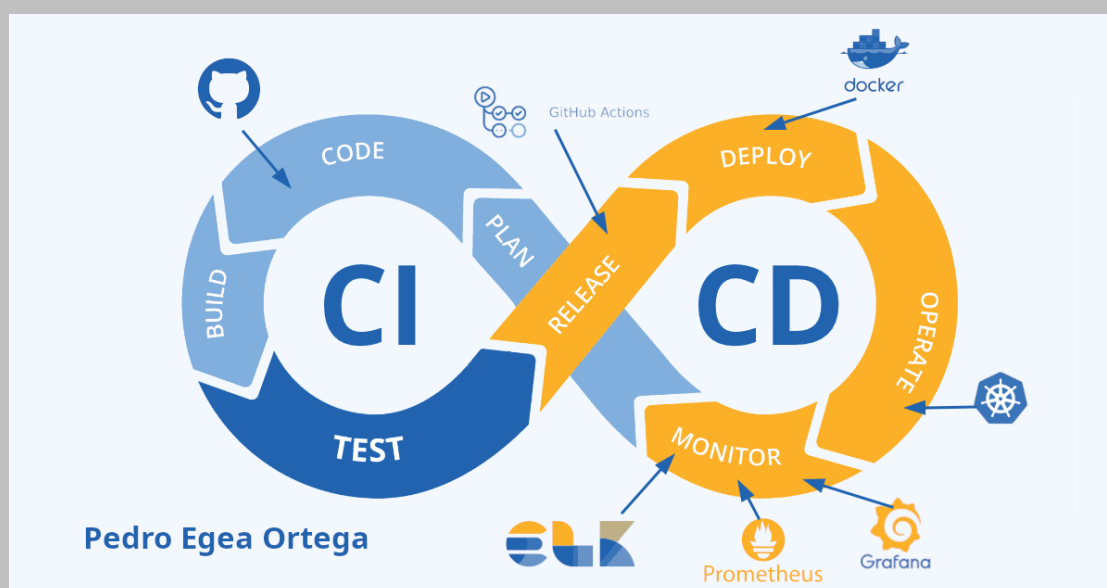


## MEMORIA FINAL DE PROYECTO

### IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES



**CICLO FORMATIVO DE GRADO SUPERIOR  
ASIR**

**AUTOR  
PEDRO EGEA ORTEGA**

**TUTOR  
EDUARDO TELLECHEA AMESTI**

**COORDINADOR  
JESÚS VIVES CESPEDES**



## **DEDICATORIAS/AGRADECIMIENTOS**

- Eduardo Tellechea Amesti
- Jesús Vives Cespedes
- Raúl Riesco Montes
- Jose Manuel Carreres Paredes
- Juan Pedro Monteagudo Toledo
- Maria Assumpcio Boix Garcia
- Joaquin Buforn Perez
- Esteban Torres Ibañez
- Amalia Torres Antonio
- Maria del Carmen Cruz Ortiz
- Sonia Guerrero Perez
- Antonia Aguilar Anton

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## INDICE

1 INTRODUCCIÓN.....	5
2 INTRODUCTION.....	6
3 ALCANCE DEL PROYECTO.....	8
4 ESTUDIO DE VIABILIDAD.....	10
4.1 Estado actual del sistema.....	10
4.2 Requisitos.....	10
4.3 Posibles soluciones.....	11
4.4 Solución elegida.....	12
4.5 Planificación temporal de las tareas del proyecto.....	12
4.6 Planificación de los recursos a utilizar.....	13
5 ANÁLISIS.....	14
5.1 Requisitos funcionales.....	14
5.2 Requisitos no funcionales.....	15
6 TECNOLOGÍAS CLAVE DEL PROYECTO.....	16
6.1 Docker y contenerización.....	16
6.2 Kubernetes como orquestador.....	18
6.3 GitHub Actions como plataforma CI/CD.....	20
6.4 Herramientas de monitorización y logging.....	21
6.4.1 Grafana.....	21
6.4.2 Prometheus.....	23
6.4.3 ELK.....	24
7 IMPLEMENTACIÓN.....	26
7.1 Estructura del workflow (flujo de trabajo).....	27
7.1.1 Explicación del workflow de GitHub Actions.....	29
7.2 Creación y uso de secrets.....	35
7.2.1 Secrets de un repositorio en GitHub.....	35
7.2.2 GHCR_TOKEN.....	37
7.2.3 KUBE_CONFIG.....	40
7.3 Crear un self-hosted runner.....	41
7.4 Consideraciones de seguridad y buenas prácticas.....	43
7.5 Implmentaciones de código realizadas.....	43
8 PRUEBAS.....	44
8.1 Pruebas de funcionamiento.....	44
8.2 Pruebas de mantenimiento.....	46
8.2.1 <i>Comprobación del estado de los pods en Kubernetes</i> .....	46
8.2.2 <i>Revisión de logs de los contenedores</i> .....	47
8.2.3 <i>Simulación de un despliegue automático</i> .....	47
8.2.4 <i>Copia de seguridad de la base de datos</i> .....	48
8.2.5 <i>Control del almacenamiento de logs</i> .....	48
9 ANEXOS.....	49
10 CONCLUSIONES.....	53
11 FUENTES.....	55
11.1 <i>Páginas web y bibliografía consultadas</i> .....	56

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## 1 INTRODUCCIÓN

En la actualidad, el desarrollo de software está en constante evolución. Cada vez se busca entregar aplicaciones más rápido, con mayor calidad y con menos errores. Para lograrlo, han surgido nuevas formas de trabajar, como las metodologías DevOps y los sistemas de integración y entrega continua (CI/CD), que ayudan a automatizar procesos y a mejorar la colaboración entre los equipos.

DevOps es una filosofía que une el trabajo de los desarrolladores (quienes crean el software) con el de los administradores de sistemas (quienes lo hacen funcionar en los servidores). El objetivo es que ambos colaboren desde el inicio del proyecto para que todo fluya mejor y más rápido. Las personas que se dedican a DevOps se encargan de preparar entornos automatizados donde el software se puede construir, probar, desplegar y monitorizar casi sin intervención manual, lo que reduce errores y mejora la eficiencia.

Este proyecto tiene como objetivo crear un sistema CI/CD para una aplicación web que funciona dentro de contenedores, usando herramientas actuales como GitHub Actions, Docker y Kubernetes. Gracias a este sistema, se podrá automatizar gran parte del trabajo que antes se hacía a mano, como subir nuevas versiones o configurar entornos, haciendo todo el proceso más sencillo y seguro. Además, se incluirá un sistema de monitorización con herramientas como Prometheus, Grafana y ELK Stack, que permiten vigilar el estado del sistema en todo momento y detectar cualquier problema antes de que afecte a los usuarios.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

La ejecución de este proyecto no solo permitirá poner en práctica los conocimientos adquiridos durante el ciclo formativo de Técnico Superior en **Administración de Sistemas Informáticos en Red (ASIR)**, sino que también simulará un escenario realista alineado con las demandas actuales de la industria en administración de sistemas, automatización e infraestructura como código.

## 2 INTRODUCTION

Nowadays, software development is constantly evolving. There is an increasing demand to deliver applications faster, with higher quality, and fewer errors. To achieve this, new approaches such as DevOps methodologies and Continuous Integration/Continuous Deployment (CI/CD) systems have gained prominence by enabling teams to deliver software in a more agile, secure, and efficient way.

DevOps is a mindset that combines the efforts of developers (who write the software) and operations teams (who ensure it runs reliably). The goal is to foster collaboration between both sides from the beginning of a project to streamline workflows and reduce delivery times. DevOps professionals play a key role by designing and managing automated infrastructures where applications can be built, tested, deployed, and monitored continuously, minimizing manual intervention and reducing the risk of errors.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

This project aims to implement a CI/CD pipeline for a containerized web application using modern tools such as GitHub Actions, Docker, and Kubernetes. The main objective is to automate the development and deployment process, eliminating repetitive manual tasks, reducing human errors, and making version control and environment management easier. In addition, the system will include a monitoring and observability setup using tools like Prometheus, Grafana, and the ELK Stack. These tools provide continuous insight into the system's health and performance, allowing proactive responses to any incidents.

The execution of this project will not only put into practice the knowledge acquired during the Advanced Technician in **Networked Computer Systems (ASIR)** program, it will also simulate a real scenario aligned with current industry demands in system administration, automation, and infrastructure as code.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

### 3 ALCANCE DEL PROYECTO

Este proyecto tiene como objetivo ejemplificar como la implementación de un sistema de integración y entrega continua (CI/CD) puede agilizar la fase de desarrollo de cualquier aplicación. Este sistema permitirá automatizar los procesos repetitivos, asegurar la calidad del código y facilitar la colaboración entre los equipos de desarrollo y operaciones; consiguiendo así velocidad y eficiencia, y evitando errores humanos.

Además, la incorporación de contenedores en el flujo de desarrollo no solo complementa el sistema CI/CD, sino que amplía significativamente el alcance del proyecto al proporcionar una solución integral para la gestión del ciclo de vida completo de la aplicación, desde la fase de desarrollo local hasta la puesta en producción. Este enfoque basado en contenedores garantiza que cada fase del proceso sea consistente, reproducible y automatizada, maximizando así los beneficios de la implementación CI/CD.

El uso de **Kubernetes** como orquestador de estos contenedores extiende aún más este alcance, permitiendo gestionar de manera declarativa no solo los contenedores individuales sino todo el ecosistema de servicios, configuraciones y dependencias necesarias para el funcionamiento de la aplicación en cualquier entorno.



IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

Gracias a la **contenerización** de los servicios del proyecto se obtienen ventajas como:

- **Estandarización:**
  - Elimina el problema "en mi máquina funciona" mediante la definición de entornos idénticos para todos los desarrolladores.
  - Garantía de que todos los miembros del equipo trabajan con las mismas versiones de dependencias, bibliotecas y servicios.
  - Simplificación del proceso de *onboarding* para nuevos desarrolladores, que pueden comenzar a trabajar con la aplicación en minutos.
- **Aislamiento y compatibilidad:**
  - Capacidad para ejecutar múltiples versiones de servicios o dependencias sin conflictos.
  - Aislamiento de la aplicación y sus dependencias del sistema *host*, reduciendo problemas de compatibilidad.
- **Optimización de recursos:**
  - Uso más eficiente de los recursos *hardware* disponibles en comparación con máquinas virtuales tradicionales.
  - Arranque más rápido de entornos de desarrollo completos.
- **Gestión de versiones:**
  - Posibilidad de mantener diferentes versiones de la aplicación o servicios como imágenes **Docker** etiquetadas.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## 4 ESTUDIO DE VIABILIDAD

### 4.1 Estado actual del sistema

Actualmente, el desarrollo y despliegue de la aplicación se realiza de forma manual. Cada nueva versión requiere intervención directa del administrador para construir la imagen del contenedor, detener el servicio anterior y lanzar el nuevo, además de comprobar que todo funcione correctamente.

Esta metodología conlleva una serie de inconvenientes como mayor riesgo de errores humanos, desfase entre versiones en diferentes entornos o dificultades para reproducir entornos de despliegue. Esto dificulta la escalabilidad del proyecto y ralentiza el desarrollo continuo de nuevas funcionalidades.

### 4.2 Requisitos

Diseñar, desarrollar e implementar un *pipeline* CI/CD para optimizar el proceso de desarrollo y despliegue de una aplicación basada en contenedores.

Este *pipeline* deberá automatizar tareas repetitivas, garantizar la coherencia entre entornos, reducir errores humanos y facilitar la gestión de versiones.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

### 4.3 Posibles soluciones

Solución	Ventajas	Inconvenientes
<b>Jenkins + Docker + Kubernetes</b>	<ul style="list-style-type: none"> <li>- Muy flexible</li> <li>- Usado en empresas grandes</li> <li>- Jenkins tiene gran cantidad de plugins</li> <li>- Independencia de proveedores externos</li> </ul>	<ul style="list-style-type: none"> <li>- Requiere instalación, configuración y mantenimiento propios</li> <li>- Mayor complejidad inicial y curva de aprendizaje</li> </ul>
<b>GitLab CI + Docker + Kubernetes</b>	<ul style="list-style-type: none"> <li>- Todo en una sola plataforma</li> <li>- Fácil integración con runners propios</li> </ul>	<ul style="list-style-type: none"> <li>- Puede consumir más recursos si se instala en local</li> <li>- Algunos runners requieren configuración avanzada</li> <li>- La versión completa de funcionalidades requiere suscripción premium</li> </ul>
<b>GitHub Actions + Docker + Kubernetes</b>	<ul style="list-style-type: none"> <li>- Integración directa con GitHub</li> <li>- Muy fácil de usar para proyectos pequeños/medianos</li> <li>- Funciona en la nube sin infraestructura propia</li> <li>- Permite ejecutar runners propios (self-hosted), lo que mitiga la dependencia total del entorno GitHub</li> </ul>	<ul style="list-style-type: none"> <li>- Dependencia del entorno de GitHub</li> <li>- Ligeramente menos flexible que Jenkins o GitLab en tareas avanzadas</li> </ul>
<b>GitHub Actions + Docker + Kubernetes + Cloud Computing</b>	<ul style="list-style-type: none"> <li>- No hay necesidad de un ordenador potente para ejecutar los servicios, solo se necesita interfaz gráfica y conexión a Internet</li> <li>- Fácil despliegue de los servicios a Internet</li> </ul>	<ul style="list-style-type: none"> <li>- Imposibilidad de coste 0, dado que las capas gratuitas de las principales plataformas como <b>AWS</b>, <b>Azure</b> o <b>Google Cloud</b> otorgan solo máquinas de 1 CPU y Kubernetes necesita mínimo 2 CPU</li> </ul>

IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

#### 4.4 Solución elegida

La solución elegida es **GitHub Actions + Docker + Kubernetes**, por las siguientes razones:

- **GitHub** es la plataforma base de repositorios de la gran mayoría de proyectos.
- **GitHub Actions** ofrece flujos de trabajo CI/CD de forma nativa e integrada.
- **Docker** es la tecnología estándar para la contenerización de aplicaciones.
- **Kubernetes** permite una orquestación robusta y escalable, incluso en entornos de prueba locales con herramientas como **Minikube** o **Kind**.
- Esta solución ofrece un buen equilibrio entre profesionalidad, escalabilidad, coste cero y facilidad de aprendizaje.

#### 4.5 Planificación temporal de las tareas del proyecto

- Estudio de herramientas: 6 días
- Diseño del sistema: 5 días
- Desarrollo del *workflow*: 10 días
- Pruebas y ajustes: 14 días
- Documentación y presentación: 10 días

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## ***4.6 Planificación de los recursos a utilizar***

Para solventar los problemas que plantea el proyecto se utilizarán:

- **Recursos hardware y software:**
- Ordenador personal (mínimo 16 GB RAM, CPU 6 núcleos, 500 GB SSD).
- Sistema operativo **Linux**, ejecutado en máquinas virtuales.
- **Docker** y **Kubernetes** instalados en máquinas virtuales.
- Clúster local con **Minikube**.

### **Recursos online:**

- Cuenta gratuita en **GitHub**.
- Documentación oficial (**GitHub Actions**, **Docker**, **Kubernetes**).
- Apoyo en tutoriales de **YouTube**.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## 5 ANÁLISIS

En esta fase se definen los requisitos del sistema en base a las necesidades planteadas y se justifica la elección de la solución técnica. El objetivo es garantizar que el sistema cumpla con las expectativas del usuario final y sea viable tanto a nivel técnico como práctico.

### 5.1 Requisitos funcionales

1. **Automatización de la integración continua (CI):** El sistema debe compilar el código y ejecutar las pruebas automáticamente al realizar un *push* o *pull request* en el repositorio de **GitHub**.
2. **Automatización del despliegue continuo (CD):** El sistema debe construir la imagen **Docker** y desplegarla automáticamente en un clúster de **Kubernetes** tras superar las pruebas.
3. **Construcción de imágenes Docker:** El sistema debe crear imágenes **Docker** a partir del código fuente, etiquetarlas correctamente y almacenarlas en un registro de contenedores (local o remoto).
4. **Despliegue en entorno de desarrollo:** El sistema debe desplegar la aplicación automáticamente en un entorno de pruebas local mediante **Minikube**.
5. **Rollback automático:** En caso de fallo durante el despliegue o ejecución de pruebas, el sistema debe revertir a la versión anterior estable del contenedor.
6. **Verificación del estado del clúster:** El sistema debe verificar el estado de los *pods* y servicios desplegados para asegurar que están funcionando correctamente tras cada despliegue.

IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

7. **Notificación de errores:** En caso de fallos en la ejecución de cualquier paso del *pipeline*, el sistema debe notificar al equipo de desarrollo a través del sistema de **GitHub** (issues, correos o alertas).
8. **Gestión de versiones:** El sistema debe gestionar las versiones de los despliegues mediante etiquetas (tags) en el repositorio y en las imágenes **Docker**.

## 5.2 Requisitos no funcionales

1. **Escalabilidad:** La solución debe poder adaptarse a un entorno más complejo sin necesidad de rediseño total (por ejemplo, migrar de **Minikube** a un clúster en la nube).
2. **Portabilidad:** El sistema debe poder ejecutarse en distintos entornos Linux sin modificaciones importantes, gracias al uso de contenedores **Docker**.
3. **Fiabilidad:** El sistema debe garantizar la consistencia de los despliegues y ofrecer mecanismos de recuperación ante errores (rollback automático).
4. **Mantenibilidad:** El *pipeline* y los archivos de configuración deben estar bien documentados y organizados para facilitar su mantenimiento.
5. **Seguridad:** Se deben minimizar los accesos innecesarios, usar *secrets* en variables protegidas y evitar almacenar contraseñas en texto plano.
6. **Observabilidad:** El sistema debe permitir monitorizar el estado de los servicios, mediante herramientas como **Prometheus**, **Grafana** y **ELK Stack**.
7. **Rendimiento:** El *pipeline* no debe introducir retrasos innecesarios en el ciclo de desarrollo. Los despliegues deben completarse en un tiempo razonable.
8. Se usará exclusivamente software de **código abierto y gratuito**

## 6 TECNOLOGÍAS CLAVE DEL PROYECTO

El proyecto hace uso de herramientas modernas como **Docker** para la contenerización de la aplicación, **Kubernetes** para su orquestación, y **GitHub Actions** como plataforma de integración y entrega continua (CI/CD). Además, se incorporan herramientas de monitorización y *logging* como **Grafana**, **Prometheus** y la pila **ELK**, con el objetivo de supervisar el estado de la infraestructura y la aplicación en tiempo real.

### 6.1 Docker y contenerización



**Docker** es una plataforma de código abierto que permite automatizar la creación, despliegue y ejecución de aplicaciones dentro de contenedores. Los contenedores son unidades ligeras, portables y autosuficientes que incluyen todo lo necesario para ejecutar una aplicación o un servicio: el código, las librerías, las dependencias y el sistema de archivos necesario.

A diferencia de las máquinas virtuales tradicionales, los contenedores no requieren un sistema operativo completo, sino que comparten el núcleo del sistema operativo del *host*. Esto los hace más eficientes en términos de consumo de recursos y más rápidos de iniciar.



