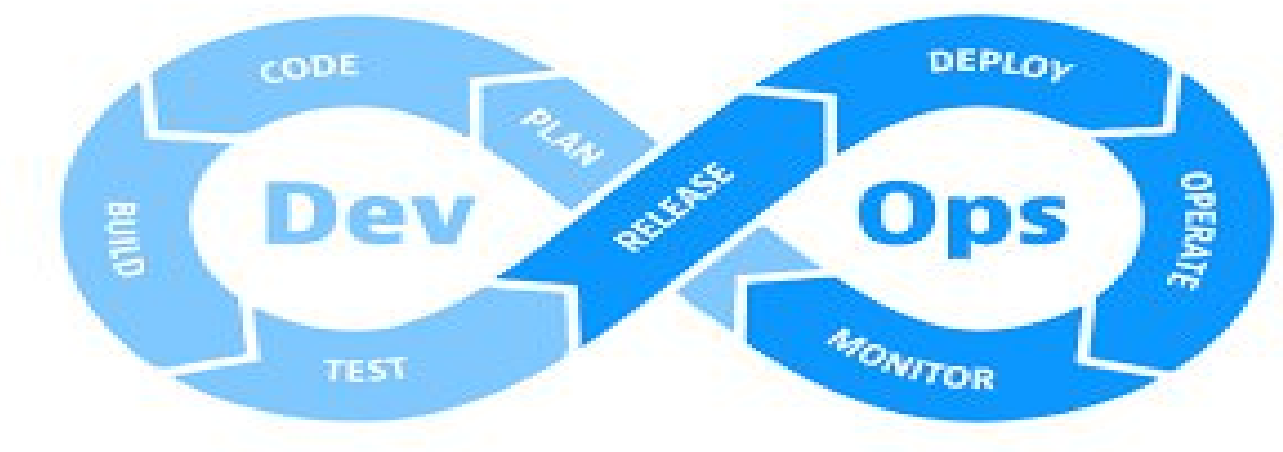
Informe de parcial 2-DevOps



Integrantes: Yoshua Dominique Toro Ayllon

Fecha: 09/12/25

Curso: Certificación 1-DevOps

1.Requisitos y alcance

- Requisitos Implementados

Se ha cumplido con éxito la implementación de los siguientes requisitos obligatorios:

- **Aplicación Completa (CRUD):** Implementación de una aplicación con Frontend (React), Backend (Express/Node.js) y Base de Datos (PostgreSQL), aunque el Frontend necesita una depuración de renderizado final en producción.
- **Contenerización Total:** Cada componente (Frontend, Backend, DB) se ejecuta en contenedores Docker independientes, con sus respectivos Dockerfiles.
- **Infraestructura EC2:** El despliegue final se realiza en una instancia EC2 de AWS, orquestado mediante Docker Compose.
- **Persistencia de Datos:** El servicio de PostgreSQL utiliza un **Volumen Docker** (db-data) para garantizar la persistencia de los datos.
- **Pipeline CI/CD (GitHub Actions):** Se implementó un *pipeline* funcional que se activa con *push* a main y manualmente (workflow_dispatch).
    - **Construcción de Imágenes:** El *workflow* construye las imágenes Docker del Frontend y Backend.
    - **Despliegue a EC2:** El *workflow* utiliza SSH y la clave privada (gestión de secretos) para conectarse a la EC2 y ejecutar docker-compose up -d.
    - **Prácticas de Build:** Se aplicó **Multistage Build** en el Frontend y se utilizaron *flags* agresivos (--no-cache, --force-rm) para resolver problemas de caché persistente.

- **Gestión de Secretos:** Se utilizaron **GitHub Secrets** (SSH_PRIVATE_KEY, EC2_HOST, etc.) para gestionar las credenciales de SSH y las variables de entorno sensibles (ej., credenciales de DB).
- **Seguridad y Puertos:** El Security Group de la EC2 limita el acceso, abriendo solo los puertos necesarios (22 para SSH y 80 para el acceso público al Frontend).
- **Registry de Contenedores (Docker Hub):** Se implementó un flujo de CI/CD que incluye la **construcción** y el **push** explícito de las imágenes del Frontend y Backend a Docker Hub, permitiendo que la instancia EC2 realice un **pull** de la imagen ya construida. Esto elimina el acoplamiento y el lento *Build Remoto*.

