entre los equipos de desarrollo y operaciones al agilizar el ciclo de desarrollo, reduciendo también los errores manuales gracias a la automatización de todo el proceso del pipeline CI/CD.

La utilización de los pipelines CI/CD se alinea perfectamente con las metodologías ágiles y DevOps, formando parte de sus prácticas, principios y procedimientos en el desarrollo de software (Lewandowski, 2023):

4. Los equipos de desarrollo y operaciones abrazan los valores de la cultura de colaboración, mejora continua y adaptabilidad a los cambios gracias a la disponibilidad de un entorno de CI/CD común para compilar, probar, integrar y desplegar el código de la aplicación, promoviendo así una sensación de responsabilidad compartida que rompe los silos que separan a estos equipos.
5. Énfasis en iteraciones cortas del producto software, en las que cada una de ellas añaden valor al producto software a través de la retroalimentación proporcionada por la monitorización y los clientes.
6. Búsqueda de la automatización de todos los procesos posibles del pipeline, que junto a las pruebas automatizadas y herramientas de monitorización mejoran la eficiencia y confiabilidad de ciclos de desarrollo ágiles permitiendo entregas rápidas y de calidad.
7. La colaboración entre equipos y resolución de fallos aumenta gracias a la visibilidad extremo a extremo de todos los procesos realizados durante el ciclo de vida del producto software, permitiendo identificar cuellos de botella, medir el progreso y tomar decisiones basados en la información obtenida para la mejora continua del producto.

En resumen, la CI/CD es una parte fundamental de la metodología DevOps puesto que ambas buscan fomentar la colaboración entre los equipos de desarrollo y operaciones, la automatización de procesos para integrar código y agilizar la implementación de nuevas funcionalidades o corrección de errores para pasar rápidamente de un entorno de desarrollo al de producción y así generar valor para el usuario (Red Hat, 2022).

A continuación, se describirán las prácticas o procesos que se han mencionado anteriormente y que forman parte de un pipeline CI/CD como puede apreciarse en la “Figura 2”.

Figura 6. Estructura de pipeline CI/CD

Fuente: (Shahin et al., 2017).

Integración Continua (CI)

La parte “CI” o “Integración Continua” del pipeline CI/CD consiste en el proceso de automatización con el que los desarrolladores fusionan los cambios de código en un mismo repositorio compartido de forma frecuente y eficiente. Es el primer paso hacia el Despliegue Continuo (CD), se asegura que varios desarrolladores puedan trabajar de forma simultánea en un mismo proyecto o funcionalidad software. Durante el proceso pueden surgir problemas de incompatibilidad entre cambios puesto que, cada desarrollador trabaja independientemente sobre una copia de la rama maestra del código, por lo que pueden aparecer problemas de funcionalidad y errores al fusionar estos cambios de nuevo con la rama principal. Por esta razón, para que la integración continua se realice de forma exitosa es necesario que las modificaciones se validen a través de compilaciones y la ejecución de pruebas automatizadas que garantizan que los cambios no tengan errores antes de la integración junto al código del repositorio. Todos los cambios deben validarse de esta manera y pasar todas las pruebas, y si se detecta una incompatibilidad entre el código nuevo y el actual del repositorio, el CI ayudará a detectar y resolver estos errores rápidamente (Red Hat, 2022).

Entrega Continua (CDE)

Es el siguiente paso después de la CI, incluso una extensión de esta, puesto que es el proceso encargado de pasar todos los cambios realizados en el código a un entorno de producción después de la fase de compilación automática. El código estaría preparado para publicarse en producción en todo momento, y para ello se utilizan entornos de prueba, preproducción... similares a la de producción. Durante el proceso tras una nueva compilación, esta se despliega de forma automática en un entorno de pruebas automáticas para el control de calidad y

buscar errores e incoherencias. Cuando pasa todas las pruebas, la entrega continua requiere de intervención humana para aprobar el despliegue de este entorno de prueba a producción. Al mismo tiempo, esta publicación manual del producto software puede provocar inconvenientes que afectan tanto al equipo de desarrollo como al de operaciones, incluso al cliente, si no se gestiona correctamente ante posibles retrasos en cada entrega, siendo además un proceso propenso a errores debido a su carácter manual. Es tarea de cada organización establecer este canal de entrega continua según sus necesidades o situación específica, por ejemplo, tomar la decisión de publicación del producto a última hora. Para este caso, es necesario que ese producto se encuentre preparado y listo para su lanzamiento al final del sprint (Atlassian, 2022; IBM, 2023).

Despliegue Continuo (CD)

Es la evolución de la Entrega Continua (CDE) hacia la automatización eliminando la intervención manual. Dicho de otra manera, y como también se ha explicado en el apartado anterior, el proceso de CD comienza con CI y consiste en automatizar la compilación, pruebas, configuración y la implementación desde entornos de prueba hacia el de producción. Las nuevas compilaciones pasan por una “canalización de versión” donde avanzan por varios entornos de ensayo para crear la infraestructura necesaria en la implementación de nuevas compilaciones, además de actividades de integración, carga y pruebas de aceptación. Los ciclos de lanzamiento software con CDE manual pueden provocar cuellos de botella y versiones del producto poco fiables, con retrasos y errores. La práctica de CD proporciona una ruta más rápida del código o componente a la implementación en producción, minimizando además el tiempo para mitigar (TTM) o tiempo para corregir (TTR) incidentes en producción. Prácticas como la infraestructura como código (IaC) y supervisión o monitorización son complementarias al CD y facilitan la canalización dentro del pipeline CI/CD (mijacobs, 2023).

2.1.5. Herramientas CI/CD

Como ya se ha podido ver en el apartado anterior, la implantación de un entorno CI/CD es fundamental para desarrollar y entregar software de alta calidad. No obstante, existen una gran cantidad de herramientas y plataformas que ofrecen soluciones de integración y despliegue continuos, por lo que la tarea de escoger una frente a otra se complica.

Por ello, se han escogido tres herramientas CI/CD muy utilizadas actualmente en el mercado software para analizar sus puntos fuertes y compararlos entre sí a través de una tabla comparativa. De esta manera se obtendrá una visión global de cada una de sus características para escoger la más adecuada para el desarrollo práctico del proyecto.

Las herramientas escogidas para la realización de la tabla comparativa han sido las siguientes:

1. Jenkins (Jenkins, 2024)
2. Github Actions (GitHub, 2020a)
3. AWS CodePipeline (Amazon Web Services, 2014)

Tabla 1. Comparación herramientas CI/CD

Característica	Jenkins	GitHub Actions	AWS CodePipeline
Tipo	Herramienta de código abierto	Integrada en GitHub	Servicio nativo de AWS
Facilidad de Configuración	Requiere configuración manual en servidores	Fácil de configurar desde la interfaz de GitHub	Integración sencilla con servicios AWS
Integración con Repositorios	Soporta varios repositorios (GitHub, GitLab, etc.)	Funciona principalmente con repos de GitHub	Principalmente AWS CodeCommit, también GitHub
Escalabilidad	Escalable, pero depende del servidor de Jenkins	Escala automáticamente en GitHub Actions	Alta escalabilidad en AWS
Coste	Gratuito, pero con costos de infraestructura	Gratuito para repos públicos, pago por uso en repos privados	Pago por uso
Ecosistema de Plugins	Amplia gama de plugins para integración con múltiples herramientas	Marketplace con una gran variedad de acciones	Integrado con muchos servicios de AWS
Flexibilidad	Altamente flexible, se puede adaptar a casi cualquier pipeline	Moderada, muy orientada a GitHub	Alta en entornos AWS, limitada fuera de AWS
Soporte de Contenedores	Compatible con Docker y Kubernetes	Compatible con Docker y Kubernetes	Compatible con Docker y ECS/EKS
Visibilidad y Monitoreo	Plugins disponibles para monitoreo (ej. Prometheus, Grafana)	Panel de control dentro de GitHub	Integración nativa con CloudWatch
Automatización	Se puede automatizar completamente, pero requiere configuración manual	Altamente automatizado, fácil de configurar	Totalmente automatizado con servicios AWS
Soporte y Comunidad	Gran comunidad de código abierto	Gran comunidad de usuarios en GitHub	Soporte de AWS y comunidad más limitada

Fuente: Elaboración propia.

2.2. Trabajos relacionados

En este apartado se destacarán algunos trabajos relacionados con la temática del proyecto a cerca de las tecnologías de orquestación de contenedores como Kubernetes y los entornos de integración y despliegue continua (CI/CD). Son los siguientes:

Caso de éxito de Booking utilizando Kubernetes (Kubernetes, 2020a): en el año 2016 la empresa “Booking.com” (Booking, 2024) inició un proyecto para de crear y gestionar una plataforma propia de Kubernetes para el despliegue de su aplicación. Al principio, los equipos de desarrolladores no disponían apenas de formación sobre la utilización de Kubernetes, pero tras formarse y superar la curva de dificultad, los resultados obtenidos a cambio fueron superiores al esfuerzo de formación del personal. Como resultado, consiguieron escalar sus servicios contenedores en cuestión de minutos ofreciendo a sus clientes una mejor calidad de servicio.

Revisión de la literatura disponible a cerca de diversas herramientas de CI/CD (Shahin et al., 2017): los autores realizaron una enorme revisión sistemática del estado del arte de las prácticas relacionadas con la integración y despliegue continuos como CI (Integración continua), CD (Despliegue continuo) o CDE (Entrega continua), clasificar enfoques y herramientas e identificar los desafíos a los que se enfrentan estos tipos de tecnologías. Es un documento denso, pero muy completo, que aporta una visión muy extensa y analítica sobre el mundo del CI/CD.

2.3. Conclusiones del estado del arte

Los orquestadores de contenedores son la clave del éxito y la solución a la gestión eficaz de las aplicaciones desplegadas en forma de contenedores, sobre todo si estas aplicaciones son desplegadas a gran escala con cientos y cientos de contenedores con interdependencias complejas. De la misma forma, los entornos de CI/CD automatizados son fundamentales en toda empresa software que busque desarrollar y entregar código de calidad, que necesite iteraciones frecuentes del producto para su mejora continua y establezca un medio para la colaboración entre los equipos de desarrollo y operaciones.

3. Objetivos y metodología de trabajo

3.1. Objetivo general

El objetivo principal de este trabajo es la actualización de un proyecto software inicial basado en una aplicación web desplegada en AWS a través de microservicios contenedores gestionados por un servicio orquestador Elastic Container Service (ECS). La actualización consistirá en la aplicación de procesos DevOps mediante la implantación de un nuevo orquestador de contenedores más potente con Kubernetes a través del servicio de AWS de Elastic Kubernetes Service (EKS) y se implantará pipeline de CI/CD, antes inexistente, a través de GitHub Actions para gestionar el ciclo de vida de la aplicación y así proporcionar un proceso que garantice su mejora continua a lo largo del tiempo. Finalmente, se expondrán las mejoras o beneficios del nuevo sistema con respecto al anterior a través de diversos criterios de medida para determinar la mejora de calidad adquirida.

3.2. Objetivos específicos

El objetivo principal se puede dividir en los siguientes objetivos específicos:

- Estudiar las tecnologías de orquestación de contenedores compatibles con AWS disponibles y justificar su elección para implementarlo en el nuevo sistema.
- Implementar el nuevo orquestador de contenedores EKS dentro del nuevo sistema.
- Estudiar las herramientas y entornos de CI/CD disponibles.
- Instalar un pipeline CI/CD dentro del nuevo sistema para gestionar el ciclo de vida de la aplicación.
- Medir la mejora del nuevo sistema con respecto al anterior.

3.3. Metodología del trabajo

Para abordar este trabajo de tipo desarrollo práctico, se ha seguido una serie de pasos o fases estructuradas, cada una alineada con el cumplimiento de los objetivos específicos, para asegurar el cumplimiento del objetivo principal propuesto:

En primer lugar, ha sido necesario un análisis del estado actual del sistema tanto a nivel de diseño, funcionalidades y componentes para entender de forma clara cómo trabajar con la

aplicación e infraestructura inicial, es decir, establecer un punto de partida para comenzar con la planificación y la recogida de requisitos necesario para la actualización del sistema.

Seguidamente, se realizó una revisión teórica y técnica de la literatura acerca de tecnologías de orquestación de contenedores Implementación y herramientas CI/CD, con la premisa de que estas deben ser compatibles con el proveedor en la nube de AWS donde se despliega el sistema actual. Como alternativa al orquestador actual de ECS, se investigó acerca de EKS puesto que forma parte del ecosistema de AWS y la implantación de Kubernetes en el nuevo sistema es uno de los objetivos principales del proyecto. El estudio y comparación de ambos orquestadores ayudó a detectar las ventajas de implantar Kubernetes con respecto a ECS, entre ellas, su amplio uso en la industria software junto a una gestión más eficiente del propio clúster, además de ser una tecnología de interés personal con la que se desea trabajar. Por otra parte, como el sistema actual no posee ningún control del ciclo de vida de la aplicación, se ha escogido GitHub Actions entre otras opciones como Jenkins y AWS CodePipeline debido principalmente a que realiza el mismo trabajo que sus competidores, pero con una facilidad de uso mayor y sin necesidad de instalar ni configurar de servidores (en el caso de Jenkins).

A continuación, comenzó la fase práctica a través de la implantación del orquestador EKS y del pipeline CI/CD mediante GitHub Actions. Primero se implantó el nuevo clúster de EKS en la infraestructura y se refactorizaron las partes necesarias del proyecto con los cambios necesarios pertinentes. Una vez implantado el clúster, se migraron los microservicios desde ECS a EKS para desplegar la aplicación en forma objetos de Kubernetes como “Services” y “Deployments”. En segundo lugar, se crearon flujos de trabajo de GitHub Actions para automatizar la compilación, empaquetado y subida al repositorio de control de versiones del código de los servicios. De la misma manera, se automatizó tanto el despliegue y destrucción de la infraestructura de AWS como el despliegue de la propia aplicación con ficheros de Kubernetes.

Finalmente, se evaluaron los beneficios y mejoras obtenidas gracias a la implantación del nuevo sistema a través de los siguientes criterios o medidas:

- Tiempo requerido para desplegar la aplicación
- Tiempo de despliegue del sistema completo
- Automatización

- Reproducibilidad
- Escalabilidad
- Compatibilidad y portabilidad multicloud

Con la consecución de estos pasos o fases, esta metodología garantizará un enfoque sistemático estructurado que permitirá cumplir con los objetivos propuestos y validará la actualización del sistema actual hacia una nueva infraestructura con una orquestación de contenedores más eficiente y la automatización tanto de su ciclo de vida como de la aplicación desplegada.

4. Desarrollo específico de la contribución

Este capítulo comenzará listando los requisitos que deberá cumplir el sistema final y el conjunto de herramientas utilizadas para la consecución de los objetivos del proyecto.

Después, se presentarán las características principales del sistema software actual antes de ser actualizado. Se explicarán los distintos componentes que lo forman, las tecnologías que utiliza y su funcionamiento para comprender el contexto y punto de partida del nuevo sistema software.

Seguidamente, se documentará el proceso de desarrollo del nuevo sistema implementado, justificando las modificaciones necesarias que se han realizado con respecto al sistema actual para la instalación de los nuevos componentes que formarán parte de la aplicación: un clúster de Kubernetes EKS dentro del proveedor en la nube de AWS y un entorno de CI/CD a través de GitHub Actions para la gestión del ciclo de vida de la aplicación.

Finalmente se comprobará que todos los componentes del nuevo sistema se encuentren correctamente desplegados y que todos los pasos o tareas del pipeline CI/CD se hayan realizado con éxito, además de una evaluación final donde se expondrán las mejoras adquiridas con respecto al sistema inicial.