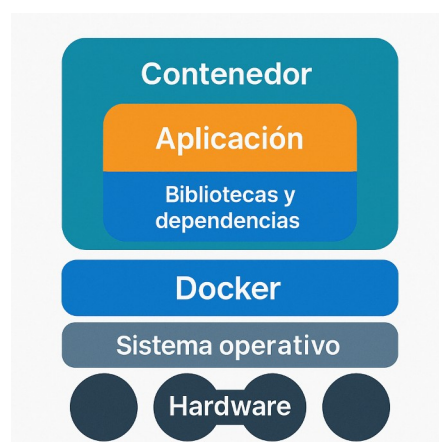
<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

¿Qué es la contenerización? La contenerización es el proceso mediante el cual una aplicación y todas sus dependencias se empaquetan en un contenedor. Este contenedor puede ejecutarse en cualquier entorno que tenga **Docker** instalado, lo que garantiza la portabilidad y la consistencia del comportamiento de la aplicación entre entornos de desarrollo, pruebas y producción.

Entre sus ventajas destacan:

- **Portabilidad:** se puede ejecutar en cualquier sistema sin preocuparse por conflictos de dependencias.
- **Aislamiento:** cada contenedor es independiente, lo que facilita la gestión y seguridad del sistema.
- **Escalabilidad:** facilita la orquestación y despliegue de aplicaciones en arquitecturas modernas basadas en **microservicios**.
- **Integración con CI/CD:** se integra fácilmente con herramientas como **GitHub Actions** para automatizar pruebas y despliegues.

En este proyecto, se ha utilizado **Docker** para contenerizar una aplicación Flask, permitiendo así su despliegue automatizado en un entorno orquestado con **Kubernetes**. La arquitectura de contenedores con **Docker** se ve de la siguiente manera:



El siguiente diagrama representa la arquitectura básica de contenerización usando **Docker**. A partir del código fuente de la aplicación, se define un **Dockerfile** que contiene las instrucciones necesarias para construir la imagen del contenedor. Esta imagen se ejecuta como contenedor sobre el motor **Docker**, aislada del sistema anfitrión, pero con acceso a los recursos necesarios.

## 6.2 Kubernetes como orquestador



# kubernetes

**Kubernetes** (también conocido como K8s) es una plataforma de código abierto desarrollada inicialmente por [Google](#) y actualmente mantenida por la **Cloud Native Computing Foundation** (CNCF). Su propósito es automatizar el despliegue, escalado y gestión de aplicaciones contenerizadas.

En esencia, **Kubernetes** actúa como un orquestador de contenedores: gestiona múltiples contenedores distribuidos en un clúster de máquinas, garantizando su disponibilidad, escalabilidad, equilibrio de carga y recuperación ante fallos.

Las posibilidades que ofrece **Kubernetes** son infinitas y podrían requerir un proyecto aparte para abarcarlas todas, pero para este proyecto solo necesitamos conocer algunos de sus principales componentes:

- **Pod:** la unidad más pequeña en **Kubernetes**. Puede contener uno o varios contenedores que comparten red y almacenamiento.
- **Cluster:** conjunto de nodos gestionados por un plano de control.
- **Deployment:** define cómo debe crearse y gestionarse un conjunto de Pods (por ejemplo, cuántas réplicas deben estar siempre activas).
- **Service:** expone una aplicación como un servicio accesible dentro o fuera del clúster, y balancea la carga entre los Pods disponibles.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

- **Kubecttl**: herramienta de línea de comandos que permite interactuar con el clúster de **Kubernetes**. Con ella se pueden crear, eliminar, monitorizar o modificar recursos definidos en archivos **YAML**.
- **Node**: máquina física o virtual que forma parte del clúster y que ejecuta los Pods. Cada Node contiene un *runtime* de contenedores (por ejemplo, **Docker**) además de los servicios necesarios para comunicarse con el plano de control de **Kubernetes**.
- **Namespace**: permite agrupar recursos dentro del clúster en entornos lógicos separados. Por ejemplo, en este proyecto se ha usado un namespace *monitoring* para **Grafana** y **Prometheus**, y *logging* para **ELK Stack**. Esto facilita la organización, el aislamiento y la gestión de los recursos.

### 6.3 GitHub Actions como plataforma CI/CD



## GitHub Actions

**GitHub Actions** es una plataforma de automatización de flujos de trabajo (workflows) integrada en **GitHub**, que permite ejecutar tareas como pruebas, compilaciones y despliegues directamente desde los repositorios. Su enfoque basado en eventos (por ejemplo, push, pull request, release, etc.) permite definir *pipelines* de CI/CD de forma sencilla y totalmente integrada con el control de versiones.

En este proyecto, **GitHub Actions** se utiliza para implementar la fase de Integración Continua (CI) y Despliegue Continuo (CD) de la aplicación Flask. Mediante un archivo de configuración **YAML**, se define un *workflow* que automatiza los pasos desde que se realiza un *push* hasta que la aplicación está lista para desplegarse en Kubernetes.

Sus principales características son:

- **Workflows:** definidos mediante archivos `.yaml` en el directorio `.github/workflows/`.
- **Ejecución basada en eventos:** como *push*, *pull\_request* o *release*.
- **Jobs y steps:** permite dividir tareas en pasos individuales, como testeo, construcción de imágenes **Docker** o despliegue.
- **Entorno flexible:** se puede ejecutar código en *runners* Linux, Windows o macOS, e incluso en **runners personalizados**.

## ***6.4 Herramientas de monitorización y logging***

### ***6.4.1 Grafana***



**Grafana** es una plataforma de código abierto para visualización y análisis de métricas, que permite crear paneles interactivos y alertas a partir de diversas fuentes de datos. Es ampliamente utilizada en entornos DevOps y de observabilidad, especialmente en combinación con herramientas como Prometheus, InfluxDB o Elasticsearch.

En el contexto de este proyecto, Grafana se utiliza como parte del sistema de monitorización, permitiendo observar en tiempo real el estado de la aplicación Flask, los contenedores desplegados en Kubernetes, así como el rendimiento de los nodos del clúster.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

Sus principales características son:

- **Visualización avanzada:** permite crear dashboards personalizados con gráficos, tablas y alertas.
- **Integración con múltiples fuentes de datos:** como Prometheus (en este caso), Elasticsearch, Loki, MySQL, entre otras.
- **Alertas configurables:** permite definir umbrales y notificaciones mediante canales como correo electrónico por ejemplo.
- **Acceso web:** interfaz accesible desde un navegador, generalmente expuesta mediante un *NodePort* en Kubernetes.

### 6.4.2 Prometheus



# Prometheus

**Prometheus** es una herramienta de monitorización y recolección de métricas de código abierto, diseñada especialmente para sistemas distribuidos y dinámicos como los desplegados en contenedores y orquestados con **Kubernetes**. Su arquitectura basada en consultas *pull* y su lenguaje propio de consultas, **PromQL**, lo hacen ideal para observar el comportamiento del sistema y detectar problemas de rendimiento.

En este proyecto, **Prometheus** cumple la función de recolectar métricas de la aplicación Flask, los contenedores Docker y los componentes del clúster Kubernetes, proporcionando así una visión detallada del estado del sistema en tiempo real.

Sus principales características son:

- **Recolección por scraping:** Prometheus accede periódicamente a los endpoints */metrics* expuestos por los servicios y recoge datos en formato texto plano.
- **Almacenamiento local:** guarda las métricas en una base de datos TSDB (Time Series Database).

- **PromQL** (Prometheus Query Language): lenguaje de consultas para generar métricas agregadas, filtros o cálculos en tiempo real.
- **Alertmanager** (opcional): permite gestionar alertas y enviar notificaciones.

### 6.4.3 ELK



**ELK** es un conjunto de herramientas de código abierto compuesto por **Elasticsearch**, **Logstash** y **Kibana**, diseñado para recolectar, almacenar, buscar, analizar y visualizar logs y datos en tiempo real.

Este stack está desarrollado principalmente en Java, por lo que requiere que el sistema anfitrión cuente con una JVM (Java Virtual Machine) instalada. No obstante, en entornos como Docker o Kubernetes, este requisito suele estar resuelto mediante el uso de imágenes oficiales que ya incluyen la versión adecuada de Java.

Además, los logs generados por los servicios y la aplicación se formatean en JSON, lo cual facilita su procesamiento e indexación en Elasticsearch, mejorando tanto la estructura de los datos como su posterior visualización en Kibana.