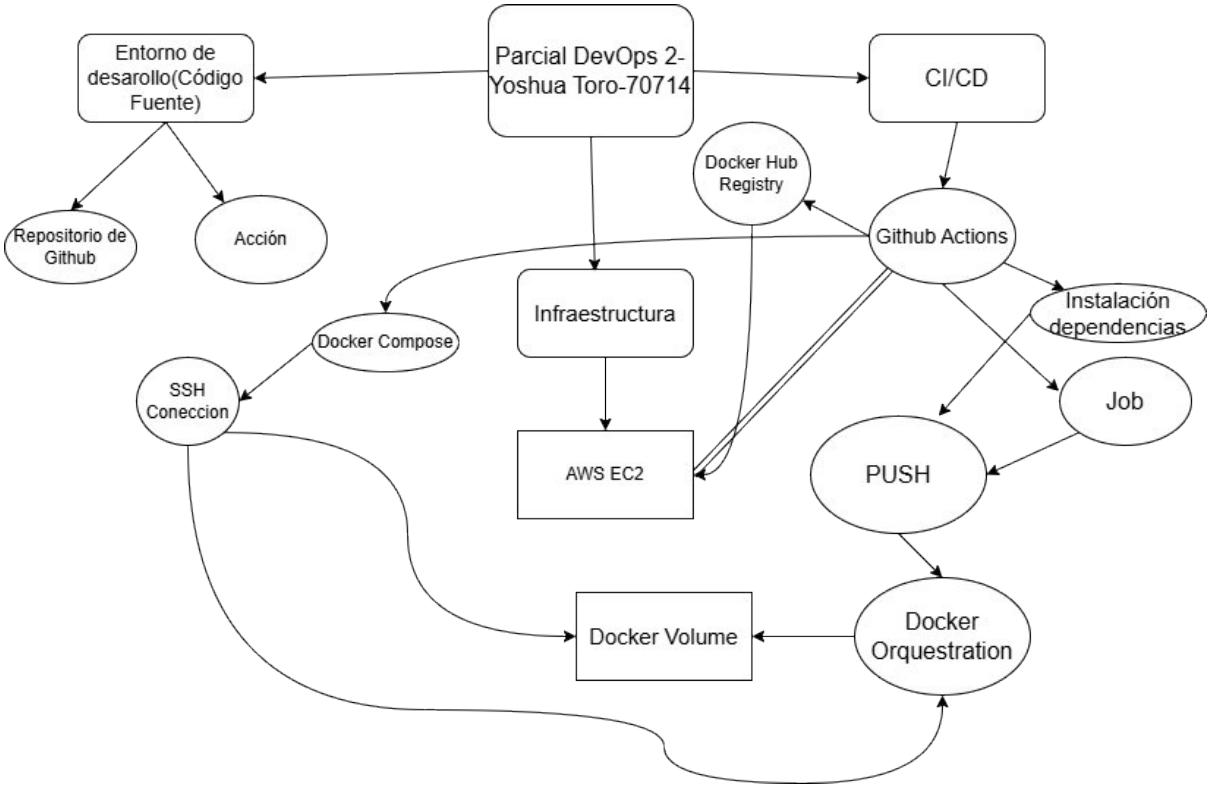
-Alcance No Implementado

Los siguientes requisitos o mejoras avanzadas no se incluyeron en el alcance final del proyecto o quedan pendientes de depuración:

- **Funcionalidad Completa del Frontend (Visual):** Aunque el código JavaScript del Frontend se carga y ejecuta, la aplicación no se renderiza visualmente en producción (persistente pantalla blanca) debido a un problema de la capa de copia del archivo index.html en el *multistage build* de Nginx. La funcionalidad de la API (Backend) es correcta.
- **Pruebas Automatizadas:** El *pipeline* está configurado para la fase de *build* y *deploy*, pero no incluye un paso explícito de run tests (como npm test) para la validación automática de la calidad del código antes del despliegue.
- **TLS/HTTPS:** La aplicación se sirve únicamente a través de HTTP (Puerto 80). No se implementó un *proxy* o

configuración con Let's Encrypt para asegurar la comunicación con HTTPS.