4.1. Planificación / Análisis / Requisitos

En primer lugar, antes de presentar tanto la arquitectura como los componentes que forman el sistema inicial, se establecerán los requisitos que deberá cumplir el sistema final de cara a la migración del clúster ECS actual hacia un orquestador de Kubernetes y a la instalación de un pipeline CI/CD junto con las herramientas utilizadas en el proceso.

4.1.1. Requisitos del sistema a implantar

1. El sistema final deberá la infraestructura utilizada por el sistema actual bajo el proveedor de nube de Amazon Web Services.
2. El nuevo orquestador y la herramienta para implantar el entorno CI/CD deben ser compatibles con AWS.
3. Las fases de compilación, empaquetado y subida de los cambios en el código de los microservicios contenedores de la aplicación deberán ser automatizadas.

4. Las fases de compilación, empaquetado y subida al repositorio de imágenes de los microservicios contenedores deberán ser realizadas de forma automática al producirse cambios en su código.
5. El despliegue y destrucción de la infraestructura del sistema deberá ser automatizado.
6. El despliegue de la aplicación a través de Kubernetes deberá ser automatizado.

4.1.2. Tecnologías y herramientas

A continuación, se presentarán las herramientas utilizadas durante el desarrollo acompañadas de una breve descripción junto a su importancia dentro del sistema final:

4.1.2.1. Amazon Web Services (AWS)

Amazon Web Services (Amazon Web Services, 2024a) se ha consolidado como el proveedor líder en el sector de la computación en la nube. AWS dispone de tres modelos de servicios compatibles entre ellos que proporcionan diferentes soluciones, según las necesidades de la empresa o del proyecto: Infraestructura como Servicio (IaaS), Plataforma como Servicio (PaaS) y Software como Servicio (SaaS). Presenta un amplio catálogo con más de 200 servicios de alta disponibilidad y administrados por AWS, listos para ser desplegados en cuestión de minutos facilitando a las empresas adaptar su infraestructura informática de manera ágil y permitiendo el acceso a recursos de cómputo, almacenamiento, red... de forma flexible pagando solo por aquellos que se consumen.

La infraestructura del nuevo sistema se mantendrá bajo este proveedor puesto que se trata de una actualización y no de una migración o integración multinube con otros proveedores. Se buscará el aprovechar al máximo la infraestructura inicial, más aún, el propio orquestador de Elastic Kubernetes Service (EKS) a implantar forma parte del ecosistema de AWS, por lo que se podrá integrar perfectamente con el resto de los servicios.

4.1.2.2. Terraform

Terraform (HashiCorp, 2014) es una herramienta de infraestructura como código creado por la empresa HashiCorp. Tiene un tipo de codificación declarativa, lo que permite describir la infraestructura que se quiere construir mediante un lenguaje de alto nivel. Después de definir la infraestructura a través de sus ficheros identificados por la extensión “.tf”, Terraform elabora un plan de acciones para orquestar cómo se va a crear la infraestructura definida y lo ejecuta. Actualmente es una de las herramientas de IaC más utilizadas en el mercado gracias

a su naturaleza agnóstica que le permite ser compatible con cualquier proveedor de servicios en la nube y a la capacidad de suministrar infraestructura inmutable capaz de responder ante los cambios de configuración en el entorno sustituyendo el componente por otro nuevo sin causar problemas de compatibilidad entre recursos (IBM, 2024).

Mediante Terraform se automatizará la configuración, despliegue y destrucción de la infraestructura en AWS y el clúster EKS. Además, gracias a su compatibilidad con el administrador de paquetes **Helm** (Helm, 2021) para Kubernetes, se han podido desplegar componentes muy importantes dentro del sistema como:

- **AWS Load Balancer Controller** (Kubernetes, 2018b): componente necesaria para la utilización de balanceadores de carga de AWS por parte del clúster de Kubernetes, permitiendo la redirección del tráfico desde Internet hacia los pods.
- **Ingress Nginx Controller** (Kubernetes, 2018a): encargado de gestionar el tráfico entrante desde los balanceadores de carga hacia los microservicios de la aplicación realizando la función de proxy inverso.

4.1.2.3. Docker y Docker Hub

Docker (Docker, 2022) es una plataforma de código abierto y el estándar de facto para la gestión de contenedores. Además, cuenta con un enorme repositorio de imágenes llamado Docker Hub (Docker Hub, 2014) donde los vendedores de software, proyecto open-source y la comunidad de desarrolladores ponen a disposición de los usuarios sus imágenes de contenedores junto a un control de versiones de cada uno de ellos.

Los microservicios que forman la aplicación, compuesta por un servicio Front-End y dos servicios APIREST, se encuentran compilados y empaquetados en forma de imágenes Docker y almacenados en un repositorio de Docker Hub. Más adelante, serán desplegados en forma de contenedores dentro de pods en el clúster de EKS.

4.1.2.4. Elastic Kubernetes Service y objetos de Kubernetes

El clúster EKS, tanto el plano de control y como los nodos trabajadores, serán desplegados mediante Terraform dentro de la infraestructura de AWS. Se establecerán todos los permisos necesarios a través de roles IAM de AWS para que los nodos puedan operar correctamente.

La aplicación será desplegada a través de objetos de Kubernetes (Kubernetes, 2023). Estos objetos son entidades persistentes dentro de Kubernetes que representan el estado deseado del clúster. Dentro de un objeto se pueden describir qué contenedores de aplicaciones correrán en el clúster, en qué nodos lo harán, qué recursos del sistema están disponibles para ser usadas por las aplicaciones, las variables que utilizarán los contenedores...

Entre ellos, la aplicación estará formada por los siguientes objetos:

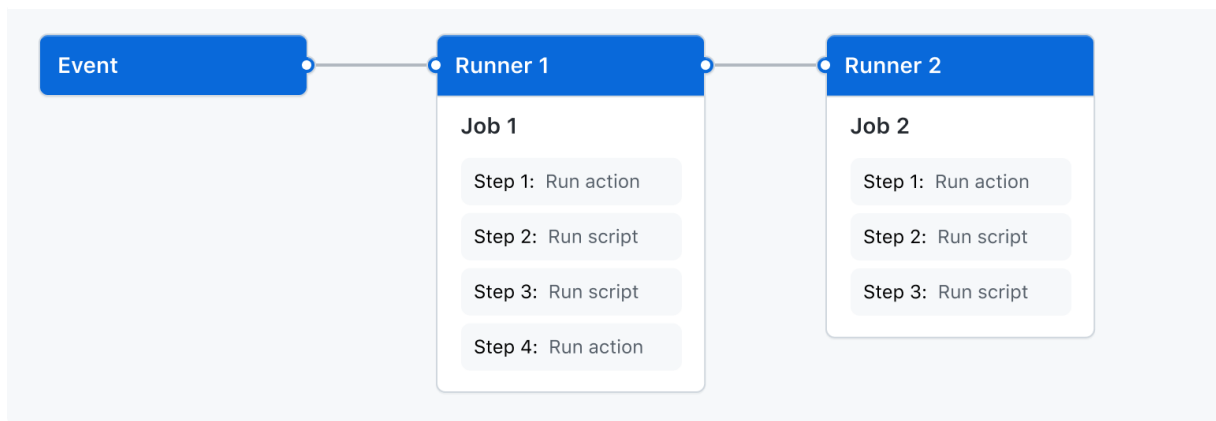
- **Namespace:** espacio lógico dentro del clúster donde se desplegarán el resto de los objetos de la aplicación.
- **Configmaps:** objeto utilizado para el almacenamiento de datos no confidenciales por clave-valor. Dentro de ellos se almacenarán las variables de entorno que utilizarán los pods.
- **Deployments:** dentro de ellos se describe el estado deseado de la carga de trabajo, donde podrán definir principalmente los pods, y contenedores dentro de estos, que se desplegarán dentro del clúster junto con el número de réplicas del pod deseadas.
- **Services:** es la forma que tiene Kubernetes de exponer los pods y establecer la manera de acceder a ellos, tanto para comunicarse entre sí como con otros componentes fuera del clúster. Hay varios tipos, los más utilizados son:
 - ClusterIP: expone el servicio internamente al clúster mediante una IP.
 - NodePort: servicio expuesto de forma externa a través de un puerto.
 - LoadBalancer: servicio expuesto externamente con función de balanceador de carga hacia otros servicios y pods.
- **Ingress:** gestiona el acceso desde el exterior hacia los servicios dentro del clúster a través del enrutamiento del tráfico, generalmente rutas HTTP y HTTPS, basado en forma de reglas.

4.1.2.5. GitHub y GitHub Actions

Todo el código del sistema se encontrará versionado dentro de un repositorio GitHub y automatizado su ciclo de vida a través de GitHub Actions (GitHub, 2020b). Esta plataforma de integración y despliegue continuos (CI/CD) permite crear flujos de trabajo para la realización de acciones como la compilación, pruebas y despliegue del código del repositorio.

Cada flujo de trabajo contiene uno o más trabajos que pueden ser ejecutados en orden secuencial, al definir varios trabajos dentro de un mismo flujo, o en paralelo (Figura 7). Cada trabajo se ejecutará en un ejecutor independiente, que son las máquinas virtuales de Linux, Windows y macOS proporcionadas por GitHub. Un flujo de trabajo es ejecutado por un evento, que es una acción específica originada dentro del repositorio como un push con los nuevos cambios sobre el código o cuando se abren una petición de pull request .

Figura 7. Estructura GitHub Actions



Fuente: (GitHub, 2020b).

Los trabajos están divididos en varios pasos donde pueden ejecutarse scripts de shell o acciones definidas en un archivo YAML, almacenado en el propio repositorio dentro de una carpeta dedicada llamada “.github/workflows”. Estas acciones son uno de los puntos fuertes de GitHub Actions, ya que son pequeñas aplicaciones personalizadas que realizan tareas complejas y repetitivas como hacer un checkout del repositorio de Git desde GitHub o configurar credenciales en un proveedor de servicios en la nube. Además, son altamente reutilizables al estar disponibles dentro del propio GitHub Marketplace donde los desarrolladores comparten sus propias acciones.

Dentro del repositorio del sistema final, se encontrarán diversos flujos de trabajo los cuales se encargarán de automatizar las fases del ciclo vida de los microservicios, infraestructura y aplicación definidas en los requisitos expuestos en este apartado.

4.2. Descripción del sistema desarrollado / Implementación

4.2.1. Presentación del sistema software actual

La funcionalidad del proyecto consiste en una aplicación web basada en microservicios destinada a la agricultura, donde los usuarios podrán controlar el riego sobre parcelas a través de sensores. El sistema mantiene un registro sobre la información referente a las parcelas como el propietario, la lista completa de sensores dentro de las parcelas, información geoespacial para localizarla en un mapa interactivo y la referencia catastral, que es un identificador oficial y obligatorio de los inmuebles. También almacena la información sobre los dispositivos sensores: el propietario, referencia catastral de su parcela, coordenadas de latitud y longitud para poder ser localizado y el tipo de información que recoge el sensor.

La aplicación estará conectada, mediante su servicio encargado de las parcelas, a la web de la “Sede electrónica del Catastro” del Gobierno de España (Ministerio de Hacienda y Función Pública, 2009). Dentro de esta web se encuentran registradas de forma oficial todas las propiedades, las cuales serán en este caso parcelas, a través de sus identificadores de referencia catastral. La aplicación verificará que las parcelas añadidas se encuentren registradas dentro del catastro.

4.2.2. Servicios que componen la aplicación

La aplicación del sistema está formada por microservicios que interactúan entre sí, implementados en contenedores Docker, almacenados en un repositorio de Docker Hub y desplegados dentro de instancias EC2 en una infraestructura de AWS cuya construcción está automatizada a través de Terraform.

Lista de servicios:

1. **Tfm-front-end**: servicio Front-End, puerto 4200.
2. **Plots**: servicio API REST, puerto 8080.
3. **Sensor**: servicio API REST, puerto 8081.

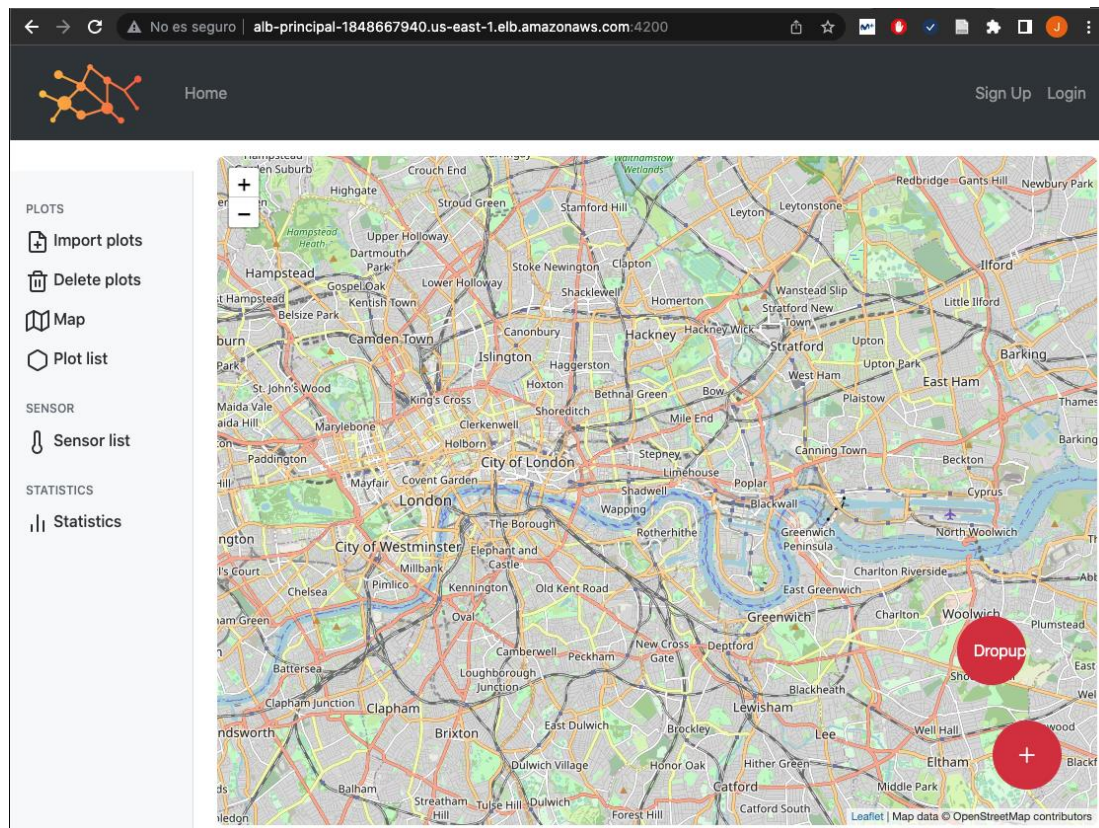
Lista de servicios de bases de datos:

1. **Postgres**, puerto 5432 y conectado al servicio **Sensor**.
2. **Mongo**, puerto 27017 y conectado al servicio **Plots**.