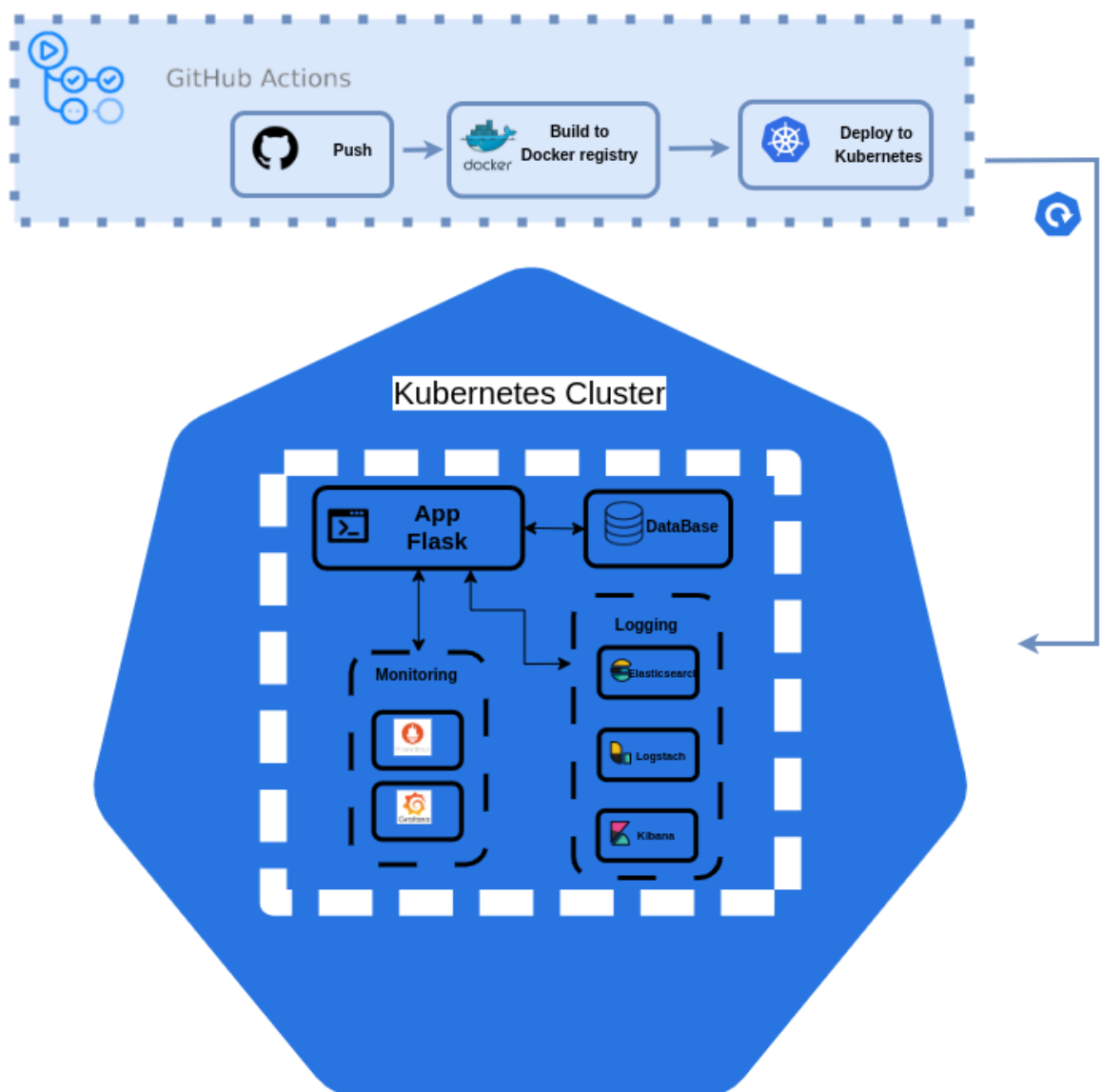
<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

Componentes principales del stack **ELK**:

- **Elasticsearch**: motor de búsqueda y análisis distribuido, almacena los datos indexados.
- **Logstash**: canaliza, transforma y envía logs desde múltiples fuentes a Elasticsearch.
- **Kibana**: interfaz gráfica para visualizar y analizar los datos almacenados en Elasticsearch.

## 7 IMPLEMENTACIÓN

Partiendo del diseño previamente definido, en esta fase se procede a la construcción del sistema. Se lleva a cabo el desarrollo de los distintos componentes que conforman el proyecto, incluyendo tanto el desarrollo de la lógica de la aplicación como la configuración de los servicios necesarios para su correcto funcionamiento.



## 7.1 Estructura del workflow (flujo de trabajo)

Con el objetivo de automatizar el ciclo de vida de la aplicación, se ha implementado el *workflow* de integración y despliegue continuo (CI/CD) utilizando **GitHub Actions**, que permite compilar, empaquetar y desplegar la aplicación Flask en un clúster de Kubernetes, así como configurar todo el entorno de monitorización y registro de logs.

Este *workflow* se encuentra definido en un archivo **.yml** dentro del directorio **.github/workflows/** del repositorio y se ejecuta automáticamente ante ciertos eventos del repositorio siempre y cuando haya algún *runner listening for jobs*, sino quedará como pendiente.

A continuación, se muestra el contenido del *workflow* y posteriormente se desarrollará cada parte del código.

```

1  name: Build and Deploy Flask App with Docker
2
3  on:
4    push:
5      branches:
6        - main
7    pull_request:
8      branches:
9        - main
10
11 jobs:
12   build:
13     runs-on: self-hosted
14
15     steps:
16       - name: Checkout code
17         uses: actions/checkout@v2
18
19       - name: Log in to GitHub Container Registry
20         uses: docker/login-action@v2
21         with:
22           registry: ghcr.io
23           username: ${ github.actor }
24           password: ${ secrets.GHCR_TOKEN }
25
26       - name: Build Docker image
27         run: docker build -t ghcr.io/pederysky/tfg_1/flask_tienda:latest .
28
29       - name: Push Docker image to GHCR
30         run: docker push ghcr.io/pederysky/tfg_1/flask_tienda:latest
31
32   deploy:
33     runs-on: self-hosted
34     needs: build
35

```

```

36 steps:
37   - name: Install kubectl
38     run: |
39       curl -LO "https://dl.k8s.io/release/${curl -L -s https://dl.k8s.io/release/stable.txt}/bin/linux/amd64/kubectl"
40       chmod +x kubectl
41       sudo mv kubectl /usr/local/bin/
42
43   - name: Install Helm
44     run: |
45       curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
46       chmod +x get_helm.sh
47       ./get_helm.sh
48       echo "export PATH=/usr/local/bin:\$PATH" >> $GITHUB_ENV
49
50   - name: Configure kubeconfig
51     run: |
52       mkdir -p $HOME/.kube
53       echo "${ secrets.KUBE_CONFIG }}" > $HOME/.kube/config
54       echo "KUBECONFIG=$HOME/.kube/config" >> $GITHUB_ENV
55
56   - name: Debug kubectl config
57     run: |
58       echo "Verificando configuración de kubectl..."
59       kubectl config view
60       kubectl config current-context
61       kubectl get nodes || { echo "Error: Kubernetes no está accesible"; exit 1; }
62
63   - name: Deploy Flask App
64     run: |
65       kubectl apply -f k8s/deployment.yaml || { echo "Error en deployment"; exit 1; }
66       kubectl apply -f k8s/service.yaml || { echo "Error en service"; exit 1; }
67
68   - name: Deploy Grafana with Helm
69     run: |
70       helm upgrade --install grafana grafana \
71         --repo https://grafana.github.io/helm-charts \
72         --namespace monitoring --create-namespace \
73         -f helm/grafana/values.yaml || { echo "Error en despliegue de Grafana"; exit 1; }
74
75   - name: Deploy Prometheus with Helm
76     run: |
77       helm upgrade --install prometheus prometheus \
78         --repo https://prometheus-community.github.io/helm-charts \
79         --namespace monitoring --create-namespace \
80         -f helm/prometheus/values.yaml || { echo "Error en despliegue de Prometheus"; exit 1; }
81
82   - name: Create logging namespace
83     run: |
84       kubectl create namespace logging || echo "Namespace 'logging' ya existe"
85
86   - name: Deploy ELK Stack
87     run: |
88       kubectl apply -f k8s/elk/deployment.yaml -n logging || { echo "Error en Elasticsearch"; exit 1; }
89       kubectl apply -f k8s/elk/service.yaml -n logging
90       kubectl apply -f k8s/elk/elasticsearch-pvc.yaml -n logging
91       kubectl apply -f k8s/elk/deployment-kibana.yaml -n logging || { echo "Error en Kibana"; exit 1; }
92       kubectl apply -f k8s/elk/service-kibana.yaml -n logging
93       kubectl apply -f k8s/elk/deployment-logstash.yaml -n logging || { echo "Error en Logstash"; exit 1; }
94       kubectl apply -f k8s/elk/service-logstash.yaml -n logging
95
96   - name: Restart Flask deployment
97     run: kubectl rollout restart deployment flask-tienda

```

### 7.1.1 Explicación del workflow de GitHub Actions

```
on:  
  push:  
    branches:  
      - main  
  pull_request:  
    branches:  
      - main
```

El **workflow** se ejecuta automáticamente cuando se realiza un *push* o un *pull request* hacia la rama main. Esto permite asegurar que los cambios principales del proyecto se validan y despliegan de manera automática.

```
jobs:  
  build:  
    runs-on: self-hosted
```

Este será el primer *job* se llamará *build* y se ejecuta en un *runner self-hosted*, es decir, un servidor controlado por el usuario donde está instalado el agente de **GitHub Actions**.

```
steps:  
  - name: Checkout code  
    uses: actions/checkout@v2
```

Descarga el contenido del repositorio en el *runner* para poder trabajar con el código fuente.

```
- name: Log in to GitHub Container Registry
  uses: docker/login-action@v2
  with:
    registry: ghcr.io
    username: ${ github.actor }
    password: ${ secrets.GHCR_TOKEN }
```

Autentica el *runner* contra el registro de contenedores de **GitHub** usando un token secreto (**GHCR\_TOKEN**) para poder subir la imagen

generada.

```
- name: Build Docker image
  run: docker build -t ghcr.io/pederysky/tfg_1/flask_tienda:latest .
```

Crea una imagen **Docker** a partir del **Dockerfile** del proyecto y la etiqueta con la ruta del registro.

```
- name: Push Docker image to GHCR
  run: docker push ghcr.io/pederysky/tfg_1/flask_tienda:latest
```

Publica la imagen **Docker** en **GHCR** (GitHub Container Registry) para que esté disponible en el clúster de **Kubernetes** al momento del despliegue.

```
deploy:
  runs-on: self-hosted
  needs: build
```

Este segundo *job* depende del anterior (*needs: build*), por lo que solo se ejecutará si la construcción y subida de la imagen han sido exitosas.

```
steps:
  - name: Install kubectl
    run: |
      curl -LO "https://dl.k8s.io/release/"
      chmod +x kubectl
      sudo mv kubectl /usr/local/bin/
```

Descarga e instala la herramienta de línea de comandos **kubectl**, necesaria para gestionar el clúster de **Kubernetes** desde el *runner*.

```
- name: Install Helm
  run: |
    curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
    chmod +x get_helm.sh
    ./get_helm.sh
    echo "export PATH=/usr/local/bin:\$PATH"
```