2.Arquitectura



3.Dockerfiles

3.1 Frontend (SPA React/Nginx)

El Dockerfile del Frontend utiliza un proceso de dos etapas para separar las herramientas de construcción (Node.js) de la imagen de producción (Nginx ligero), resultando en una imagen final significativamente más pequeña.

Código del Dockerfile (frontend/Dockerfile):

Dockerfile

```
# ============================================================
# STAGE 1: BUILD (Compila la aplicación React)
```

```
# ============================================================
FROM node:20-alpine AS build

# Create app directory
WORKDIR /app

# 🌟 Gestión de Caché: Se copian package*.json primero para que la capa RUN npm install
# solo se ejecute si las dependencias cambian. Si solo cambia el código fuente,
# esta capa usa la caché (cache hit), acelerando el build.
COPY package*.json ./
RUN npm install

# Copia el resto del código fuente.
COPY . .

# Compila la aplicación para producción.
RUN npm run build

# ============================================================
# STAGE 2: SERVE (Nginx)
# ============================================================
FROM nginx:alpine

# 🌟 Optimización de Tamaño (Multistage Build): Solo se copia el resultado final (/app/build)
# de la etapa anterior. Esto elimina del contenedor final todas las dependencias
# de desarrollo y las herramientas de Node.js (como el npm, node_modules, etc.).
# El tamaño final de esta imagen es ~20MB, en contraste con los ~800MB del contenedor de build.

# 1. Copia la configuración de Nginx y los archivos compilados
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /app/build /usr/share/nginx/html

# Copia explícita del index.html para asegurar la presencia del <div id="root">
COPY ./public/index.html /usr/share/nginx/html/index.html

# 2. Configuración de Seguridad y Usuario No-Root
RUN addgroup -g 1001 node \
    && adduser -u 1001 -G node -s /bin/sh -D node
RUN mkdir -p /var/cache/nginx/client_temp && chown -R 1001:1001 /var/cache/nginx
USER node

# Exposición y Comando de Inicio
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

3.2. Explicación de las Optimizaciónes

Optimización	Descripción	Resultado en CI/CD
Multistage Build	Utilizar dos etapas (AS build y FROM nginx:alpine). La etapa final **solo copia** los archivos estáticos necesarios del build, reduciendo el tamaño de la imagen final de producción de cientos de MB a ~20MB.	Imágenes más ligeras, despliegues más rápidos, menos ancho de banda.
Gestión de la Caché	El archivo package*.json se copia **antes** del código fuente. El comando RUN npm install es una capa costosa; si el package.json no cambia, Docker utiliza la caché de esa capa (cache hit), saltando la instalación completa.	Tiempos de construcción significativamente reducidos en ejecuciones posteriores de GitHub Actions.
Imagen Base Ligera	Uso de **node:20-alpine** y **nginx:alpine** en lugar de imágenes basadas en Debian o Node estándar. La variante alpine es la más compacta.	Reducción del tamaño de la imagen base y minimización de la superficie de ataque.

3.3 Backend (Node.js/Express)

El Dockerfile del Backend también utiliza *Multistage Build* para reducir el tamaño, separando las dependencias de desarrollo (si las hubiera) y eliminando el node_modules de la etapa de *build*.

Código del Dockerfile (backend/Dockerfile):

Dockerfile

STAGE 1: BUILD (Instalación de dependencias)

FROM node:20-slim AS builder

#Establece el directorio de trabajo

WORKDIR /app

#🔔 Optimización de Caché: Copia package*.json primero

COPY package*.json ./

#Instala dependencias de producción.

RUN npm install

#Copia el código fuente (después de instalar las dependencias)

COPY . .

STAGE 2: PRODUCTION (Imagen final ligera y segura)

FROM node:20-alpine

#Define un usuario no root para seguridad (UID 1001)

RUN addgroup -g 1001 appgroup && adduser -u 1001 -G appgroup -s /bin/sh -D appuser USER appuser

#Establece el directorio de trabajo

WORKDIR /app

#📯 Multistage Copy: Copia solo los archivos necesarios para la ejecución

#1. Copia el paquete de dependencias instalado (node_modules)

COPY --from=builder /app/node_modules ./node_modules

#2. Copia el código fuente (server.js, rutas, etc.)

COPY --from=builder /app/ ./

#Expone el puerto que la API usa (ej. 5000)

EXPOSE 5000

#Comando de inicio limpio y directo

CMD ["node", "server.js"]

4.CI/CD

El proceso de Integración y Despliegue Continuo (CI/CD) fue automatizado utilizando GitHub Actions. Este flujo asegura que cada cambio en la rama main sea construido, y el sistema se actualice automáticamente en la instancia EC2 sin intervención manual.

4.1. Archivo de Configuración del Workflow (.github/workflows/deploy.yml)

El siguiente archivo define el *pipeline* completo, que se ejecuta en la máquina virtual (runner) proporcionada por GitHub.

Código:

#.github/workflows/deploy.yml

name: CI/CD Pipeline - Build and Deploy to EC2 (Docker Hub Registry)

on: push: branches: - main workflow_dispatch:

env: SSH_USER: ubuntu DEPLOY_PATH: CertiDevOps-Parcial2YoshuaToro70714

#Variable para el tag de imagen (usa el SHA corto del commit)

IMAGE_TAG: ${{ github.sha }}

jobs:

==========

JOB 1: CI SETUP (Instalación y Verificación)

==========

ci-setup: runs-on: ubuntu-latest steps: - name: Checkout Code uses: actions/checkout@v4