4.2.2.1. Servicio Front-End

El servicio Front-End permite navegar por la aplicación hacia el resto de los servicios (Figura 8). Está desarrollado con el framework Angular (Angular, 2015) para aplicaciones web. Su página principal mostrará un mapa interactivo para visualizar las parcelas dentro de la aplicación:

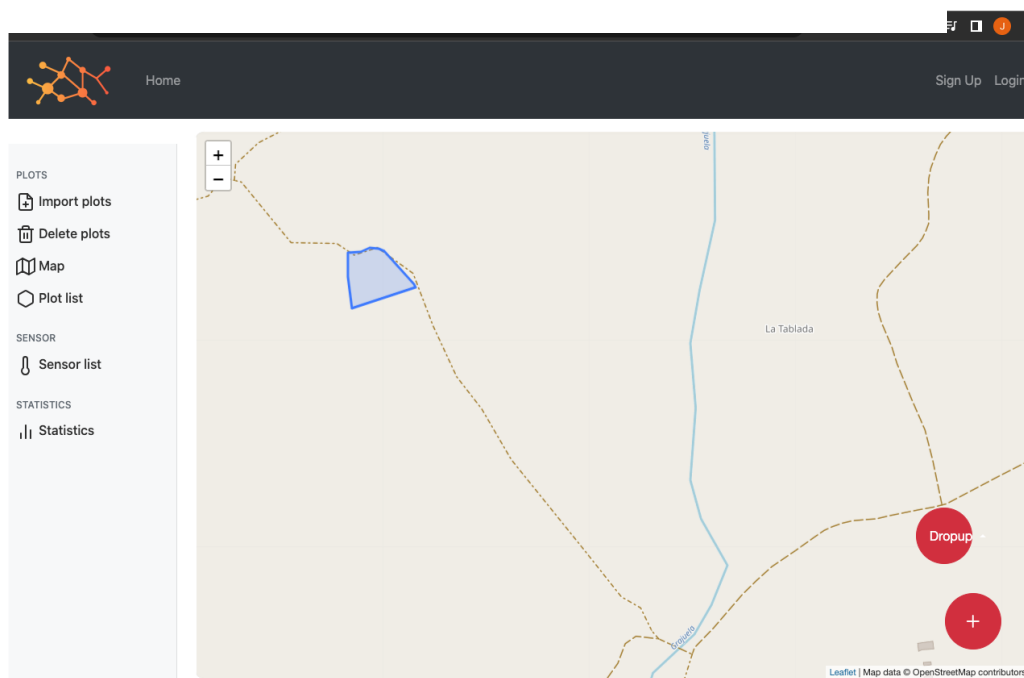
Figura 8. Navegador Front-End



Fuente:

Elaboración propia.

Las parcelas que se encuentran registradas en la aplicación aparecen coloreadas en el mapa interactivo gracias al objeto geométrico al que da forma sus coordenadas (Figura 9):

Figura 9. *Front parcelas*

Fuente: Elaboración propia.

Como puede verse en la imagen anterior, la parcela que se encuentra registrada en la aplicación está localizada en la parte oeste de la localidad de Arroyo de la Luz en la provincia de Cáceres. Como esta parcela existe y se encuentra registrada en el catastro (Figura 10), es posible consultar sus datos en el apartado de [buscador de propiedades de la propia página del Catastro](#):

Figura 10. *Parcela catastro*

DATOS DESCRIPTIVOS DEL INMUEBLE	
Referencia catastral	10022A010000380000WZ  
Localización	Polígono 10 Parcela 38 LA CHURRUA. ARROYO DE LA LUZ (CÁCERES)
Clase	Rústico
Uso principal	Agrario

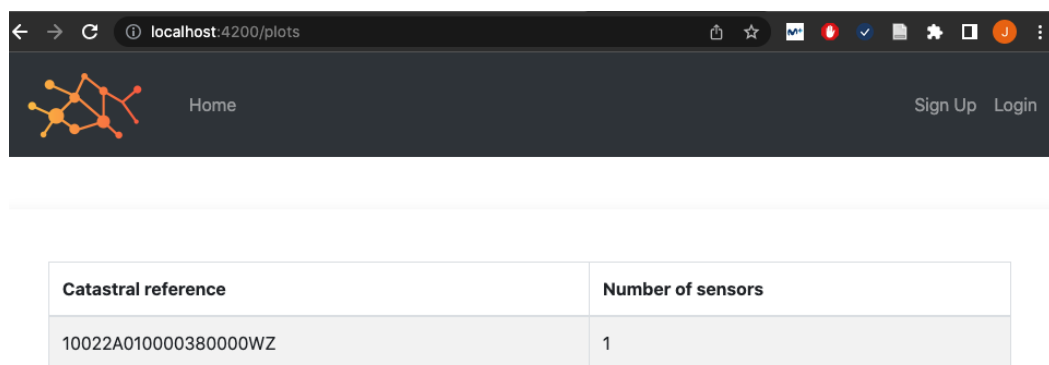
PARCELA CATASTRAL	
	Localización
	Polígono 10 Parcela 38 LA CHURRUA. ARROYO DE LA LUZ (CÁCERES)
	Superficie gráfica
	3.410 m ²

Fuente: Elaboración propia.

4.2.2.2. Servicios API REST de “plots” (parcelas) y sensores

Estos microservicios son aplicaciones con funcionalidad API REST, los cuales interactúan cada uno con su base de datos correspondiente que almacenará los datos que utilizará cada servicio. Están desarrollados con Java y mediante el framework de Spring Boot (Spring Boot, 2013) para el desarrollo de aplicaciones web y microservicios:

1. **Servicio de Parcelas:** funcionalidad de guardado y consulta de los datos referentes a las parcelas o “plots” conectado a una base de datos MongoDB. Este servicio interactúa con la web del Catastro para comprobar que la parcela existe y con el servicio de sensores para listar cuántos de ellos componen la parcela. Dentro de la aplicación web, se puede acceder a la lista de las parcelas registradas:

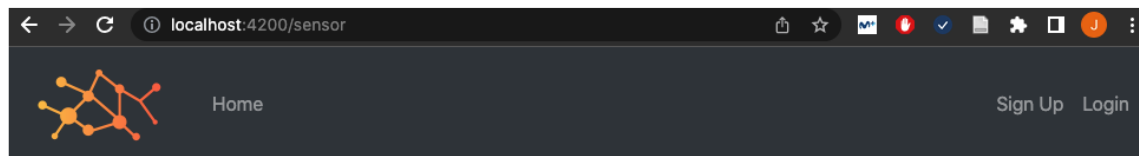


Catastral reference	Number of sensors
10022A010000380000WZ	1

La conexión con la base de datos de MongoDB es configurada a través del fichero “application.properties” propio de los proyectos de Spring Boot. Para el servicio de parcelas, obtendrá el host de la máquina donde se ha desplegado la base de datos a través de la variable de entorno `${MONGODB_HOST}`:

```
spring.data.mongodb.host=${MONGODB_HOST}
spring.data.mongodb.port=27017
spring.data.mongodb.username=admin
spring.data.mongodb.password=example
spring.data.mongodb.database=test
spring.data.mongodb.authentication-database=admin
server.port=8080
```

2. **Servicio de Sensores:** funcionalidad de guardado y consulta de los datos de sensores conectado a una base de datos PostgreSQL. En la aplicación se puede acceder a la lista de sensores registrados en la aplicación y que forman parte de una parcela (identificada por su referencia catastral):



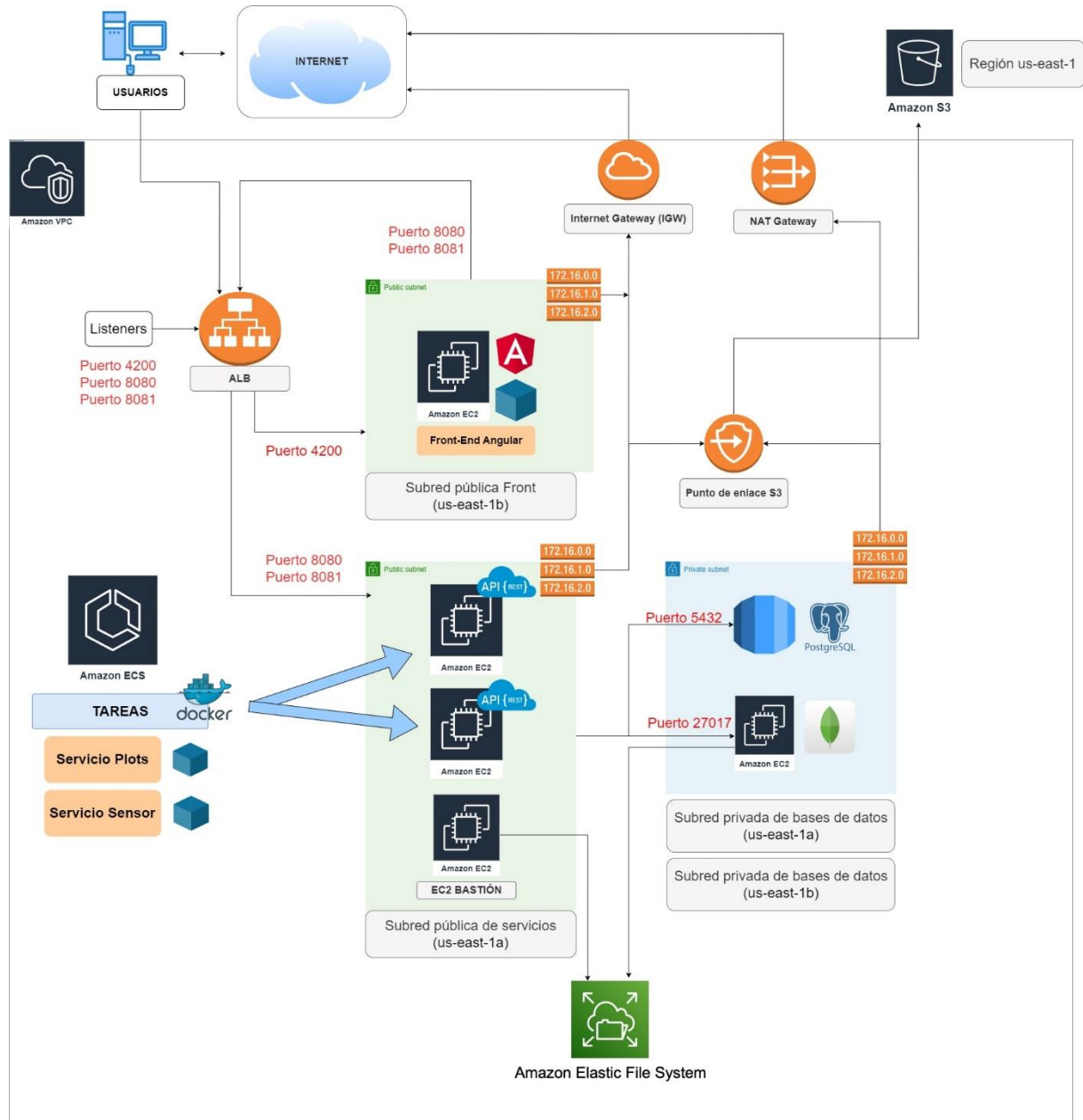
Id	Catastral reference	Latitude	Longitude	Type
b05c0778-6fca-11ec-90d6-0242ac120003	10022A010000380000WZ	12.12	14.5	temperature

Al igual que el servicio de sensores, la configuración de la conexión con la base de datos de PostgreSQL se realiza en el fichero “application.properties”. La variable de entorno `${POSTGRES_HOST}` contendrá el valor del host de la máquina donde se despliega la base de datos:

```
spring.jpa.database=POSTGRESQL
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://${POSTGRES_HOST}:5432/sensors
spring.datasource.username=postgres
spring.datasource.password=password
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
server.port=8081
```

4.2.3. Diagrama de arquitectura cloud en AWS

A continuación, se presentará la infraestructura inicial del sistema donde se despliega la aplicación:



4.2.3.1. Infraestructura de red

La red donde se despliegan los componentes de la infraestructura será una VPC. Esta es una red privada dentro de AWS en la que se podrán crear otras subredes que asignarán direcciones IP a los servicios dentro de esta. Las subredes tendrán sus correspondientes tablas de enrutamiento que les permitirá comunicarse con el resto de la red y con Internet a través de un Internet Gateway y un servicio de NAT Gateway.

La infraestructura de AWS está localizada en la región del Norte de Virginia “us-east-1” y las subredes utilizarán sus zonas de disponibilidad “us-east-1a” y “us-east-1b”.

Lista de subredes:

1. Subred pública de servicios (us-east-1a)

Es la subred principal donde se desplegarán las máquinas virtuales en instancias EC2 para lanzar los contenedores con los microservicios de la aplicación. Tendrá acceso a Internet a través del Internet Gateway. Adicionalmente, se desplegará una máquina EC2 denominada “EC2 Bastión”. La finalidad de esta instancia bastión será realizar tareas de configuración de otros servicios desplegados dentro de la infraestructura de una forma directa. Es una práctica que se suele realizar dentro de este tipo de infraestructuras en la nube para conectarse mediante SSH y un par de claves a las máquinas dentro de las subredes de la VPC, tanto públicas, pero sobre todo privadas a través del bastión situado dentro de la propia VPC, puesto que desde fuera no se podría acceder a las EC2 privadas de forma directa.

2. Subred pública con el servicio Front (us-east-1b)

Destinada a alojar una instancia EC2 con el servicio Front-End de la aplicación, la cual podrá redirigir el tráfico al resto de microservicios situados en la subred de servicios. También usará la Internet Gateway para acceder a la red.

3. Subred privada con las bases de datos (us-east-1a)

No será accesible desde fuera de la VPC puesto que alojará los servicios de bases de datos de la aplicación ejecutados en servicios RDS y dentro de instancias EC2 mediante contenedores.

4. Subred privada con las bases de datos (us-east-1b)

Se desplegará otra red con las mismas características que la anterior situada en la zona de disponibilidad “us-east-1a”, pero esta vez dentro la zona de disponibilidad “us-east-1b”. Estas subredes se asociarán para formar un grupo de subredes, que será necesario para poder crear la base de datos RDS (requisito de creación).

4.2.3.2. Clúster ECS y despliegue de microservicios

Los contenedores Docker con servicios Front-End y API REST de la aplicación se encontrarán almacenados en un repositorio de Docker Hub para ponerlos a disposición de la infraestructura. Se creará un clúster con dos instancias o nodos EC2 de tipo “t2.micro” (1 CPU

/ 1 GB de memoria) muy ligeras levantadas dentro de la subred de servicios y se utilizarán las herramientas que posee ECS, llamadas tareas, para lanzar los contenedores. Mediante estas tareas, se podrán importar las imágenes Docker que se subieron a Docker Hub con la implementación de los microservicios de la aplicación para su posterior lanzamiento a través de contenedores en estas instancias EC2. Se creará una tarea para cada servicio API REST de la aplicación: Plots y Sensor.

El Front-End de la aplicación se despliega en una instancia EC2 un poco más potente de tipo “t2.micro” (1 CPU / 2 GB de memoria) independiente al clúster ECS debido a un elevado consumo de memoria al ser una imagen Docker de gran tamaño.

4.2.3.3. Servicios de bases de datos

Los servicios RDS son los encargados de levantar las bases de datos relacionales, en este caso un PostgreSQL. Se utiliza un contenedor para desplegar el servicio de MongoDB dentro de una instancia EC2. Las RDS permiten por defecto la persistencia de datos a través de copias de seguridad o “snapshots” automáticas, pero la EC2 no cuentan con ello. Es por eso que se requiere de un servicio EFS externo que utilizará la instancia para persistir los datos, el cual será explicado a continuación.

4.2.3.4. Servicios de almacenamiento

Para el almacenamiento de la información dentro de la infraestructura se dispondrá de un bucket S3 con el que se almacenará todos los ficheros de configuración que utiliza la aplicación. Como el S3 no se encuentra en la VPC sino a nivel regional, las instancias EC2 accederán a este bucket de forma segura mediante un punto de enlace S3 que permitirá que sólo puedan acceder al contenido del bucket estas instancias dentro de la VPC.