Instala el gestor de paquetes **Helm**, utilizado para desplegar aplicaciones como **Grafana** y **Prometheus** de forma sencilla.

```
- name: Configure kubeconfig
  run: |
    mkdir -p $HOME/.kube
    echo "${{ secrets.KUBE_CONFIG }}" > $HOME/.kube/config
    echo "KUBECONFIG=$HOME/.kube/config" >> $GITHUB_ENV
```

Se crea el archivo de configuración (*kubeconfig*) necesario para que **kubectl** pueda interactuar con el clúster. Este archivo se genera a partir del *secret* **KUBE\_CONFIG** almacenado en **GitHub Secrets**.

```
- name: Debug kubectl config
  run: |
    echo "Verificando configuración de kubectl..."
    kubectl config view
    kubectl config current-context
    kubectl get nodes || { echo "Error: Kubernetes no está accesible"; exit 1; }
```

Muestra información del contexto actual de **Kubernetes** y verifica que el clúster es accesible correctamente desde el *runner*.

```
- name: Deploy Flask App
  run: |
    kubectl apply -f k8s/deployment.yaml || { echo "Error en deployment"; exit 1; }
    kubectl apply -f k8s/service.yaml || { echo "Error en service"; exit 1; }
```

**Despliegue de la aplicación Flask:** Se aplica la configuración de **Kubernetes** para la aplicación (*deployment.yaml* y *service.yaml*), lo que crea o actualiza los recursos necesarios en el clúster.

```
- name: Deploy Grafana with Helm
  run: |
    helm upgrade --install grafana grafana \
      --repo https://grafana.github.io/helm-charts \
      --namespace monitoring --create-namespace \
      -f helm/grafana/values.yaml || { echo "Error en despliegue de Grafana"; exit 1; }
```

Utiliza **Helm** para instalar o actualizar **Grafana** en el *namespace monitoring*, aplicando la configuración definida en *helm/grafana/values.yaml*.



```
- name: Deploy Prometheus with Helm
run: |
  helm upgrade --install prometheus prometheus \
    --repo https://prometheus-community.github.io/helm-charts \
    --namespace monitoring --create-namespace \
    -f helm/prometheus/values.yaml || { echo "Error en despliegue de Prometheus"; exit 1; }
```

Al igual que con **Grafana**, se instala **Prometheus** con **Helm**, también en el *namespace monitoring*, aplicando la configuración definida en *helm/prometheus/values.yaml*.

```
- name: Create logging namespace
run: |
  kubectl create namespace logging || echo "Namespace 'logging' ya existe"
```

Crea un nuevo *namespace* llamado *logging*, donde se desplegará el stack **ELK**.

```
- name: Deploy ELK Stack
run: |
  kubectl apply -f k8s/elk/deployment.yaml -n logging || { echo "Error en Elasticsearch"; exit 1; }
  kubectl apply -f k8s/elk/service.yaml -n logging
  kubectl apply -f k8s/elk/elasticsearch-pvc.yaml -n logging
  kubectl apply -f k8s/elk/deployment-kibana.yaml -n logging || { echo "Error en Kibana"; exit 1; }
  kubectl apply -f k8s/elk/service-kibana.yaml -n logging
  kubectl apply -f k8s/elk/deployment-logstash.yaml -n logging || { echo "Error en Logstash"; exit 1; }
  kubectl apply -f k8s/elk/service-logstash.yaml -n logging
```

Aplica los manifiestos necesarios para desplegar los tres componentes principales del stack ELK:

- Elasticsearch: almacén central de logs.
- Logstash: ingesta y transformación de logs.
- Kibana: interfaz para visualización.

```
- name: Restart Flask deployment
run: kubectl rollout restart deployment flask-tienda
```

Fuerza un reinicio del *deployment* de la aplicación Flask para asegurarse de que se usa la última imagen subida al registro.

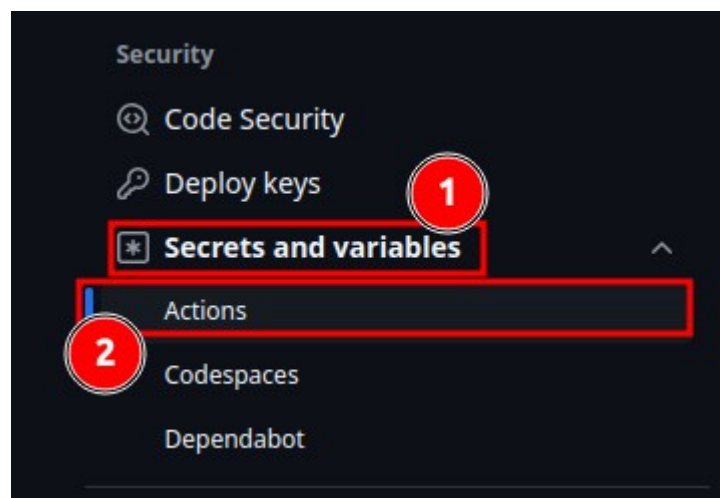
## 7.2 Creación y uso de secrets

En **GitHub**, los *secrets* son valores confidenciales, como contraseñas, *tokens* de acceso, claves API o cualquier dato sensible que no debería incluirse directamente en el código fuente ni en los *workflows* de CI/CD. Estos secretos se almacenan de forma segura y se pueden utilizar dentro de los **GitHub Actions** para automatizar despliegues o integraciones, manteniendo la privacidad de esa información.

Su objetivo es proteger credenciales y datos críticos durante la ejecución de *workflows*, evitando que queden expuestos en los repositorios.

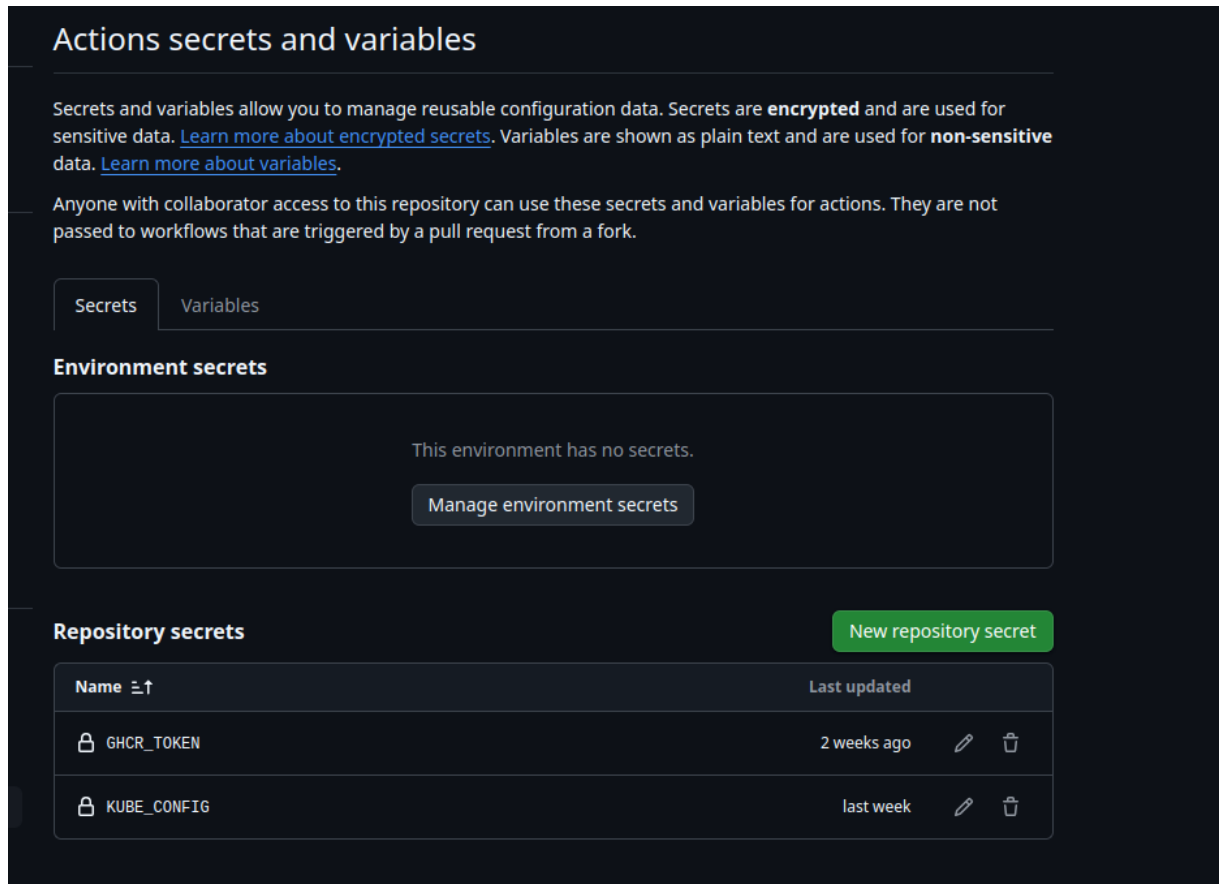
### 7.2.1 Secrets de un repositorio en GitHub

Para crear un *secret* en un repositorio de **GitHub**, debemos ir al apartado de configuración (settings), en el menú lateral de la izquierda vamos a seguridad (security), secretos y variables (secrets and variables) y a *actions*



<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

Aquí podemos crear tanto variables como *secrets*, es aconsejable nombrarlas con un nombre descriptivo



The screenshot shows the 'Actions secrets and variables' page in GitHub. It has two tabs: 'Secrets' (selected) and 'Variables'. Under 'Secrets', there are two sections: 'Environment secrets' and 'Repository secrets'. The 'Environment secrets' section shows a message 'This environment has no secrets.' with a 'Manage environment secrets' button. The 'Repository secrets' section has a 'New repository secret' button and a table of existing secrets.