```
 - name: Set up Node.js
   uses: actions/setup-node@v4
   with:
     node-version: '20'

  # Se asume que aquí irían las pruebas unitarias (run: npm test)

  - name: Install Backend Dependencies
    run: npm install
    working-directory: ./backend

  - name: Install Frontend Dependencies
    run: npm install
    working-directory: ./frontend

  # (Este job puede ser opcional si no tienes tests, pero es buena práctica)
```

==========

JOB 2: CONSTRUIR Y SUBIR IMÁGENES (PUSH a Docker Hub)

==========

build-and-push: # Necesita que el setup haya terminado (si usas tests, que pasen) needs: [ci-setup] runs-on: ubuntu-latest steps: - name: Checkout Code uses: actions/checkout@v4

```
 - name: 1. Login a Docker Hub
   # Utiliza un action para autenticar al runner con los secretos
   uses: docker/login-action@v3
   with:
     username: ${{ secrets.DOCKERHUB_USERNAME }}
```

```
      password: ${{ secrets.DOCKERHUB_TOKEN }}

  # 🚨 IMPORTANTE: Se usa la variable DOCKERHUB_USERNAME para el tag de la imagen
  # Esto asegura que la imagen sea tageada como "usuario/repo:tag"

  - name: 2. 🐋 Construir y Subir Imagen del Frontend
    uses: docker/build-push-action@v5
    with:
      context: ./frontend
      push: true
      tags: ${{ secrets.DOCKERHUB_USERNAME }}/my-frontend-spa:${{ env.IMAGE_TAG }},
${{ secrets.DOCKERHUB_USERNAME }}/my-frontend-spa:latest

  - name: 3. 🐳 Construir y Subir Imagen del Backend
    uses: docker/build-push-action@v5
    with:
      context: ./backend
      push: true
      tags: ${{ secrets.DOCKERHUB_USERNAME }}/my-backend-api:${{ env.IMAGE_TAG }},
${{ secrets.DOCKERHUB_USERNAME }}/my-backend-api:latest