Otro servicio que se utilizará es Amazon Elastic File System o EFS, capaz de definir sistemas de ficheros portátiles. Este EFS se encargará de la persistencia de datos de la instancia EC2 con MongoDB para que pueda guardar los datos fuera de la instancia en caso de que esta se destruya. Al desplegar el EC2 Bastión, este realizará el montaje del sistema de ficheros EFS para crear los directorios dentro de este que permitirán la persistencia de datos en las máquinas EC2 que ejecutarán los servicios de bases de datos de MongoDB

4.2.3.5. Conducción del tráfico de la aplicación

El usuario interactuará directamente con un servicio balanceador de carga de tipo ALB o “Application Load Balancer” que se encontrará escuchando las peticiones entrantes a través de “listeners” en distintos puertos. En función del puerto donde reciba la petición, la redirige a su servicio destino. El servicio Front-End también utilizará este ALB para interactuar con los servicios API REST.

Para acceder a la página principal el usuario se conectará a esta por el puerto 4200 utilizando el ALB, lo que le redigirá al servicio Front-End. Dentro del Front, para navegar por las diferentes páginas realizando las diferentes peticiones a las API REST de los servicios de parcelas y sensores utilizará el ALB, que estará escuchando las peticiones tanto en el puerto 4200 como los demás puertos donde están desplegados los microservicios:

- **Puerto 4200:** servicio Front-End
- **Puerto 8080:** servicio Plots.
- **Puerto 8081:** servicio Sensor.

Una vez presentada tanto la aplicación del sistema actual, su arquitectura, tecnologías y servicios que la forman, en los siguientes apartados se procederá con la explicación del sistema final.

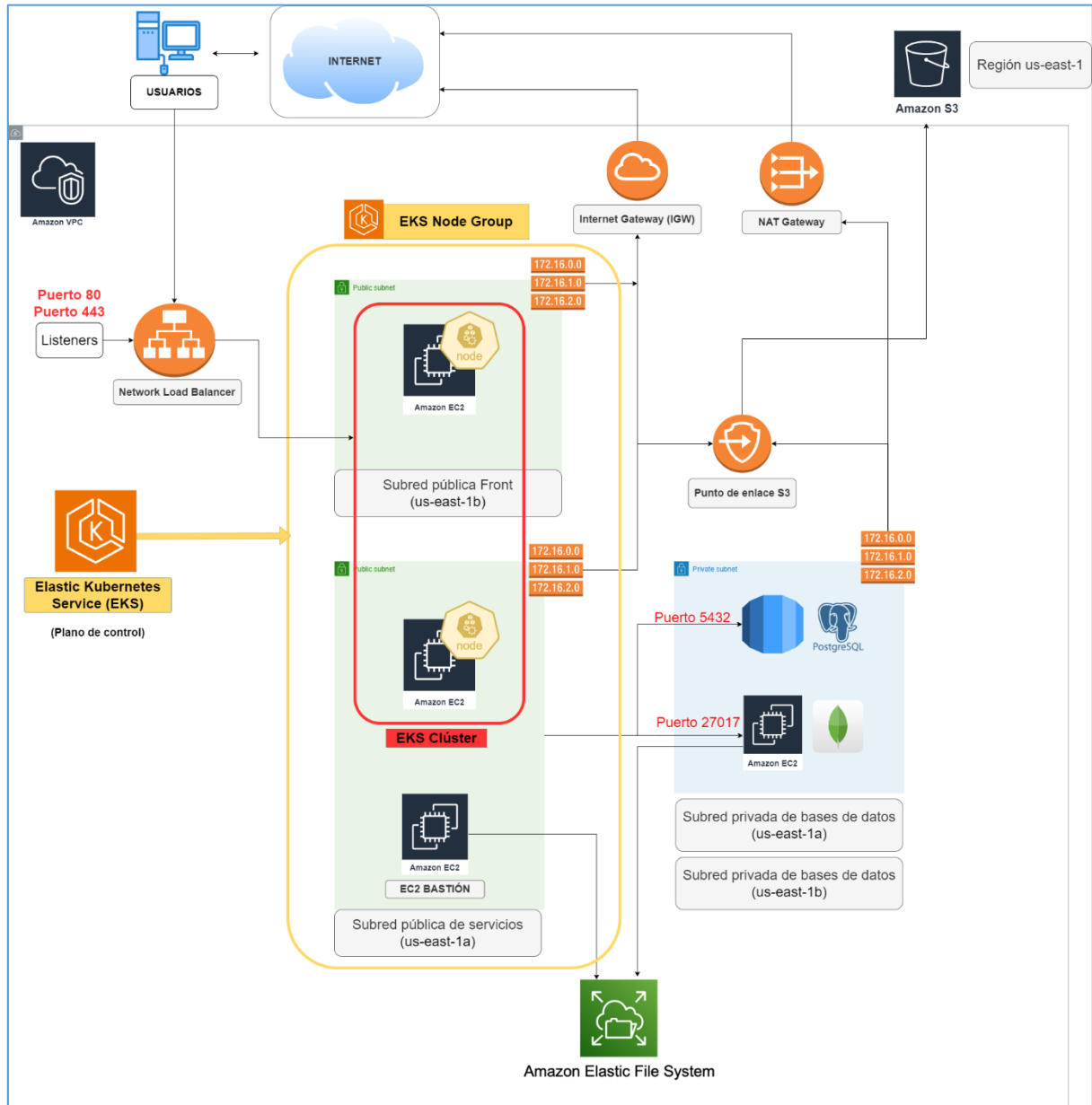
4.2.4. Presentación del sistema final actualizado

Para entender mejor los cambios y el funcionamiento del nuevo sistema tras implementar el orquestador EKS de Kubernetes junto con el pipeline de GitHub Actions, se presentarán diversos diagramas de las partes más importantes del sistema actualizado como lo son: la arquitectura final, funcionamiento del entorno CI/CD y el flujo de tráfico de la aplicación a través del clúster de Kubernetes.

Todos los detalles del proceso de desarrollo, configuraciones y ficheros utilizados se explicarán más adelante en el siguiente apartado del documento.

4.2.4.1. Arquitectura cloud del nuevo sistema en AWS

Mediante el siguiente diagrama se explicarán y visualizarán los cambios realizados directamente en la infraestructura de AWS:



Por una parte, el antiguo orquestador de ECS se ha sustituido por EKS. Tal y como se puede apreciar en la imagen anterior, el nuevo clúster desplegará el plano de control completamente administrado por AWS y establecerá un grupo de nodos trabajadores para ejecutar las cargas de trabajo. El clúster EKS operará en cualquiera de las dos subredes públicas en las zonas de disponibilidad “us-east-1a” y “us-east-1b” para garantizar la alta disponibilidad de servicio.

En el sistema anterior, el servicio Front se desplegaba en una EC2 independiente del clúster ECS donde se encontraban los servicios Plots y Sensor. Esto se debe en parte al propio diseño del sistema ya que los tipos de instancia máquinas EC2 del clúster no disponían de muchos recursos de computación, por lo que, en el caso de que ECS escogiera el mismo nodo para ejecutar el servicio Front con el resto de los servicios, provocaba problemas de memoria debido al tamaño de la imagen del Front y los recursos que consumen su contenedor resultante.

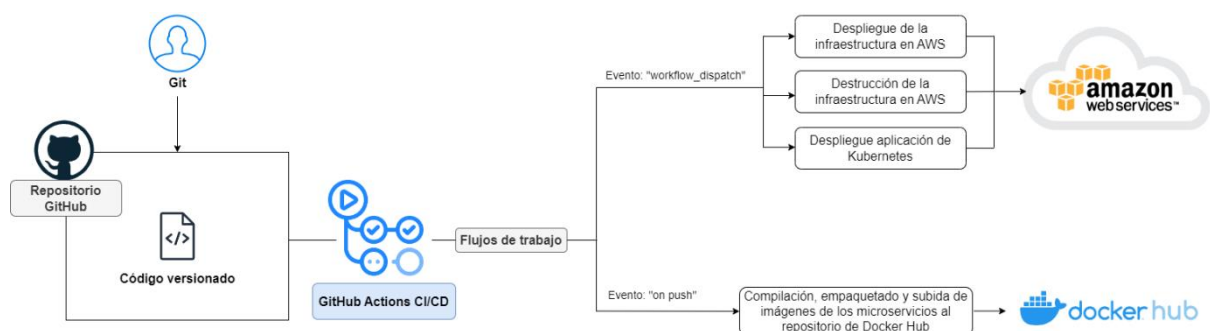
Ahora, todos los microservicios de la aplicación se desplegarán en forma de pods dentro de los nodos del grupo de nodos EKS, a los cuales se les ha asignado un tipo de instancia EC2 más potente para no tener problemas de recursos en el futuro. Estos cambios han beneficiado en gran manera a la aplicación puesto que:

1. Elimina la necesidad de desplegar y configurar una máquina EC2 extra dedicada al servicio Front, reduciendo el tiempo de despliegue de la infraestructura.
2. El servicio Front se despliega mucho más rápido y puede ser escalado por Kubernetes de una forma eficiente y en cualquiera de los nodos del clúster.

Por otra parte, el balanceador de carga de tipo ALB ha sido sustituido por un NLB o “Network Load Balancer” desplegado como servicio de Kubernetes. Este NLB ahora escuchará solo las peticiones HTTP en el puerto 80 y trabajará en conjunto con un pod NGINX que se desplegará en el clúster para redirigir el tráfico de la aplicación desde fuera del clúster hacia los pods expuestos a través de servicios dentro de Kubernetes.

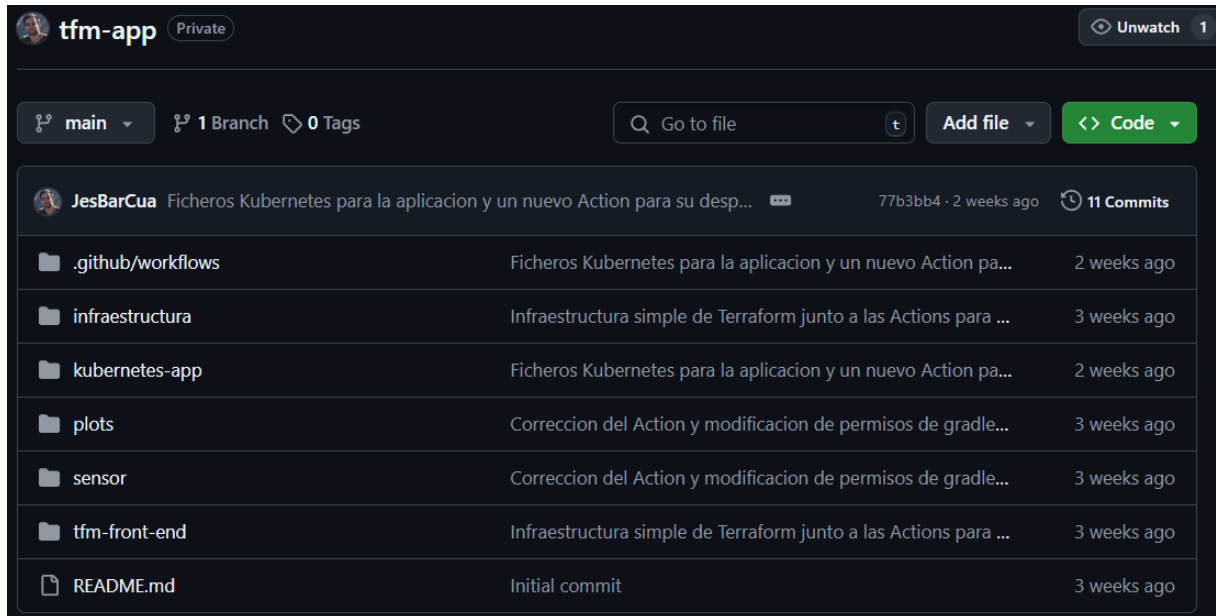
4.2.4.2. Diagrama del entorno CI/CD en el nuevo sistema

Se ha implantado el siguiente pipeline mediante GitHub Actions dentro del nuevo sistema:

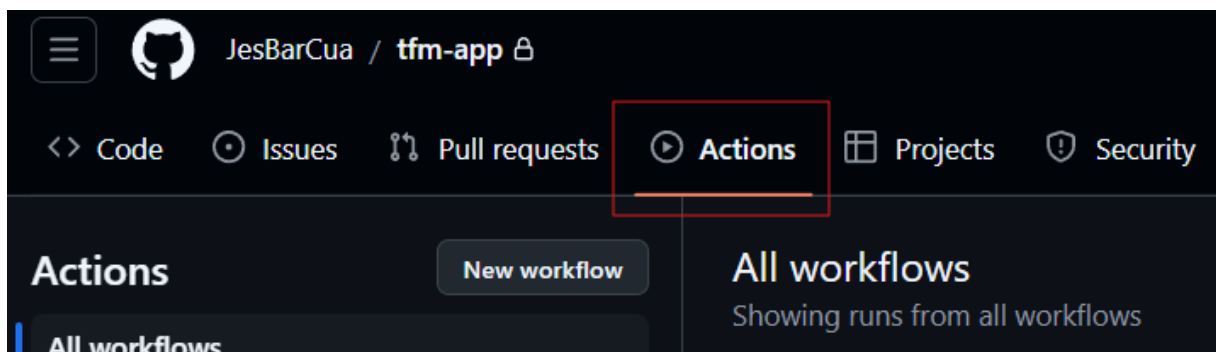


El código se encuentra versionado en el siguiente repositorio de GitHub dentro de la rama “main”

- **LINK DEL RESPOSITORIO PÚBLICO:** <https://github.com/jbarcua/tfm-app>

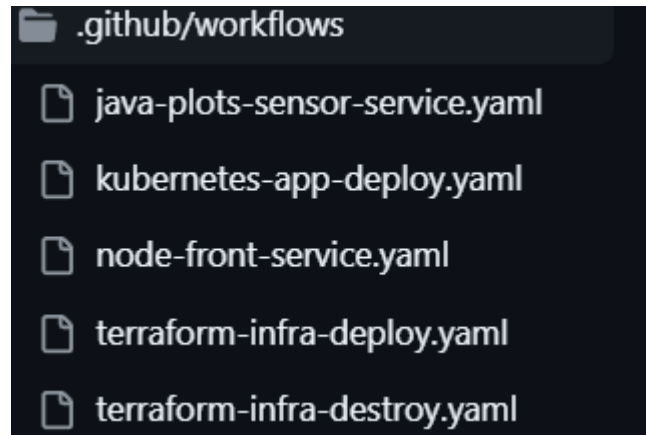


La herramienta del GitHub Actions está integrada dentro del repositorio, por lo que no es necesario ni instalar ni aprovisionar ningún tipo de servidor, sino que el propio GitHub ejecuta los trabajos en sus propios servidores llamados ejecutores o “runners”. Para acceder a la lista de flujos de trabajo y los resultados de sus ejecuciones solo hay que dirigirse al apartado “Actions”:



El repositorio se estructura en forma de carpetas ordenadas de la siguiente manera:

- **.github/workflows:** dentro de esta carpeta se encuentran los flujos de trabajo automatizarán el ciclo de vida del sistema. En su interior se encuentran los siguientes ficheros:



Como se puede apreciar en el diagrama con el entorno CI/CD completo al principio del apartado, los flujos de trabajo son ejecutados en función de eventos. Estos se especifican dentro del propio fichero de flujo. Se encuentran definidos dos tipos de evento:

1. **Evento “on push”:** disparado al iniciar una operación “push” a la rama “main” del repositorio. Es utilizado para que, al realizar cambios en el código de los microservicios Front, Plots o Sensor se ejecuten de forma automática los trabajos necesarios para reducir el tiempo en que estos servicios se encuentren disponibles dentro del repositorio de Docker Hub para su uso inmediato. Estos son los flujos de trabajo que se ejecutarán:
 - a. **java-plots-sensor-service.yaml:** compilado, empaquetado y subida a Docker Hub de los servicios Plots y Sensor.
 - b. **node-front-service.yaml:** compilado, empaquetado y subida a Docker Hub del servicio Front.
2. **Evento “workflow_dispatch”:** este tipo de evento es para la ejecución manual del flujo de trabajo. Se utilizan cuando se desea tener algo más de control sobre las ejecuciones de los trabajos. En este caso, es utilizado para ejecutar los trabajos

encargados de automatizar los pasos necesarios para el despliegue tanto de la infraestructura como de la aplicación de Kubernetes. Los flujos de trabajos que se podrán ejecutar son los siguientes:

- a. **terraform-infra-deploy.yaml**: despliegue de la infraestructura en AWS.
 - b. **terraform-infra-destroy.yaml**: destrucción de la infraestructura de AWS.
 - c. **kubernetes-app-deploy.yaml**: despliegue de la aplicación de Kubernetes.
- **Infraestructura**: se encuentran los ficheros de Terraform necesarios para el despliegue de la infraestructura de AWS.
 - **Kubernetes-app**: están todos los ficheros de Kubernetes necesarios para desplegar la aplicación dentro del clúster.
 - **Plots**: código del microservicio de parcelas o “Plots”.
 - **Sensor**: código del microservicio de sensores o “Sensor”.
 - **Tfm-front-end**: código del microservicio Front-End de la aplicación.