Name	Last updated
GHCR_TOKEN	2 weeks ago
KUBE_CONFIG	last week

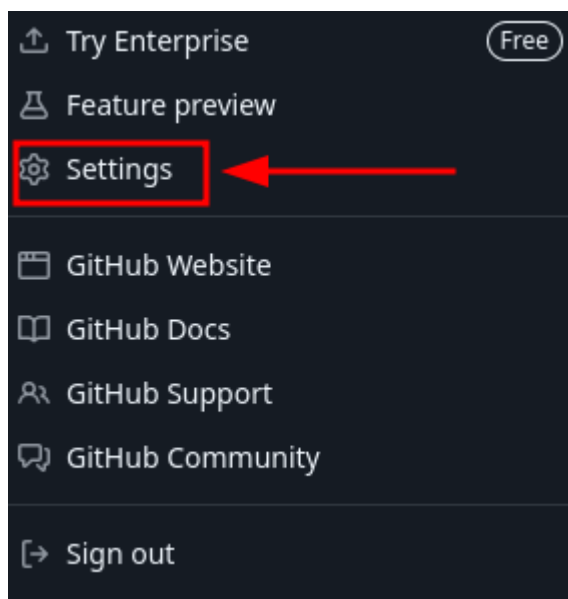
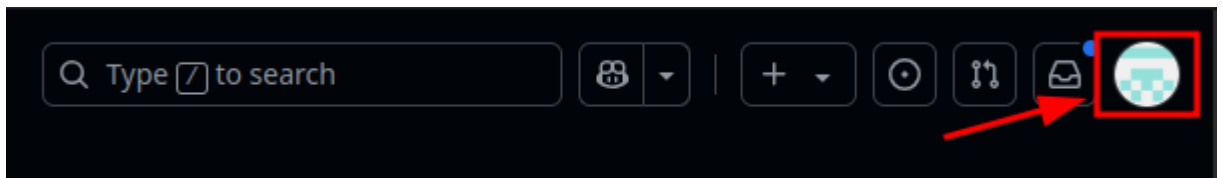
Una vez creado, ese secret estará disponible para su uso dentro de los workflows de GitHub Actions mediante la sintaxis `${{ secrets.NOMBRE_DEL_SECRET }}`.

Para este proyecto se han creado 2 *secrets*, **GHCR\_TOKEN** y **KUBE\_CONFIG**

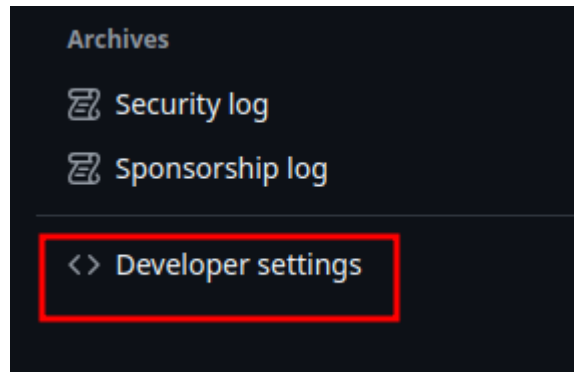
### 7.2.2 GHCR\_TOKEN

Este *secret* almacena un *token* de acceso al registro de contenedores de **GitHub** (**GitHub Container Registry**). Sirve para que un *workflow* pueda subir o descargar imágenes de **Docker** de forma segura desde el **GHCR**, sin necesidad de exponer el usuario y la contraseña.

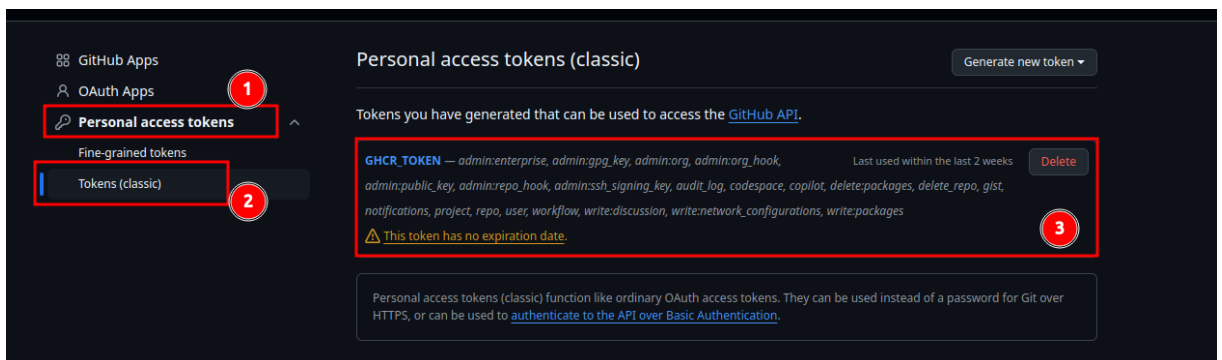
Para crear este secret, vamos a la foto de perfil en la esquina superior derecha y vamos a **Settings**.



En el menú lateral izquierdo, busca y haz clic en **Developer settings** (abajo del todo).



Dentro de **Developer settings**, selecciona **Personal access tokens > Tokens (classic)** (o **Fine-grained tokens** si lo prefieres más restrictivo) y haz clic en **Generate new token**.



Podemos configurar dicho **token** como deseemos, desde la duración de su validez hasta que permisos otorga y si se aplica a uno, varios o todos los repositorios.

Este **token** solo será visible al momento de crearlo, así que debemos de copiarlo en algún lugar seguro. Si se pierde, se puede generar un valor nuevo sin necesidad de crear otro **token** pero habría que actualizarlo también en todos los repositorios que haya sido utilizado.

Tiene este aspecto: ghp\_AyR7CLqCOaAWCxGUjZ6GWqxfXnErP90EviWj

Una vez creado el token, debemos añadirlo en la máquina donde se va a correr el workflow con el siguiente comandos

```

pedro@Ubuntu22: ~
pedro@Ubuntu22:~$ kubectl create secret docker-registry github-registry-secret \
--docker-server=ghcr.io \
--docker-username=<GITHUB_USER> \
--docker-password=<GHCR_TOKEN> \
--docker-email=<EMAIL_GITHUB>

```

Podemos comprobar si está añadido con el comando

```

pedro@Ubuntu22: ~
pedro@Ubuntu22:~$ kubectl get secrets github-registry-secret
NAME                                TYPE                                DATA  AGE
github-registry-secret             kubernetes.io/dockerconfigjson    1      8d
pedro@Ubuntu22:~$

```

### 7.2.3 KUBE\_CONFIG

Este *secret* contiene el archivo de configuración de **Kubernetes** (kubeconfig) que permite al *workflow* conectarse al clúster de **Kubernetes**. Gracias a este fichero, **GitHub Actions** puede desplegar aplicaciones o hacer cambios en un entorno **Kubernetes** sin tener que compartir abiertamente las credenciales del clúster.

En este secret debemos pegar la configuración de **Kubernetes**, podemos obtenerla en el directorio `~/.kube/config`

Tiene este aspecto:

```

pedro@Ubuntu22:~$ cat ~/.kube/config
apiVersion: v1
clusters:
- cluster:
  certificate-authority: /home/pedro/.minikube/ca.crt
  extensions:
  - extension:
    last-update: Tue, 15 Apr 2025 19:39:15 CEST
    provider: minikube.sigs.k8s.io
    version: v1.35.0
    name: cluster_info
  server: https://192.168.49.2:8443
  name: minikube
contexts:
- context:
  cluster: minikube
  extensions:
  - extension:
    last-update: Tue, 15 Apr 2025 19:39:15 CEST
    provider: minikube.sigs.k8s.io
    version: v1.35.0
    name: context_info
  namespace: default
  user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/pedro/.minikube/profiles/minikube/client.crt
    client-key: /home/pedro/.minikube/profiles/minikube/client.key
pedro@Ubuntu22:~$

```



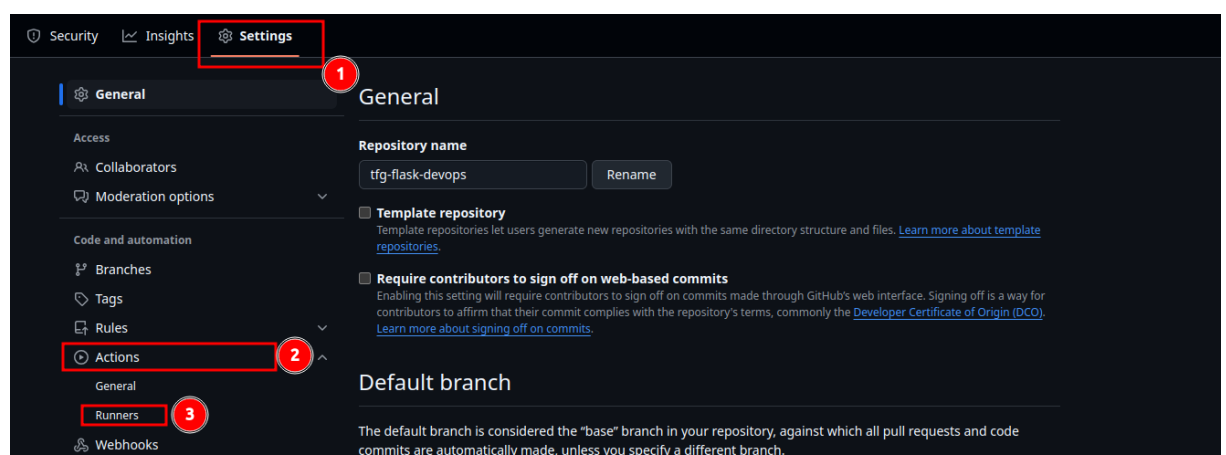
### 7.3 Crear un self-hosted runner

Un **self-hosted runner** es un servidor o equipo que tú mismo gestionas y que se encarga de ejecutar los *workflows* de **GitHub Actions**. A diferencia de los *runners* en la nube que ofrece **GitHub** por defecto, este lo tienes bajo tu control, ya sea en tu ordenador personal, en un servidor local o en la nube.