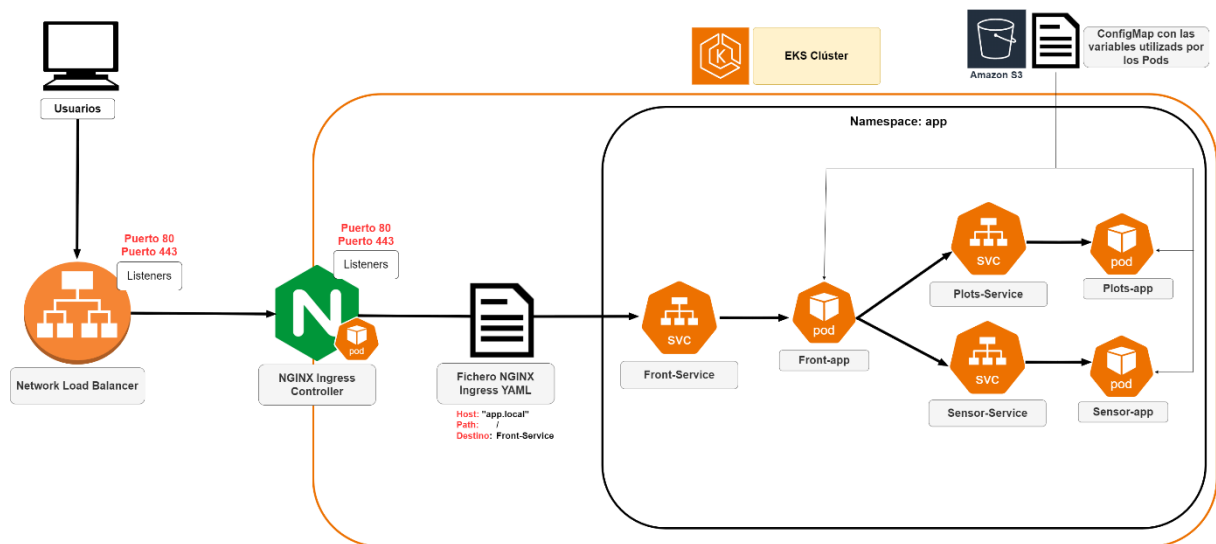
Cada flujo de trabajo actuará sobre una carpeta determinada en función de las tareas que se quieran realizar, así se tiene todo el código de la aplicación dentro de un mismo repositorio sin la necesidad de gestionar múltiples repositorios al mismo tiempo.

4.2.4.3. Flujo del tráfico de la aplicación a través del clúster de Kubernetes.

Entre las modificaciones realizadas en la infraestructura, la forma en la que se redirige el tráfico también ha recibido una serie de cambios necesarios para poder comunicar los microservicios de la aplicación dentro del clúster EKS con el exterior.

En el sistema anterior, el tráfico de la aplicación era controlado por un ALB, escuchando los puertos y redirigiendo las peticiones directamente a las instancias EC2 destino donde estaban desplegados los microservicios contenedores Docker. Con la implementación de un clúster de Kubernetes estos contenedores, ahora contenidos en pods, no pueden ser accedidos desde el exterior tan fácilmente. Estos necesitan de objetos de Kubernetes de tipo servicio para poder exponer los pods al resto del clúster internamente y al exterior. Uno de los servicios más comunes para exponer los pods son los servicios de tipo balanceador de carga, aunque existen también diversos mecanismos para realizar esta tarea.

A continuación, con la ayuda del siguiente diagrama se explicará cómo se controla es el flujo de tráfico de la aplicación desplegada en Kubernetes:



Primero, todas las peticiones entran desde Internet directamente al clúster EKS a través de un NLB o “Network Load Balancer”. Este balanceador de carga se encuentra desplegado tanto en la infraestructura de AWS como dentro del clúster en forma de servicio de tipo “LoadBalancer”. Está expuesto a Internet y será la puerta de enlace al clúster, siendo además un balanceador de red (capa 4) configurado para enviar las peticiones directamente a las IPs de los pods destino. En este caso, redirige hacia el pod del NGINX Ingress Controller.

Después, las peticiones son enviadas hacia el NGINX Ingress Controller para redirigir el tráfico hacia los pods dentro del clúster a través de una serie de reglas. Para ello utiliza un objeto de tipo Ingress donde se definen estas reglas, donde se establece el host origen esperado que cumpla la regla y una lista de rutas o “paths” a partir del host hacia las que dirigir la petición. La aplicación envía el tráfico hacia el servicio Front que expone el pod donde se encuentra desplegado el contenedor (igual para todos los microservicios).

Por último, el servicio Front redirige finalmente las peticiones hacia los servicios Sensor o Front utilizando las direcciones DNS internas “plots-service” y “sensor-service” del clúster. Estas direcciones estarán definidas en un objeto ConfigMap creado y subido a un bucket S3 durante el despliegue de la infraestructura con Terraform, en forma de variables de entorno que utilizarán los pods.

4.2.5. Proceso de desarrollo del sistema final

A continuación, se explicarán los detalles del proceso de desarrollo y el proceso seguido para la creación del sistema final:

4.2.5.1. Construcción del clúster EKS

El despliegue tanto del plano de control como el grupo de nodos trabajadores del clúster EKS ha sido realizado mediante Terraform al igual que el resto de la infraestructura.

1. Elastic Kubernetes Service

El clúster EKS se ha desplegado bajo la versión 1.29 de Kubernetes y sobre las subredes públicas principales de la infraestructura (zonas de disponibilidad “us-east-1a” y “us-east-1b”) donde también serán desplegados el grupo de nodos. Es importante activar la opción “bootstrap_cluster_creator_admin_permissions” para que el usuario con el que se está creando el clúster a través de Terraform tenga permisos de administrador.

```
resource "aws_eks_cluster" "eks-tf" {
  name      = "app-cluster-eks"
  version   = "1.29"
  role_arn  = aws_iam_role.eks-role-tf.arn

  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true

    subnet_ids = [
      aws_subnet.subred-servicios-1a-tf.id,
      aws_subnet.subred-front-1b-tf.id
    ]
  }

  access_config {
    authentication_mode                = "API"
    bootstrap_cluster_creator_admin_permissions = true
  }

  depends_on = [aws_iam_role_policy_attachment.asignacion-politica-AmazonEKSClusterPolicy]
}
```

Es necesario establecer y asignar los permisos pertinentes al clúster en AWS para su correcto funcionamiento. El siguiente rol de IAM creado solo afectará al servicio de EKS (“eks.amazonaws.com”) y se le asignará la política principal “AmazonEKSClusterPolicy” con los permisos necesarios para operar, junto con la política extra “AmazonS3FullAccess” para otro tipo de tareas relacionadas con el acceso al servicio S3.

```
#####
# Rol con politica de permisos para el funcionamiento del cluster EKS
#####

resource "aws_iam_role" "eks-role-tf" {
  name = "eks-role-tf"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        "Effect" : "Allow",
        "Action" : "sts:AssumeRole",
        "Principal" : {
          "Service" : "eks.amazonaws.com"
        }
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "asignacion-politica-AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.eks-role-tf.name
}

resource "aws_iam_role_policy_attachment" "asignacion-politica-AmazonS3FullAccess-eks" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonS3FullAccess"
  role       = aws_iam_role.eks-role-tf.name
}
```

Con esta configuración ya se puede desplegar el plano de control del clúster por lo que se procede con la creación del grupo de nodos:

2. Grupo de nodos del clúster

Los nodos han sido desplegados con la misma versión 1.29 de Kubernetes que los nodos máster del plano de control y en las mismas subredes. Dentro de este grupo se crearán como máximo tres nodos y se espera que siempre se encuentre un nodo operativo. El tipo de instancia EC2 de los nodos será una “t3.medium” (2 CPU / 4 GB de memoria) con recursos suficientes para soportar la carga de trabajo de la aplicación sin problemas.

```

resource "aws_eks_node_group" "eks-nodes-tf" {
  cluster_name = aws_eks_cluster.eks-tf.name
  version      = "1.29"
  node_group_name = "eks-cluster-nodegroup"
  node_role_arn = aws_iam_role.eks-role-nodes-tf.arn

  subnet_ids = [
    aws_subnet.subred-servicios-1a-tf.id,
    aws_subnet.subred-front-1b-tf.id
  ]

  capacity_type = "ON_DEMAND"
  instance_types = ["t3.medium"]

  scaling_config {
    desired_size = 1
    max_size     = 3
    min_size     = 1
  }

  update_config {
    max_unavailable = 1
  }

  labels = {
    role = "general"
  }

  depends_on = [
    aws_iam_role_policy_attachment.asignacion-politica-AmazonEKSWorkerNodePolicy,
    aws_iam_role_policy_attachment.asignacion-politica-AmazonEKS_CNI_Policy,
    aws_iam_role_policy_attachment.asignacion-politica-AmazonEC2ContainerRegistryReadOnly,
  ]

  # Allow external changes without Terraform plan difference
  lifecycle {
    ignore_changes = [scaling_config[0].desired_size]
  }
}

```

Los nodos necesitan también los permisos necesarios para operar correctamente. Se le asignará el siguiente rol de IAM que afectará a las instancias EC2 de los nodos (ec2.amazonaws.com) con las siguientes políticas:

- “AmazonEKSWorkerNodePolicy”: permisos para las funcionalidades principales de los nodos trabajadores.
- “AmazonEKS_CNI_Policy”: permisos para realizar tareas relacionadas con conectividad y red de los nodos.

- “AmazonEC2ContainerRegistryReadOnly”: acceso al registro de contenedores ECR de AWS en el caso de que se almacenen contenedores dentro de este.

```
#####
# Rol con politica de permisos para los nodos worker del cluster EKS
#####

resource "aws_iam_role" "eks-role-nodes-tf" {
  name = "eks-role-nodes-tf"

  assume_role_policy = jsonencode({
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
    }]
    Version = "2012-10-17"
  })
}

resource "aws_iam_role_policy_attachment" "asignacion-politica-AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.eks-role-nodes-tf.name
}

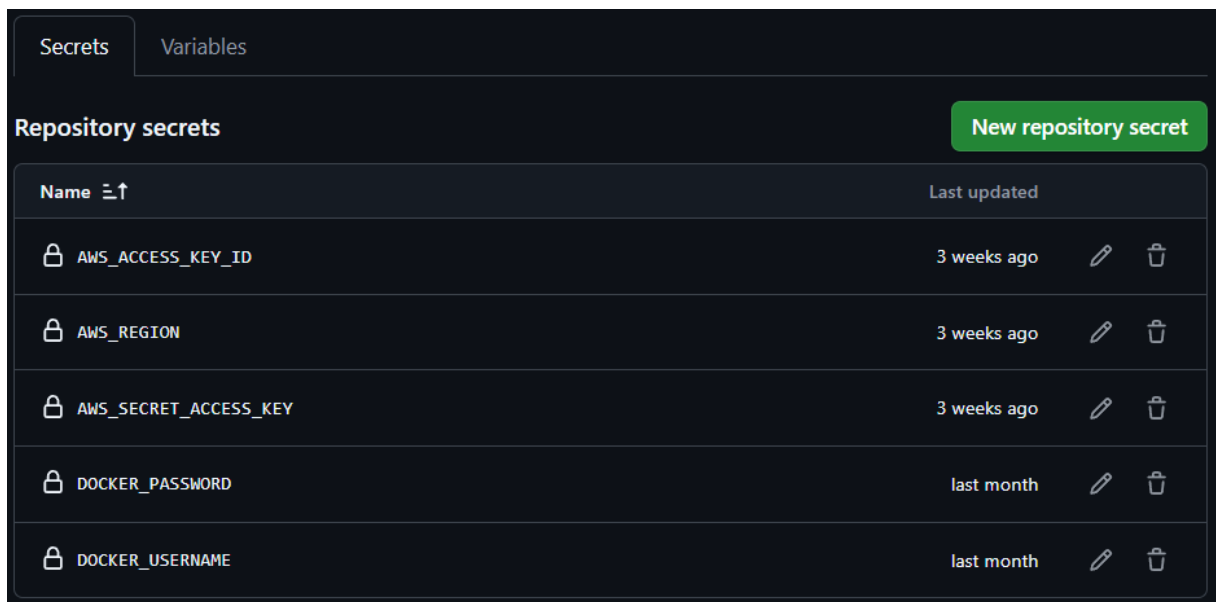
resource "aws_iam_role_policy_attachment" "asignacion-politica-AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.eks-role-nodes-tf.name
}

resource "aws_iam_role_policy_attachment" "asignacion-politica-AmazonEC2ContainerRegistryReadOnly" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.eks-role-nodes-tf.name
}
```

4.2.5.2. Definición de los flujos de trabajo con GitHub Actions

Dentro de la carpeta “.github/workflows” del repositorio de GitHub se han desarrollado los siguientes flujos de trabajo, los cuales se ejecutarán en la plataforma de CI/CD de GitHub Actions integrada de forma nativa.

Antes de proceder con la explicación, dentro del repositorio han de guardarse las variables con las claves, contraseñas... y demás información que utilizarán los flujos de trabajo. En este caso de guardarán en forma de secretos ya que es información sensible de acceso a la cuenta de Docker y AWS.



Una vez establecidos los secretos necesarios, estos ya pueden ser utilizados en cualquier momento por los flujos de trabajo como si fueran variables de entorno mediante el formato `${{ secrets.NOMBRE_VARIABLE }}`.

java-plots-sensor-service.yaml: compilado, empaquetado y subida a Docker Hub de los servicios Plots y Sensor.

El siguiente flujo de trabajo se activará de forma automática al realizar una operación “push” de Git sobre el repositorio y solo afectará a las carpetas con los microservicios Plots y Sensor si se han producido cambios en estos proyectos. De esta manera, si se producen cambios en el código de otros servicios no activarán este flujo.

```
name: Compilar, empaquetar con Docker y subir al repositorio los servicios Plots y Sensor.

on:
  push:
    branches:
      - main
    paths:
      - 'plots/**'
      - 'sensor/**'
```

En el flujo se encuentran definidos dos trabajos que se ejecutarán secuencialmente. El primer trabajo se lanzará para el servicio Plots y el segundo para Sensor:

1. Job “build-plots”

Dentro de este trabajo se ejecutan en orden una serie de pasos para el servicio Plots: hacer un checkout del repositorio, instalar JDK 11 para la compilación del proyecto, compilar el proyecto, hacer log in en la cuenta de Docker Hub, empaquetar la imagen con Docker y subirlo al repositorio. Mediante la opción “working-directory”, estos comandos son ejecutados dentro del directorio donde se encuentran los ficheros del servicio.

```
jobs:
  build-plots:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout del código
        uses: actions/checkout@v2

      - name: Instalar JDK 11
        uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '11'

      - name: Build Plots con Gradle Wrapper integrado en el proyecto
        working-directory: ./plots
        run: ./gradlew bootJar

      - name: Logearse en Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${ secrets.DOCKER_USERNAME }
          password: ${ secrets.DOCKER_PASSWORD }

      - name: Build Docker image servicio Plots
        working-directory: ./plots
        run: docker build -t ${ secrets.DOCKER_USERNAME }/plots:kubernetes .

      - name: Push Docker image servicio Plots
        run: docker push ${ secrets.DOCKER_USERNAME }/plots:kubernetes
```

En algunos pasos se utilizan comandos corrientes como “Docker build...” y en otros se utilizan acciones ya creadas por la comunidad de Docker Hub para la realización de tareas repetitivas con más pasos, como por ejemplo “docker/login-action@v3”, donde solo es necesario proporcionar las credenciales de Docker para realizar el log in.

2. Job “build-sensor”

Este trabajo es idéntico al anterior, pero se aplica al servicio Sensor

```
build-sensor:
  runs-on: ubuntu-latest

  steps:
    - name: Checkout del código
      uses: actions/checkout@v2

    - name: Instalar JDK 11
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: '11'

    - name: Build Sensor con Gradle Wrapper integrado en el proyecto
      working-directory: ./sensor
      run: ./gradlew bootJar

    - name: Logearse en Docker Hub
      uses: docker/login-action@v3
      with:
        username: ${ secrets.DOCKER_USERNAME }
        password: ${ secrets.DOCKER_PASSWORD }

    - name: Build Docker image for Sensor
      working-directory: ./sensor
      run: docker build -t ${ secrets.DOCKER_USERNAME }/sensor:kubernetes .

    - name: Push Docker image for Sensor
      run: docker push ${ secrets.DOCKER_USERNAME }/sensor:kubernetes
```

node-front-service.yaml: compilado, empaquetado y subida a Docker Hub del servicio Front.

El siguiente flujo de trabajo se activará al realizar una operación “push” para subir los cambios realizados en el código sobre la carpeta con el servicio Front. Sigue el mismo orden de pasos que el flujo de trabajo de los microservicios Plots y Sensor, pero con el servicio Front.

```

name: Compilar, empaquetar con Docker y subir al repositorio el servicio Front.

on:
  push:
    branches:
      - main
    paths:
      - 'tfm-front-end/**'

jobs:
  build-front:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout del código
        uses: actions/checkout@v2

      - name: Logearse en Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${ secrets.DOCKER_USERNAME }
          password: ${ secrets.DOCKER_PASSWORD }

      - name: Build Docker image servicio Front
        working-directory: ./tfm-front-end
        run: docker build -t ${ secrets.DOCKER_USERNAME }/tfm-front-end:kubernetes .

      - name: Push Docker image servicio Front
        run: docker push ${ secrets.DOCKER_USERNAME }/tfm-front-end:kubernetes

```

terraform-infra-deploy.yaml: despliegue de la infraestructura en AWS.

Este flujo de trabajo de tipo “workflow_dispatch” se activará de forma manual en la interfaz de GitHub Actions para tener un mejor control sobre el despliegue de la infraestructura.

```

name: Crear S3 Bucket para guardar el estado y desplegar la infraestructura de AWS

on:
  workflow_dispatch:

```

Realiza un solo trabajo “setup_and_deploy”, pero el orden de sus pasos puede estructurarse en dos tareas diferenciadas: creación de un bucket S3 dentro de AWS para guardar el estado de Terraform y despliegue de la infraestructura.

En primer lugar, tras hacer un checkout del código del repositorio y configurar las credenciales de acceso a AWS, procede a crear un bucket S3 donde Terraform almacenará el fichero

“tfstate” para guardar el estado de la infraestructura. Antes, realiza una comprobación para prevenir la creación del bucket si este ya existía previamente.

```
jobs:
  setup_and_deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout del código
        uses: actions/checkout@v2

      - name: Configurar las credenciales de AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ${ secrets.AWS_REGION }

      - name: Comprobar si existe el S3 Bucket
        id: check_bucket
        run: |
          if aws s3api head-bucket --bucket app-bucket-tfstate 2>/dev/null; then
            echo "El bucket ya existe."
            echo "bucket_exists=true" >> $GITHUB_ENV
          else
            echo "El bucket no existe."
            echo "bucket_exists=false" >> $GITHUB_ENV
          fi

      - name: Crear el S3 Bucket para guardar el estado
        if: env.bucket_exists == 'false'
        run: |
          aws s3api create-bucket --bucket app-bucket-tfstate --region ${ secrets.AWS_REGION }
```

La segunda parte del trabajo instala Terraform y realiza el despliegue de la infraestructura en AWS con los comandos terraform init, validate y apply sobre los ficheros de Terraform del directorio de la infraestructura.

```
- name: Setup Terraform
  uses: hashicorp/setup-terraform@v2
  with:
    terraform_version: 1.7.1

- name: terraform init
  working-directory: ./infraestructura
  run: terraform init

- name: terraform validate
  working-directory: ./infraestructura
  run: terraform validate

- name: terraform apply
  working-directory: ./infraestructura
  run: terraform apply -auto-approve
```

terraform-infra-destroy.yaml: destrucción de la infraestructura de AWS.

Este flujo también se ejecutará de forma manual para controlar el momento en el que se desee destruir la infraestructura.

El trabajo “destroy” hará un checkout del código, instalará Terraform y configurará las credenciales de AWS. Después, tras hacer un “Terraform int” recupera el estado de la infraestructura de Terraform guardado dentro del bucket creado previamente y procede con la destrucción de la infraestructura (necesita el estado de Terraform previamente).

```
name: Destruir la infraestructura de AWS

on:
  workflow_dispatch:

jobs:
  destroy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout del código
        uses: actions/checkout@v2

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.7.1

      - name: Configurar las credenciales de AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ${ secrets.AWS_REGION }

      - name: terraform init
        working-directory: ./infraestructura
        run: terraform init

      - name: terraform destroy
        working-directory: ./infraestructura
        run: |
          terraform destroy --target helm_release.nginx-externo-tf || true
          terraform destroy -auto-approve
```

kubernetes-app-deploy.yaml: despliegue de la aplicación de Kubernetes.

Mediante este flujo se desplegará de forma manual todos los pasos necesarios para el despliegue de la aplicación de Kubernetes a través de los ficheros almacenados en la carpeta “Kubernetes-app”.

```

name: Desplegar aplicacion en Kubernetes

on:
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout del codigo
        uses: actions/checkout@v2

      - name: Instalacion de Kubernetes (kubectl)
        uses: azure/setup-kubectl@v3
        with:
          version: 'latest'

      - name: Configurar las credenciales de AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ${ secrets.AWS_REGION }

      - name: Descargar ConfigMap que usara la aplicacion
        run: aws s3 cp s3://db-services-bucket-tf/app-vars-configmap.yaml ./kubernetes-app/app-vars-configmap.yaml

      - name: Actualizar archivo kubeconfig para acceder al cluster EKS
        run: aws eks update-kubeconfig --region us-east-1 --name app-cluster-eks

      - name: Desplegar Namespace de la aplicacion
        run: kubectl apply -f ./kubernetes-app/namespace-app.yaml

      - name: Desplegar ConfigMap con las variables de la aplicacion
        run: kubectl apply -f ./kubernetes-app/app-vars-configmap.yaml

      - name: Desplegar servicios Front, Sensor y Plots
        run: kubectl apply -f ./kubernetes-app/servicios-app/

      - name: Desplegar el Ingress NGINX
        run: kubectl apply -f ./kubernetes-app/nginx-ingress.yaml

      - name: Mostrar despliegue del Ingress NGINX junto a los endpoints de la aplicacion
        run: kubectl describe ingress nginx-ingress -n app

```

4.2.5.3. Instalación de “AWS Load Balancer Controller” y “NGINX Ingress Controller”

Antes de proceder con la explicación del proceso de despliegue de la aplicación en el clúster de EKS, es necesario presentar algunos componentes importantes que deben ser desplegados en la infraestructura de AWS mediante Terraform. Estos son el “AWS Load Balancer Controller” y “Nginx Ingress Controller”, encargados de la redirección del tráfico hacia el clúster.

Los dos componentes han sido instalados mediante el administrador de paquetes de Kubernetes “Helm”, por lo que es necesaria la instalación de su proveedor para que Terraform pueda utilizarlo.

```

data "aws_eks_cluster" "eks" {
  name = aws_eks_cluster.eks-tf.name
}

data "aws_eks_cluster_auth" "eks" {
  name = aws_eks_cluster.eks-tf.name
}

provider "helm" {
  repository_config_path = "${path.module}/.helm/repositories.yaml"
  repository_cache       = "${path.module}/.helm"
  kubernetes {
    host                = data.aws_eks_cluster.eks.endpoint
    cluster_ca_certificate = base64decode(data.aws_eks_cluster.eks.certificate_authority[0].data)
    token               = data.aws_eks_cluster_auth.eks.token
  }
}

```

1. AWS Load Balancer Controller

El siguiente controlador correrá en el clúster de EKS en forma de pod y permitirá que se puedan crear balanceadores de carga en AWS.

Para desplegarlo con Terraform es necesario indicar el chart (paquete) de Helm y el repositorio a partir del cual se va a instalar. También es importante establecer el namespace kube-system dentro del clúster donde se levantará, el nombre del clúster EKS y la cuenta de servicio con la que obtendrá los permisos necesarios para operar en AWS.

```

resource "helm_release" "helm-load-balancer-tf" {
  name = "aws-load-balancer-controller"

  repository = "https://aws.github.io/eks-charts"
  chart      = "aws-load-balancer-controller"
  namespace  = "kube-system"
  version    = "1.7.2"

  set {
    name  = "clusterName"
    value = aws_eks_cluster.eks-tf.name
  }

  set {
    name  = "serviceAccount.name"
    value = "aws-load-balancer-controller" # Cuenta servicio asociada
  }

  depends_on = [aws_eks_node_group.eks-nodes-tf]
}

```

2. NGINX Ingress Controller

Gracias a este controlador que se instalará en el clúster se encargará de redirigir el tráfico entrante desde el Network Load Balancer externo hacia los pods dentro del clúster.

Para su despliegue con Terraform es necesario especificar el chart (paquete) de Helm y el repositorio a partir del cual se va a instalar, junto con el namespace Ingress donde se ejecutará.

```
resource "helm_release" "nginx-externo-tf" {
  name = "nginx-externo-tf"

  repository      = "https://kubernetes.github.io/ingress-nginx"
  chart           = "ingress-nginx"
  namespace       = "ingress"
  create_namespace = true
  version         = "4.10.1"

  values = [file("nginx/nginx-ingress.yaml")]

  depends_on = [helm_release.helm-load-balancer-tf]
}
```

Cuando se levante el controlador desplegará a su vez el Network Load Balancer asociado al este. Para ello, hay que proporcionarle el fichero con la definición del servicio NLB, que en este caso será “nginx-ingress.yaml” almacenado en el proyecto de Terraform previamente.

```
---
controller:
  ingressClassResource:
    name: external-nginx
  service:
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-type: external
      service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip
      service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
```

Este servicio será de tipo “external-nginx”, y cuando se cree un objeto Ingress con ese mismo tipo, enviará todo el tráfico a ese nuevo Ingress creado.

4.2.5.4. Despliegue de la aplicación final en Kubernetes

Tras levantar toda la infraestructura con Terraform, el orden de despliegue de la aplicación final seguirá una serie de pasos que serán automatizados por los flujos de trabajo de GitHub Actions desarrollados. Son los siguientes:

1. Actualización del archivo “kubeconfig” con el nombre del clúster EKS desplegado.

Para poder realizar operaciones sobre el clúster mediante el cliente “kubectl”, es necesario actualizar el archivo “kubeconfig” con el nombre y región del clúster. Se realiza mediante el siguiente comando utiliza también el AWS-cli:

Comando: `aws eks update-kubeconfig --region us-east-1 --name app-cluster-eks`

2. Creación del namespace de la aplicación.

Una vez establecida la conexión con el clúster, se crea el namespace “app” donde se desplegarán el resto de los objetos de Kubernetes que componen la aplicación.

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

3. Lanzar ConfigMap con las variables de entorno utilizados por los pods.

Al lanzar la infraestructura, Terraform se encargó de la construcción y subida a un bucket S3 del ConfigMap que usarán los microservicios en el clúster.

En la siguiente imagen se puede observar la creación del recurso “aws_s3_object” que se encargará de subir el fichero “app-vars-configmap.yaml” del ConfigMap. Este recurso también proporcionará al fichero en forma de variables de entorno de Terraform, las direcciones de conexión de las bases de datos de Postgres y MongoDB utilizadas por los servicios Sensor y Plots.

```
# Configmap con las variables que utilizaran los pods del cluster y las bases de datos
resource "aws_s3_object" "app-vars-configmap" {
  bucket = aws_s3_bucket.db-services-bucket-tf.id
  key     = "app-vars-configmap.yaml"
  content = templatefile("kubernetes/app-vars-configmap.yaml", {
    postgres_host = aws_db_instance.rds-sensor-postgres-tf.address
    mongo_host    = aws_instance.ec2-plots-mongo-tf.private_ip
  })
}
```

De esta manera, el fichero ConfigMap obtendrá los valores con las direcciones de las bases de datos junto a las variables “APP_PLOTS_SERVICE” y “APP_SENSOR_SERVICE” que utilizará el servicio Front para comunicarse con el resto de los servicios.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-vars-configmap
  namespace: app
data:
  # Endpoints para el servicio Front-End
  APP_PLOTS_SERVICE: localhost #plots-service
  APP_SENSOR_SERVICE: localhost #sensor-service
  # Endpoints Bases de Datos para servicios Sensor y Plots
  POSTGRES_HOST: ${postgres_host}
  MONGODB_HOST: ${mongo_host}
```

Como resultado, el fichero de ConfigMap subido al bucket s3 obtendrá todos los valores necesarios para el despliegue de la aplicación:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-vars-configmap
  namespace: app
data:
  # Endpoints para el servicio Front-End
  APP_PLOTS_SERVICE: localhost #plots-service
  APP_SENSOR_SERVICE: localhost #sensor-service
  # Endpoints Bases de Datos para servicios Sensor y Plots
  POSTGRES_HOST: sensor-postgres-tf.cvt9cc7dxazz.us-east-1.rds.amazonaws.com
  MONGODB_HOST: 10.1.30.217
```

4. Despliegue de los objetos de Kubernetes de la aplicación

Una vez desplegado el namespace “app” y el ConfigMap con las variables de entorno necesarias para la aplicación, se procede con el lanzamiento del resto de objetos de Kubernetes:

Services

Los siguientes servicios son de tipo ClusterIP y expondrán los pods con los microservicios a través de su IP privada dentro del clúster:

- Front-service: expone el pod “front-app” con la aplicación del Front-End.
- Sensor-service: expone el pod “sensor-app” con la aplicación Sensor.
- Plots-service: expone el pod “plots-app” con la aplicación Plots.

```
apiVersion: v1
kind: Service
metadata:
  name: front-service
  namespace: app
spec:
  type: ClusterIP
  ports:
    - port: 4200
      targetPort: 4200
  selector:
    app: front-app
```

```
apiVersion: v1
kind: Service
metadata:
  name: sensor-service
  namespace: app
spec:
  type: ClusterIP
  ports:
    - port: 8081
      targetPort: 8081
  selector:
    app: sensor-app
```

```
apiVersion: v1
kind: Service
metadata:
  name: plots-service
  namespace: app
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: plots-app
```

Deployments

Los siguientes deployments de encargarán de desplegar los contenedores en los pods y realizar su escalado. En este caso solo se despliega una sola réplica en cada deployment.