- Permite ejecutar *jobs* de **GitHub Actions** en tu propia máquina o infraestructura.
- Evita las limitaciones de tiempo, espacio o recursos que tienen los *runners* gratuitos en la nube.
- Puedes personalizar el entorno de ejecución con las herramientas, dependencias y configuraciones que necesites.
- Ideal si quieres reducir la dependencia de los servidores de **GitHub** o trabajar en entornos locales.

Para crear el *runner*, debemos seguir los pasos que nos indica el propio **GitHub**.

Se crea en **Settings > Actions > Runners > New self-hosted runner**.



Se puede crear tanto para Linux como Windows o macOS

## Runners / Add new self-hosted runner · pederysky/tfg-flask-devops

Adding a self-hosted runner requires that you download, configure, and execute the GitHub Actions Runner. If you do not already have an existing volume licensing agreement for your GitHub purchases, by downloading and configuring the GitHub Actions Runner, you agree to the [GitHub Customer Agreement](#).

**Runner image**

☐ macOS
 ☒ Linux
 ☐ Windows

**Architecture**

x64

**Download**

```
# Create a folder
$ mkdir actions-runner && cd actions-runner

# Download the latest runner package
$ curl -o actions-runner-linux-x64-2.323.0.tar.gz -L
https://github.com/actions/runner/releases/download/v2.323.0/actions-runner-linux-x64-2.323.0.tar.gz

# Optional: Validate the hash
$ echo "0dbc9bf5a58620fc52cb6cc0448abcca964a8d74b5f39773b7afcad9ab691e19 actions-runner-linux-x64-
2.323.0.tar.gz" | shasum -a 256 -c

# Extract the installer
$ tar xzf ./actions-runner-linux-x64-2.323.0.tar.gz
```

**Configure**

```
# Create the runner and start the configuration experience
$ ./config.sh --url https://github.com/pederysky/tfg-flask-devops --token
BCTH3MGXUBLQHKNPYY03C3IBDCEC

# Last step, run it!
$ ./run.sh
```

**Using your self-hosted runner**

```
# Use this YAML in your workflow file for each job
runs-on: self-hosted
```

For additional details about configuring, running, or shutting down the runner, please check out our [product docs](#).

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## **7.4 Consideraciones de seguridad y buenas prácticas**

Durante la implementación se han seguido buenas prácticas para garantizar la seguridad del *pipeline* de CI/CD. Los *secrets* se almacenan en **GitHub Secrets**, evitando incluir credenciales sensibles en el código fuente. Además, se ha utilizado un **self-hosted runner** controlado por el usuario, permitiendo mayor aislamiento y personalización del entorno. Se recomienda rotar periódicamente los *tokens* de acceso y restringir los permisos al mínimo necesario para cada *token* y servicio.

## **7.5 Implantaciones de código realizadas**

En el repositorio de **GitHub** del proyecto se puede encontrar todo el código y las configuraciones que se han utilizado. Desde allí se puede consultar la aplicación completa, la configuración de los despliegues en **Kubernetes** y el *workflow* de **GitHub Actions** que automatiza todo el proceso.

Se puede acceder al repositorio desde el siguiente enlace:

<https://github.com/pederysky/tfg-flask-devops>

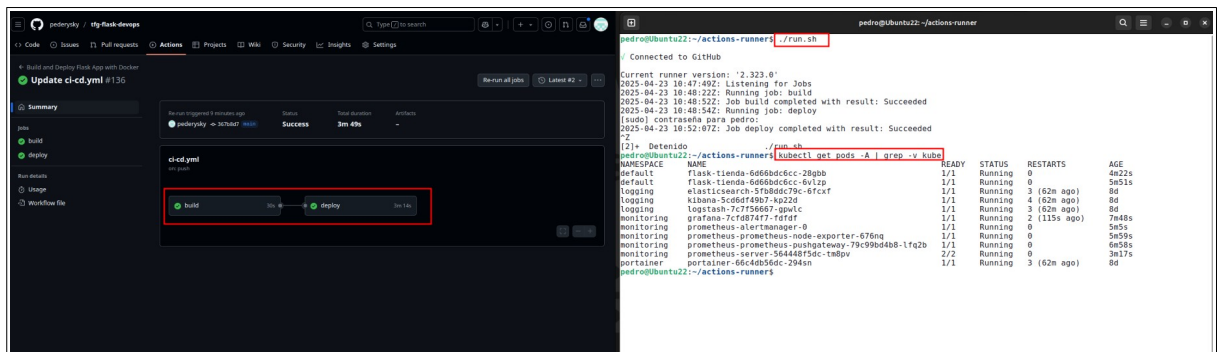
## 8 PRUEBAS

### 8.1 Pruebas de funcionamiento

Para probar el *workflow* que hemos configurado en el proyecto, vamos a realizar dos pasos fundamentales:

- Ejecutamos en una terminal de la máquina que *hostea* el clúster de **Kubernetes** el *script* que comenzará a ejecutar el *runner*, lo que permitirá que **GitHub** lo reconozca y lo utilice cuando se active un *workflow*.
- Con el *runner* corriendo en la máquina, desde **Github** ejecutamos el *workflow* o bien hacemos un *commit* en el repositorio.

Si todo sale bien durante la ejecución del *workflow*, significa que el CI/CD *pipeline* ha funcionado correctamente, y el flujo de trabajo ha sido completado sin errores.



# IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES

ASIR

Update ci-cd.yml #136

Summary

Jobs

- build
- deploy

Run details

- Usage
- Workflow file

Re-run triggered 22 minutes ago

Status

- pederysky → 367b8d7 main

Success

Total duration

- 3m 49s

Artifacts

-

ci-cd.yml

on: push

build30s

→

deploy3m 14s

pedro@Ubuntu22: ~/actions-runner

pedro@Ubuntu22:~/actions-runner\$ ./run.sh

✓ Connected to GitHub

Current runner version: '2.323.0'

2025-04-23 10:47:49Z: Listening for Jobs

2025-04-23 10:48:22Z: Running job: build

2025-04-23 10:48:52Z: Job build completed with result: Succeeded

2025-04-23 10:48:54Z: Running job: deploy

[sudo] contraseña para pedro:

2025-04-23 10:52:07Z: Job deploy completed with result: Succeeded

^Z

[2]+ Detenido ./run.sh

pedro@Ubuntu22:~/actions-runner\$ kubectl get pods -A | grep -v kube

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	flask-tienda-6d66bdc6cc-28gbb	1/1	Running	0	4m22s
default	flask-tienda-6d66bdc6cc-6vlzp	1/1	Running	0	5m51s
logging	elasticsearch-5fb8ddc79c-6fcxf	1/1	Running	3 (62m ago)	8d
logging	kibana-5cd6df49b7-kp22d	1/1	Running	4 (62m ago)	8d
logging	logstash-7c7f56667-gpwl	1/1	Running	3 (62m ago)	8d
monitoring	grafana-7cfd874f7-fdfdf	1/1	Running	2 (115s ago)	7m48s
monitoring	prometheus-alertmanager-0	1/1	Running	0	5m5s
monitoring	prometheus-prometheus-node-exporter-676nq	1/1	Running	0	5m59s
monitoring	prometheus-pushgateway-79c99bd4b8-lfq2b	1/1	Running	0	6m58s
monitoring	prometheus-server-564448f5dc-tm8pv	2/2	Running	0	3m17s
portainer	portainer-66c4db56dc-294sn	1/1	Running	3 (62m ago)	8d

pedro@Ubuntu22:~/actions-runner\$

IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

## 8.2 Pruebas de mantenimiento

Con el objetivo de garantizar que la infraestructura desplegada siga funcionando correctamente a lo largo del tiempo, se ha definido un conjunto de pruebas de mantenimiento preventivo que se recomienda realizar de forma periódica. Estas pruebas permiten detectar posibles errores, comprobar el estado de los servicios desplegados y asegurarse de que tanto la aplicación como las herramientas de monitorización y CI/CD siguen operativas.

### 8.2.1 Comprobación del estado de los pods en Kubernetes

Se revisa que todos los **pods** estén en estado *Running* y que no haya reinicios inesperados. Esto permite identificar posibles fallos en el despliegue o en la ejecución de los contenedores. Existen otros estados como *Failed*, que nos indica que algo ha fallado o *Pending*, que nos indica que el pod está esperando a que se creen/descarguen las imágenes o se asigne a un nodo.

Comando utilizado: `kubectl get pods -A`

Este comando nos lista todos los **pods** en todos los **namespaces**.

Si queremos comprobar los **pods** en un **namespace** concreto (por ejemplo, *monitoring*): `kubectl get pods -n monitoring`

Se revisa que todos los servicios estén expuestos correctamente y funcionando.

Comando general: `kubectl get services -A`

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

### 8.2.2 *Revisión de logs de los contenedores*

Se analizan los **logs** de los servicios más críticos para comprobar que no existan errores o mensajes inusuales.

Comando utilizado: `kubectl logs <nombre-del-pod>`

O para un **namespace** concreto (por ejemplo, *logging*):

`kubectl get services -n logging`

### 8.2.3 *Simulación de un despliegue automático*

Para comprobar que el **pipeline CI/CD** sigue operativo, se realiza un pequeño cambio en el repositorio (por ejemplo, modificar el mensaje en la página de inicio) y se verifica que el despliegue se realiza correctamente mediante **GitHub Actions** y **Kubernetes**.