Dentro de la especificación de los contenedores se especificará el ConfigMap “app-vars-configmap” creado para que estos puedan importar sus variables de entorno.

- Front-app:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: front-app
  template:
    metadata:
      labels:
        app: front-app
    spec:
      containers:
        - name: front-app
          image: jesusbc/tfm-front-end:kubernetes
          ports:
            - containerPort: 4200
          envFrom:
            - configMapRef:
                name: app-vars-configmap
```

- Sensor-app:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sensor-app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sensor-app
  template:
    metadata:
      labels: # Etiquetar a los Pods
        app: sensor-app
    spec:
      containers:
        - name: sensor-app
          image: jesusbc/sensor:kubernetes
          ports:
            - containerPort: 8081
          env:
            - name: POSTGRES_HOST
              valueFrom:
                configMapKeyRef:
                  name: app-vars-configmap
                  key: POSTGRES_HOST
```

- Plots-app:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: plots-app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: plots-app
  template:
    metadata:
      labels: # Etiquetar a los Pods
      app: plots-app
    spec:
      containers:
        - name: plots-app
          image: jesusbc/plots:kubernetes
          ports:
            - containerPort: 8080
          env:
            - name: MONGODB_HOST
              valueFrom:
                configMapKeyRef:
                  name: app-vars-configmap
                  key: MONGODB_HOST
            - name: APP_SENSOR_SERVICE
              valueFrom:
                configMapKeyRef:
                  name: app-vars-configmap
                  key: APP_SENSOR_SERVICE
```

Ingress

Tras desplegar todos los microservicios de la aplicación a través de deployments y exponer los pods mediante los servicios, se levantará un objeto Ingress de tipo NGINX que habilitará la redirección del tráfico hacia el clúster de las peticiones enviadas por el Network Load Balancer.

Como puede verse en la especificación del Ingress, la regla definida espera una petición HTTP entrante con el nombre de host “app.local”. Entre sus rutas o “paths” esperadas, se encuentra

definida la dirección el directorio principal “/” del servicio Front desplegado en el puerto 4200. Dentro del Front, este se encargará internamente de redireccionar las peticiones hacia el resto de servicios del cluster de forma interna.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: app
spec:
  ingressClassName: external-nginx
  rules:
    - host: app.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: front-service
                port:
                  number: 4200
```

Gracias al Ingress, el Front podrá ser accedido en la dirección <http://app.local/>. No obstante, para poder acceder desde este host en concreto es necesario de una configuración previa del cliente.

5. Detalles importantes para el funcionamiento de la aplicación

Esta aplicación se despliega de forma remota en AWS, aunque al no utilizar un dominio registrado que resuelva los nombres DNS de los servicios, se han tenido que realizar unos pasos extra para conseguir acceder a la aplicación desde la dirección <http://app.local/> de forma “local”.

En primer lugar, para poder acceder a la aplicación desde el host “app.local” es necesario obtener la dirección IP del Network Load Balancer y registrarlo en el fichero local del sistema “/etc/hosts” para resolver manualmente el DNS desde el equipo cliente.

Se puede obtener la dirección del NLB realizando el siguiente comando “nslookup” en Windows:

```
PS C:\Users\usuario> nslookup k8s-ingress-nginxext-2005b9928e-55153f646e4a774a.elb.us-east-1.amazonaws.com
Servidor: 250.red-80-58-61.staticip.rima-tde.net
Address: 80.58.61.250

Respuesta no autoritativa:
Nombre: k8s-ingress-nginxext-2005b9928e-55153f646e4a774a.elb.us-east-1.amazonaws.com
Address: 34.228.177.130
```

Ahora se mapea la dirección obtenida en el fichero “/etc/hosts” del cliente hacia “app.local”. Con este paso, se podría acceder a la aplicación correctamente.

```
hosts
1 # Copyright (c) 1993-2009 Microsoft Corp.
2 #
3 # This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
4 #
5 # This file contains the mappings of IP addresses to host names. Each
6 # entry should be kept on an individual line. The IP address should
7 # be placed in the first column followed by the corresponding host name.
8 # The IP address and the host name should be separated by at least one
9 # space.
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27 # Direccion DNS de la app para tfm
28 34.228.177.130 app.local
29 # End of section
```

En segundo lugar, una vez que la petición es dirigida al servicio Front, este se encarga de redirigir el tráfico hacia los servicios Plots y Sensor de forma interna utilizando sus nombres de servicio por los que se identifican en el clúster. Esta tarea se complica al no disponer de un servidor DNS o mecanismo que pueda resolver estos nombres, puesto que la petición web es enviada desde el servicio cliente y no conoce los nombres internos de estos servicios.

La solución propuesta para resolver este problema es una operación de “Port-Forwarding”, creando un túnel desde la API de Kubernetes entre el cliente y el clúster de forma interna. Para ello, se mapean los puertos donde se encuentran desplegados los pods con los microservicios hacia los puertos locales del cliente.

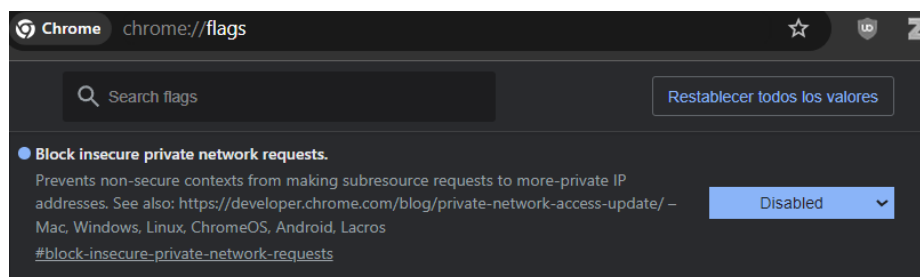
El servicio Plots se mapeará en el puerto 8080:8080 con el equipo local y el servicio Sensor en el puerto 8081.

Port Forwarding		2 items					
<input type="checkbox"/>	Name	Namespace	Kind	Pod Port	Local Port	Protocol	Status
<input type="checkbox"/>	plots-service	app	service	8080	8080	http	Active
<input type="checkbox"/>	sensor-service	app	service	8081	8081	http	Active

Tras mapear los puertos, el cliente puede acceder a los servicios directamente desde el host “localhost”. El servicio Front utilizará entonces esta dirección para redirigir al resto de servicios las peticiones puesto que ya el cliente puede resolver los nombres de forma local.

```
APP_PLOTS_SERVICE: localhost #plots-service
APP_SENSOR_SERVICE: localhost #sensor-service
```

Para finalizar, el siguiente paso es adicional y se debe realizar si se accede a la aplicación desde un navegador Chrome. Consiste en desactivar la flag “Block insecure private network request” para evitar errores de tipo CORS al navegar por la aplicación debido al arreglo realizado para acceder con un host personalizado como lo es “<http://app.local/>”.



4.3. Evaluación

4.3.1. Medición de la mejora adquirida con el nuevo sistema

El sistema inicial del que se partió para la realización de este proyecto ha experimentado una serie de beneficios debido a su actualización hacia un sistema final más potente tecnológicamente, gracias a la implantación del nuevo orquestador EKS con la capacidad de mejorar de forma continua en el tiempo debido al entorno CI/CD instalado mediante GitHub Actions.

Gracias al desarrollo práctico realizado, se ha evaluado esta mejora en forma de diversos criterios y medidas que se presentarán a continuación:

4.3.1.1. Tiempo requerido para desplegar la aplicación

Con ECS, los contenedores eran desplegados manualmente desde la interfaz de AWS a partir de tareas ECS donde se definían la configuración y despliegue de los microservicios. Ahora con Kubernetes, todos los ficheros con los pods, despliegues, servicios... del clúster son lanzados al mismo tiempo junto con la posibilidad de ejecutar todo tipo de comandos para automatizar y personalizar la configuración del despliegue.

4.3.1.2. Tiempo de despliegue del sistema completo

Las fases de compilación, empaquetado y subida de las imágenes Docker de los microservicios de la aplicación ya no se realizan de forma manual. Antes cualquier cambio en el código requería un elevado número de pasos y comandos requeridos para que el servicio estuviera listo para despliegue. Ahora, de forma automática al subir los cambios al repositorio de código, el pipeline de GitHub Actions se encarga de realizar todos estos pasos automáticamente reduciendo considerablemente el tiempo tarda aplicación en ser desplegada, puesto que cada servicio se encuentra siempre actualizado para su uso inmediato.

También, aunque en menor medida, el despliegue de la infraestructura mediante Terraform requiere de una serie de pasos y confirmaciones para desplegarse y destruirse. Al igual que con los microservicios, a través de flujos de trabajo estos pasos se realizan de una vez. Aunque este flujo se ha configurado para ser un paso manual en este proyecto, también se puede activar el despliegue de la infraestructura mediante un “push” del código al repositorio, de forma que cualquier cambio en la configuración de esta será aplicado de forma inmediata.

4.3.1.3. Automatización

La reducción de la intervención manual provoca también la reducción de los errores humanos durante el desarrollo y despliegue tanto del sistema como de la aplicación. Esto se traduce en un aumento de la eficiencia y agilidad para implementar nuevos cambios constantemente de cara a la mejora continua del proyecto.

4.3.1.4. Reproducibilidad

Ahora todas las fases del ciclo de vida del sistema siempre se realizan de la misma forma, por lo que siempre se obtendrán los mismos resultados. Los flujos de trabajo del pipeline ofrecen un alto valor de reproducibilidad al establecer los pasos ordenados para que el sistema se ejecute correctamente.

4.3.1.5. Escalabilidad

El nuevo sistema ha ganado considerablemente en cuanto a escalabilidad. Por una parte, el orquestador ECS utiliza principalmente el servicio de autoescalado proporcionado por AWS. Sin embargo, se encuentra muy limitado al no tener muchas opciones para realizar esta tarea.

EKS utiliza las capacidades de escalado proporcionadas por Kubernetes para escalar las cargas de trabajo del clúster. Por defecto, el propio Kubernetes maneja de forma automática la redistribución de cargas de trabajo cuando hay cambios en la infraestructura. Otras opciones para escalar son:

- Horizontal Pod Autoscaling (HPA): escala automáticamente el número de pods en función de métricas como uso de recursos del sistema (CPU, memoria...)
- A través de la modificación de las definiciones de los objetos de Kubernetes como Deployments o ReplicaSets.
- Compatible con el servicio de autoescalado al pertenecer al ecosistema de AWS.

4.3.1.6. Compatibilidad y portabilidad multicloud

Las aplicaciones y entorno de Kubernetes son compatibles entre los diferentes proveedores de servicios en la nube como Azure o Google Cloud. Sin embargo, la definición de tareas en ECS para desplegar los contenedores solo es válido dentro de la infraestructura de AWS.

4.3.2. Comprobación del correcto despliegue de los componentes del sistema

A continuación, para validar que la aplicación y el sistema se despliega correctamente, se adjuntarán las siguientes capturas de los servicios y componentes:

- Clúster EKS



- Grupo de nodos

Grupos de nodos (1) Información						Editar	Eliminar	Agregar grupo de nodos
	Nombre del grupo	Tamaño deseado	Versión de lanzamiento de la AMI	Plantilla de lanzamiento	Estado			
	eks-cluster-nodgroup	1	1.29.6-20240910	-	Activo			

- NLB desplegado y conectado al clúster

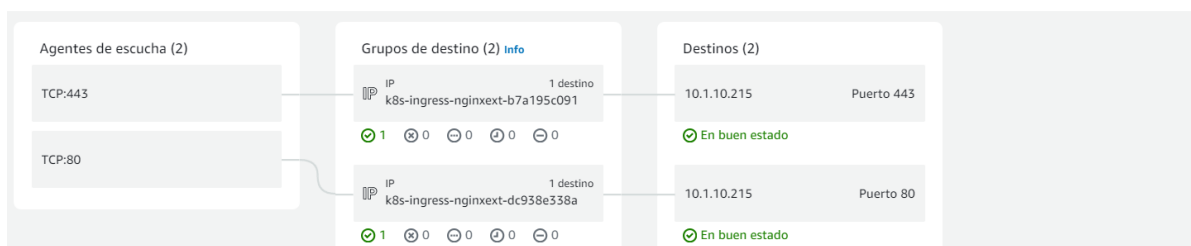
Balanceadores de carga						Acciones	Crear balanceador de carga
Balanceadores de carga (1/1)							
Elastic Load Balancing escala automáticamente la capacidad del equilibrador de carga en respuesta a los cambios en el tráfico entrante.							
Filtrar equilibradores de carga							
<input checked="" type="checkbox"/>	Nombre	Nombre de DNS	Estado	ID de VPC	Zonas c		
<input checked="" type="checkbox"/>	k8s-ingress-nginx-2005b9928e	k8s-ingress-nginx-2005b9928e-55153f646e4a774a.elb.us-east-1.amazonaws.com	Activo	vpc-04aa2de805e813...	2 Zonas disponi		

Equilibrador de carga: k8s-ingress-nginx-2005b9928e

k8s-ingress-nginx-2005b9928e

Se recuperó por última vez hace unos segundos

Exportación



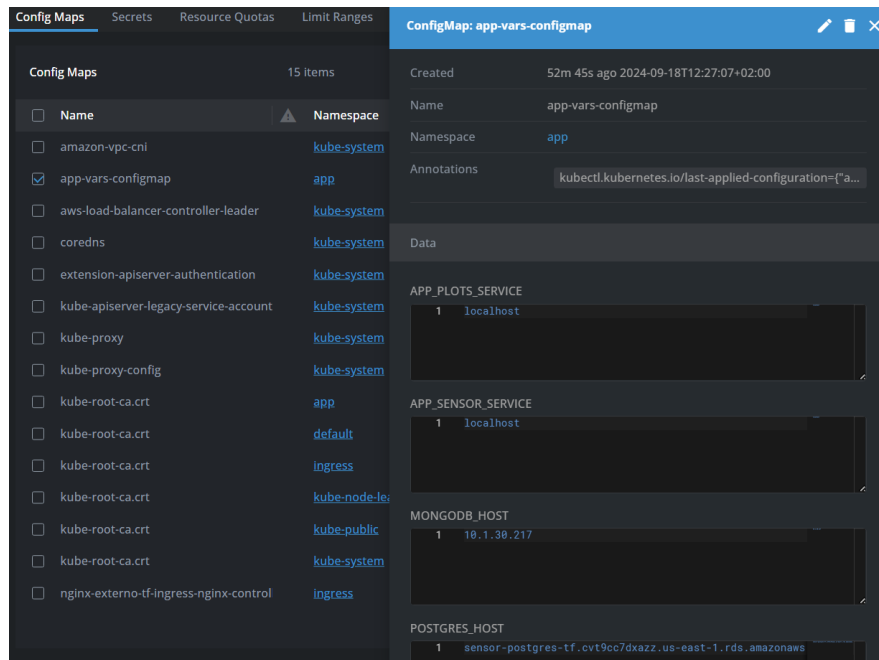
- Visualización de los objetos del clúster de Kubernetes creados satisfactoriamente y ejecutándose.

Deployments 6 items						All namespaces
<input type="checkbox"/>	Name	Namespace ▲	Pods	Replicas	Age	Conditions
<input type="checkbox"/>	front-app	app	1/1	1	48m	Available Progressing
<input type="checkbox"/>	plots-app	app	1/1	1	48m	Available Progressing
<input type="checkbox"/>	sensor-app	app	1/1	1	48m	Available Progressing

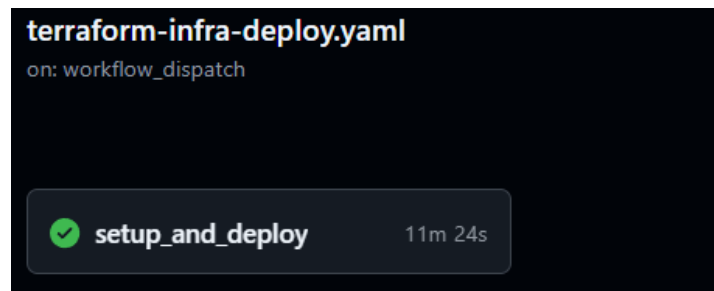
Pods 11 items				
<input type="checkbox"/>	Name	Namespace ▲	Status	Contai...
<input type="checkbox"/>	front-app-6b89897c49-8fd5c	app	Running	■
<input type="checkbox"/>	plots-app-57465956c9-kx486	app	Running	■
<input type="checkbox"/>	sensor-app-7d757f49c7-7w6ps	app	Running	■
<input type="checkbox"/>	nginx-externo-tf-ingress-nginx-c	ingress	Running	■
<input type="checkbox"/>	aws-load-balancer-controller-7b	kube-system	Running	■
<input type="checkbox"/>	aws-load-balancer-controller-7b	kube-system	Running	■

Services 8 items						All namespaces
<input type="checkbox"/>	Name	Namespace ▲	Type	Cluster IP	Ports	External IP
<input type="checkbox"/>	front-service	app	ClusterIP	172.20.142.73	4200/TCP	-
<input type="checkbox"/>	plots-service	app	ClusterIP	172.20.73.137	8080/TCP	-
<input type="checkbox"/>	sensor-service	app	ClusterIP	172.20.63.162	8081/TCP	-
<input type="checkbox"/>	kubernetes	default	ClusterIP	172.20.0.1	443/TCP	-
<input type="checkbox"/>	nginx-externo-tf-ingress-nginx-ı	ingress	LoadBalancer	172.20.200.187	80:32632/TCP, 443:	k8s-ingress-nginxext-2005b
<input type="checkbox"/>	nginx-externo-tf-ingress-nginx-ı	ingress	ClusterIP	172.20.215.187	443:webhook/TCP	-
<input type="checkbox"/>	aws-load-balancer-webhook-se	kube-system	ClusterIP	172.20.229.165	443:webhook-servi	-

Ingresses 1 item				All namespaces
<input type="checkbox"/>	Name	LoadBalancers	Namespace	Rules
<input type="checkbox"/>	nginx-ingress	k8s-ingress-nginxext-2005b9928e-551!	app	http://app.local/ → front-service:4200



- Flujos de trabajo ejecutados correctamente



kubernetes-app-deploy.yaml
on: workflow_dispatch

✓ **deploy** 19s

java-plots-sensor-service.yaml
on: push

✓ **build-plots** 55s

✓ **build-sensor** 1m 3s

node-front-service.yaml
on: push

✓ **build-front** 1m 37s

5. Conclusiones y trabajo futuro

En este último apartado del proyecto se encuentran las conclusiones que se han obtenido tras finalizar el desarrollo del mismo, así como algunas líneas de trabajo futuro con posibles ampliaciones o mejoras:

5.1. Conclusiones

En este proyecto final se han consolidado los conocimientos adquiridos en el máster a través de la resolución de una problemática basada en la actualidad empresarial como DevOps.

Esta problemática ha sido la actualización de un sistema software inicial hacia un nuevo sistema actualizado, mejorando tanto su capacidad tecnológica y garantizando su mejora continua a lo largo en tiempo. Para ello, la mejora tecnológica se ha producido al actualizar el orquestador de contenedores inicial ECS que gestionaba la aplicación hacia un clúster EKS más potente. Después, se ha mejorado radicalmente la mejora continua del sistema inicial mediante la implementación de un pipeline CI/CD, antes inexistente, con GitHub Actions. Finalmente, se han expuesto las mejoras obtenidas por el nuevo sistema con respecto al anterior.

En la consecución del objetivo principal, se han cumplido los objetivos específicos, cuya finalidad es el aprendizaje propio como futuro DevOps:

- Estudiar las tecnologías de orquestación de contenedores compatibles con AWS disponibles y justificar su elección para implementarlo en el nuevo sistema.
- Implementar el nuevo orquestador de contenedores EKS dentro del nuevo sistema.
- Estudiar las herramientas y entornos de CI/CD disponibles.
- Instalar un pipeline CI/CD dentro del nuevo sistema para gestionar el ciclo de vida de la aplicación.
- Medir la mejora del nuevo sistema con respecto al anterior.

5.2. Líneas de trabajo futuro

Una línea de trabajo futuro principal que se puede realizar sobre este proyecto es registrar un dominio para poder desplegar correctamente la aplicación de Kubernetes a través de Internet, puesto que la solución proporcionada en este trabajo ha sido adaptada para poder realizar la resolución de nombres DNS mediante configuraciones adicionales en el cliente como la modificación del fichero “/etc/hosts” y los “port-forward” de los puertos internos de los servicios de Kubernetes.

Referencias bibliográficas

- al Jawarneh, I., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., & Palopoli, A. (2019, mayo 1). *Container Orchestration Engines: A Thorough Functional and Performance Comparison*. 1-6. <https://doi.org/10.1109/ICC.2019.8762053>
- Amazon Web Services. (2014, noviembre 13). Canalización de CI/CD: AWS CodePipeline. Recuperado 18 de septiembre de 2024, de Amazon Web Services, Inc. website: <https://aws.amazon.com/es/codepipeline/>
- Amazon Web Services. (2017, diciembre 6). Servicio administrado de Kubernetes—Características de Amazon EKS - Amazon Web Services. Recuperado 3 de julio de 2024, de Amazon Web Services, Inc. website: <https://aws.amazon.com/es/eks/features/>
- Amazon Web Services. (2020a, junio 29). Registro de contenedores completamente administrado—Amazon Elastic Container Registry—Amazon Web Services. Recuperado 1 de julio de 2024, de Amazon Web Services, Inc. website: <https://aws.amazon.com/es/ecr/>
- Amazon Web Services. (2020b, julio 29). ¿Qué es Amazon EC2? - Amazon Elastic Compute Cloud. Recuperado 29 de junio de 2024, de https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/concepts.html
- Amazon Web Services. (2020c, noviembre 5). Amazon ECS vs Amazon EKS: Making sense of AWS container services | Containers. Recuperado 26 de junio de 2024, de <https://aws.amazon.com/blogs/containers/amazon-ecs-vs-amazon-eks-making-sense-of-aws-container-services/>

Amazon Web Services. (2023a, mayo 16). AWS | Gestión de contenedores (ECS) compatible con los de Docker. Recuperado 29 de junio de 2024, de Amazon Web Services, Inc. website: <https://aws.amazon.com/es/ecs/>

Amazon Web Services. (2023b, agosto 11). ¿Qué es un Equilibrador de carga de aplicación? - Elastic Load Balancing. Recuperado 30 de junio de 2024, de https://docs.aws.amazon.com/es_es/elasticloadbalancing/latest/application/introduction.html

Amazon Web Services. (2024a). *Descripción general de Amazon Web Services—AWS Documento técnico*. Recuperado de https://docs.aws.amazon.com/es_es/whitepapers/latest/aws-overview/aws-overview.pdf

Amazon Web Services. (2024b, junio 25). AWS Fargate para Amazon ECS - Amazon Elastic Container Service. Recuperado 29 de junio de 2024, de https://docs.aws.amazon.com/es_es/AmazonECS/latest/developerguide/AWS_Fargate.html

Angular. (2015, marzo 5). Angular. Recuperado 16 de septiembre de 2024, de <https://angular.dev/>

Anton Putra (Director). (2024). *Create AWS EKS Cluster using Terraform: AWS EKS Kubernetes Tutorial - Part 2*. Recuperado de <https://www.youtube.com/watch?v=uiuoNToeMFE>

Atlassian. (2022, mayo 13). Entrega continua: Primeros pasos con CI/CD. Recuperado 19 de mayo de 2024, de Atlassian website: <https://www.atlassian.com/es/continuous-delivery>

Booking. (2024). Booking.com: La mayor selección de hoteles, casas y alquileres vacacionales.

Recuperado 18 de septiembre de 2024, de Booking.com website:

<https://www.booking.com/index.es.html>

Casalicchio, E. (2019). Container Orchestration: A Survey. En A. Puliafito & K. S. Trivedi (Eds.),

Systems Modeling: Methodologies and Tools (pp. 221-235). Cham: Springer

International Publishing. https://doi.org/10.1007/978-3-319-92378-9_14

Casero, A. (2023, agosto 18). Elastic Container Service vs otras soluciones. Recuperado 23 de

junio de 2024, de <https://keepcoding.io/blog/elastic-container-service/>

Docker. (2022, mayo 10). Docker: Accelerated Container Application Development.

Recuperado 16 de septiembre de 2024, de <https://www.docker.com/>

Docker Hub. (2014, junio 8). Docker Hub Container Image Library | App Containerization.

Recuperado 16 de septiembre de 2024, de <https://hub.docker.com>

GitHub. (2020a, julio 1). GitHub Actions documentation. Recuperado 18 de septiembre de

2024, de GitHub Docs website: <https://docs.github.com/en/actions>

GitHub. (2020b, septiembre 16). Entender las GitHub Actions—Documentación de GitHub.

Recuperado 16 de septiembre de 2024, de GitHub Docs website:

<https://docs.github.com/es/actions/about-github-actions/understanding-github-actions>

GitHub. (2024). GitHub: Let's build from here. Recuperado 1 de julio de 2024, de GitHub

website: <https://github.com/>

GitLab. (2020, agosto 11). What is CI/CD? Recuperado 24 de junio de 2024, de

<https://about.gitlab.com/topics/ci-cd/>

Group, Z. (2023, agosto 29). DevOps en la era de las IA generativa. Recuperado 26 de junio de 2024, de Medium website: <https://zengtagroup.medium.com/devops-en-la-era-de-las-ia-generativa-8a2e7badf49e>

Guerrero, J., Certuche Díaz, S., Zúñiga, K., & Pardo, C. (2019, junio 1). *What is there about DevOps? Preliminary Findings from a Systematic Mapping Study*. Recuperado de https://www.researchgate.net/publication/334376433_What_is_there_about_DevOps_Preliminary_Findings_from_a_Systematic_Mapping_Study

HashiCorp. (2014, septiembre 29). Terraform by HashiCorp. Recuperado 15 de septiembre de 2024, de Terraform by HashiCorp website: <https://www.terraform.io/>

Helm. (2021, abril 14). Helm. Recuperado 15 de septiembre de 2024, de <https://helm.sh/es/>

IBM. (2023, julio 13). ¿Qué es el despliegue continuo? | IBM. Recuperado 19 de mayo de 2024, de <https://www.ibm.com/es-es/topics/continuous-deployment>

IBM. (2024, abril 24). ¿Qué es Terraform? | IBM. Recuperado 15 de septiembre de 2024, de <https://www.ibm.com/es-es/topics/terraform>

Jenkins. (2024, septiembre 4). Jenkins. Recuperado 18 de septiembre de 2024, de Jenkins website: <https://www.jenkins.io/>

Kubernetes. (2018a, abril 28). How it works—Ingress-Nginx Controller. Recuperado 15 de septiembre de 2024, de <https://kubernetes.github.io/ingress-nginx/how-it-works/>

Kubernetes. (2018b, junio 1). Welcome—AWS Load Balancer Controller. Recuperado 15 de septiembre de 2024, de <https://kubernetes-sigs.github.io/aws-load-balancer-controller/latest/>

Kubernetes. (2020a, febrero 13). Booking.com Case Study. Recuperado 18 de septiembre de 2024, de Kubernetes website: <https://kubernetes.io/case-studies/booking-com/>

Kubernetes. (2020b, agosto 8). Componentes de Kubernetes. Recuperado 6 de julio de 2024, de <https://kubernetes.io/es/docs/concepts/overview/components/>

Kubernetes. (2022, julio 17). ¿Qué es Kubernetes? Recuperado 3 de julio de 2024, de <https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/>

Kubernetes. (2023, julio 25). Entender los Objetos de Kubernetes. Recuperado 16 de septiembre de 2024, de Kubernetes website: <https://kubernetes.io/es/docs/concepts/overview/working-with-objects/kubernetes-objects/>

Kubernetes. (2024, enero 1). Command line tool (kubectl). Recuperado 6 de julio de 2024, de <https://kubernetes.io/docs/reference/kubectl/>

Lewandowski, B. (2023, julio 4). DevOps & Agile: Synergy for Software Development Success. Recuperado 30 de abril de 2024, de NDS website: <https://newdigitalstreet.com/devops-agile/>

Microsoft. (2024). ¿Qué es un contenedor? | Microsoft Azure. Recuperado 23 de abril de 2024, de <https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-a-container>

mijacobs. (2023, octubre 5). ¿Qué es la entrega continua? - Azure DevOps. Recuperado 20 de mayo de 2024, de <https://learn.microsoft.com/es-es/devops/deliver/what-is-continuous-delivery>

- Mohindroo, S. K. (2023, agosto 2). Mastering Kubernetes: Empowering Organizations in the World of Containers | LinkedIn. Recuperado 24 de junio de 2024, de <https://www.linkedin.com/pulse/mastering-kubernetes-empowering-organizations-world-mohindroo/>
- nOps. (2024, marzo 25). AWS EKS Vs. ECS: The Ultimate Guide. Recuperado 24 de junio de 2024, de nOps website: <https://www.nops.io/blog/aws-eks-vs-ecs-the-ultimate-guide/>
- Novotný, A. (2023, julio 3). AWS ECS vs. EKS: Breaking Down the Pros and Cons! | StormIT. Recuperado 30 de junio de 2024, de <https://www.stormit.cloud/blog/aws-ecs-vs-eks/>
- Red Hat. (2022, noviembre 5). La integración y la distribución continuas (CI/CD). Recuperado 17 de mayo de 2024, de <https://www.redhat.com/es/topics/devops/what-is-ci-cd>
- Redacción APD. (2023, febrero 14). Metodología Lean: Qué es y cómo puede ayudar a tu empresa. Recuperado 25 de junio de 2024, de APD España website: <https://www.apd.es/metodologia-lean-que-es/>
- Redondo, A. M. F., & Cárdenas, F. de J. N. (2022). DevOps: Un vistazo rápido. *Ciencia Huasteca Boletín Científico de la Escuela Superior de Huejutla*, 10(19), 35-40. <https://doi.org/10.29057/esh.v10i19.8121>
- Sanjurjo Royo, E. (2022). *Modelo de madurez de BizDevOps* ([Http://purl.org/dc/dcmitype/Text](http://purl.org/dc/dcmitype/Text), Universidade da Coruña; p. 1). Universidade da Coruña. Recuperado de <https://dialnet.unirioja.es/servlet/tesis?codigo=307662>

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment:

A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, PP.

<https://doi.org/10.1109/ACCESS.2017.2685629>

Spring Boot. (2013, septiembre 9). Spring Boot. Recuperado 16 de septiembre de 2024, de

Spring Boot website: <https://spring.io/projects/spring-boot>