Comando para comprobar el rollout del deployment o realizar un despliegue progresivo:

`kubectl rollout status deployment <nombre-del-deployment> -n <namespace>`

Se puede volver a una versión anterior con el parámetro **undo** o ver el historial de despliegues con **history**:

`kubectl rollout undo deployment <nombre-deployment>`

`kubectl rollout history deployment <nombre-deployment>`

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

### 8.2.4 *Copia de seguridad de la base de datos*

Se realiza un **backup** manual o automático de la base de datos utilizada por la aplicación (por ejemplo, **MySQL** o **PostgreSQL**) y se prueba la restauración en un entorno de pruebas para asegurarse de que el procedimiento funciona correctamente.

Comando ejemplo para exportar una base de datos MySQL en un pod:

```
kubectl exec -it <mysql-pod> -n <namespace> -- mysqldump -u root -p <nombre_bd>
> backup.sql
```

### 8.2.5 *Control del almacenamiento de logs*

Se revisa que los **logs** almacenados por **ELK** no estén ocupando demasiado espacio. En caso necesario, se podría ajustar la retención o eliminar **logs** antiguos.

Comando para consultar el uso de almacenamiento por pods:

```
kubectl describe pvc -A
```

Lo ideal sería automatizar estas tareas con un **script** o otro **workflow** que nos vaya mostrando por pantalla cada salida.



## 9 ANEXOS



<https://github.com/pederysky/tfg-flask-devops.git>

Despliegue de la aplicación:

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: flask-tienda
5  spec:
6    replicas: 2
7    selector:
8      matchLabels:
9        app: flask-tienda
10   template:
11     metadata:
12       labels:
13         app: flask-tienda
14     spec:
15       containers:
16         - name: flask-tienda
17           image: ghcr.io/pederysky/tfg-flask-devops/flask_tienda:latest
18           ports:
19             - containerPort: 5000
20       imagePullSecrets:
21         - name: github-registry-secret

```

Este fichero **YAML** sirve para crear un **Deployment** en **Kubernetes**, que básicamente es el encargado de levantar varios pods iguales y mantenerlos funcionando.

- **apiVersion: apps/v1:** Es la versión de la API de **Kubernetes** que estoy usando para este recurso. En este caso **apps/v1**, que es la recomendada para los despliegues.
- **kind: Deployment:** Aquí le indico a **Kubernetes** que el recurso que voy a crear es un **Deployment**, que se encarga de gestionar réplicas de **pods**.

IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

- **metadata:** Dentro de esto, pongo el nombre que va a tener mi **Deployment**, que en este caso es **flask-tienda**.
- **replicas: 2:** Esto es para decir cuántos **Pods** quiero que estén siempre funcionando. He puesto 2 para tener alta disponibilidad, así si uno se cae, el otro sigue dando servicio.
- **selector y matchLabels:** Esto es para que el **Deployment** sepa qué **Pods** tiene que controlar. En este caso, los que lleven la etiqueta **app: flask-tienda**.
- **template:** Aquí defino cómo van a ser los **Pods** que se van a crear. Dentro de **metadata** le pongo la etiqueta **app: flask-tienda** y dentro de **spec** le digo qué contenedor tiene que levantar.
- **containers:** Se define como van a ser los contenedores se van a crear
- **imagePullSecrets:** Esto es para que **Kubernetes** pueda acceder a mi imagen privada de **Docker** en **GitHub** con el **secret** mencionado en el proyecto.

En resumen, este **Deployment** se encarga de levantar **2 Pods** iguales de mi aplicación Flask, que usan una imagen privada en **GitHub**. Están etiquetados con **app: flask-tienda** y exponen el **puerto 5000**. Además, he añadido un secreto para que pueda descargar la imagen.

Servicio de la aplicación:

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: flask-tienda-service
5  spec:
6    selector:
7      app: flask-tienda
8    ports:
9      - protocol: TCP
10      port: 80      # Puerto interno del contenedor
11      targetPort: 5000 # Puerto donde Flask está escuchando
12      nodePort: 30001 # Puerto en el nodo
13    type: NodePort

```

Este fichero **YAML** sirve para crear un **Service** en **Kubernetes**, que es lo que permite que los **pods** sean accesibles desde fuera del clúster o desde otros servicios.

- **apiVersion: v1:** Aquí se indica la versión de la API que se usa para crear este recurso, en este caso v1, que es la adecuada para los servicios.
- **kind: Service:** Esto define que el recurso que voy a crear es un Service.
- **metadata:** Le pongo nombre al **Service**, que en este caso es **flask-tienda-service**.
- **spec: selector:** Aquí se define a qué **pods** va a dirigir el tráfico este servicio. En este caso, selecciona todos los **pods** que tengan la etiqueta **app: flask-tienda**. Esto hace que solo esos **pods** reciban las peticiones.
- **ports:** Aquí se configuran los puertos por los que se va a comunicar el servicio y el protocolo.

IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES	ASIR
---	------

- **type: NodePort:** Aquí defino que el servicio va a ser de tipo **NodePort**, lo que significa que **Kubernetes** abre un puerto en cada nodo del clúster para poder acceder desde fuera.

Este **Service** se encarga de exponer los **pods** de **flask-tienda** para que puedan recibir peticiones desde fuera de **Kubernetes**, a través del **puerto 30001**. Redirige ese tráfico al puerto 80 del servicio y de ahí al 5000 donde está escuchando Flask en los contenedores.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## 10 CONCLUSIONES

Desarrollar este sistema CI/CD para una aplicación basada en contenedores ha sido todo un reto que me ha permitido poner en práctica lo aprendido durante el ciclo de ASIR y, además, sumergirme en el mundo de **DevOps** con tecnologías que están en pleno auge.

Lo que buscaba con este proyecto era básicamente automatizar todo el proceso de desarrollo y despliegue, para acabar con esas tareas repetitivas que todos odiamos y reducir los errores que cometemos los humanos cuando hacemos cosas manualmente. Y puedo decir que lo he conseguido: combinando **GitHub Actions**, **Docker** y **Kubernetes** he creado un *pipeline* que se encarga de todo, desde compilar el código hasta ponerlo en producción, pasando por las pruebas necesarias.

Durante el camino me he encontrado con varios desafíos técnicos. Configurar correctamente los *workflows* de **GitHub Actions** no fue precisamente un paseo, y aprender a gestionar las credenciales de forma segura con *secrets* o a orquestrar contenedores con **Kubernetes** me ha llevado horas de investigación, pruebas y, por qué no decirlo, algún que otro momento de frustración. Pero al final, toda esta experiencia ha valido la pena y ha enriquecido enormemente mi formación.

Este proyecto me ha dejado claro por qué las metodologías **DevOps** están revolucionando el desarrollo de software. La velocidad, calidad y eficiencia que aportan son increíbles. Y lo mejor es que todo lo he implementado con herramientas de código abierto, lo que demuestra que se pueden crear soluciones profesionales sin gastarse un dineral en licencias, algo perfecto para pequeñas empresas o entornos educativos como el nuestro.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

A nivel personal, este trabajo me ha servido para reforzar lo que ya sabía sobre administración de sistemas y, además, me ha abierto las puertas al mundo de la automatización y la infraestructura como código, que están súper demandados en el mercado laboral. La experiencia que he ganado con **Docker**, **Kubernetes** y **GitHub Actions** seguro que me será muy útil en mi futuro profesional.

<b>IMPLANTACIÓN DE UN SISTEMA CI/CD PARA UNA APLICACIÓN BASADA EN CONTENEDORES</b>	<b>ASIR</b>
--	-------------

## 11 FUENTES

Legislación:

ASIR

Enseñanzas mínimas: [R.D. 1629/2009 \(B.O.E. 18/11/2009\)](#)

[http://www.madrid.org/fp/ense\\_fp/catalogo\\_LOE/pdf/IFCS01/titulo/  
RD20091629\\_TS\\_Admon\\_Sistemas\\_Informaticos\\_en\\_Red.pdf](http://www.madrid.org/fp/ense_fp/catalogo_LOE/pdf/IFCS01/titulo/RD20091629_TS_Admon_Sistemas_Informaticos_en_Red.pdf)

Currículo: [D 12/2010 \(B.O.C.M. 15/04/2010\)](#)

[http://www.madrid.org/fp/ense\\_fp/catalogo\\_LOE/pdf/IFCS01/curriculo/D20100012\\_Ad-  
ministracion\\_SistemasInformaticos.pdf](http://www.madrid.org/fp/ense_fp/catalogo_LOE/pdf/IFCS01/curriculo/D20100012_Administracion_SistemasInformaticos.pdf)

Definición de procedimientos de control y evaluación:

- <http://www.xperta.es/es/descripcion.asp>
- <http://www.xperta.es/es/aquienvadirigido.asp>
- <http://churriwifi.wordpress.com/2010/04/10/gestion-de-incidencias/>
- [http://es.wikipedia.org/wiki/Control\\_de\\_versiones](http://es.wikipedia.org/wiki/Control_de_versiones)

### 11.1 Páginas web y bibliografía consultadas

- **GitHub:** <https://docs.github.com/es/actions>
- **Docker:** <https://docs.docker.com/>
- **Kubernetes:** <https://kubernetes.io/docs/>
- **Tutoriales de Youtube**
  - [https://www.youtube.com/watch?v=IQNxRSJ2ZVQ&t=9s&ab\\_channel=DevOpsHint](https://www.youtube.com/watch?v=IQNxRSJ2ZVQ&t=9s&ab_channel=DevOpsHint)
  - [https://www.youtube.com/watch?v=-8haKKGc55Y&t=1140s&ab\\_channel=Platzi](https://www.youtube.com/watch?v=-8haKKGc55Y&t=1140s&ab_channel=Platzi)
  - [https://www.youtube.com/watch?v=ID0t-UgKfEo&ab\\_channel=AntonPutra](https://www.youtube.com/watch?v=ID0t-UgKfEo&ab_channel=AntonPutra)
  - [https://www.youtube.com/watch?v=DcOBcpOA7W4&ab\\_channel=PeladoNerd](https://www.youtube.com/watch?v=DcOBcpOA7W4&ab_channel=PeladoNerd)