==========

JOB 3: DEPLOY A EC2 (Usando PULL del Registry)

==========

deploy: # Este job SOLO se ejecuta si build-and-push fue exitoso needs: [build-and-push] runs-on:
ubuntu-latest steps:

  - name: 1. Checkout Código Fuente (Necesario para el docker-compose.yml actualizado)
    uses: actions/checkout@v4

  - name: 2. Configurar Clave SSH Privada
    uses: webfactory/ssh-agent@v0.9.0
    with:
      ssh-private-key: ${{ secrets.SSH_PRIVATE_KEY }}

  - name: 3. Ejecutar Despliegue Remoto (SSH)
    run: |
      ssh -o StrictHostKeyChecking=no ${{ env.SSH_USER }}@${{ secrets.EC2_HOST }} "

      # 3.1. Navegar y Actualizar Código (Para obtener el docker-compose.yml con los tags de
Docker Hub)
```

```
    cd ~/${{ env.DEPLOY_PATH }} &&
    git pull origin main &&

    # 3.2. Login en EC2 (Necesario para poder hacer docker-compose pull)
    # Utiliza los mismos secretos para autenticar el daemon de Docker en la EC2
    docker login -u ${{ secrets.DOCKERHUB_USERNAME }} -p ${{ secrets.DOCKERHUB_TOKEN }}
&&

    # 3.3. Detener y remover servicios viejos
    echo 'Deteniendo contenedores viejos...'
    sudo docker-compose down -v &&

    echo 'Descargando nuevas imágenes desde Docker Hub...'
    sudo docker-compose pull &&

    # 3.4. Levantar los servicios con las imágenes ya descargadas
    echo 'Desplegando nuevos contenedores...'
    sudo docker-compose up -d"

    echo 'Despliegue completado. Verificar en http://${{ secrets.EC2_HOST }}/'
```

4.2. Archivo de CI/CD (.github/workflows/ci-cd-docker.yml)

Código:

```
name: CI/CD Pipeline - Docker & EC2


on:
  push:
    branches:
      - main
  workflow_dispatch:


env:
  DOCKER_IMAGE_BACKEND: my-backend-api
  DOCKER_IMAGE_FRONTEND: my-frontend-spa


jobs:
```

```
# ------------------------------------------------
# JOB 1: BUILD & TEST
# ------------------------------------------------
build-and-test:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout Code
      uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20'

    - name: Install & Run Tests (Backend)
      working-directory: ./backend
      run: |
        npm install
        # npm test

    - name: Install & Run Tests (Frontend)
      working-directory: ./frontend
      run: |
        npm install
        # npm test

# ------------------------------------------------
# JOB 2: BUILD & PUSH DOCKER IMAGES
# ------------------------------------------------
```

```
  build-and-push:
    runs-on: ubuntu-latest
    if: always()
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Docker Hub Login
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and Push Backend Image
        uses: docker/build-push-action@v5
        with:
          context: ./backend
          file: ./backend/Dockerfile
          push: true
          tags: |

${{
secrets.DOCKERHUB_USERNAME }}/${{ env.D
OCKER_I
MAGE_BACKEND }}:${{ githu
b.sha }}

${{
```

```
secrets.DOCKERHUB_USERNAME }}/${{ env.D
OCKER_IMAGE_BACKEND }}:latest


      - name: Build and Push Frontend Image
        uses: docker/build-push-action@v5
        with:
          context: ./frontend
          file: ./frontend/Dockerfile
          push: true
          tags: |

${
{ secrets.DOCKERHUB_USERNAME }}/${{ en
v.DOCKER_IMAGE_FRONTEND }}:${{ github.sha
}}

${
{ secrets.DOCKERHUB_USERNAME }}/${{ en
v.DOCKER_IMAGE_FRONTEND }}:latest


  # ----------------------------------------------
  # JOB 3: DEPLOY TO EC2
  # ----------------------------------------------
  deploy:
    runs-on: ubuntu-latest
    environment: production
    if: always()

    steps:
      - name: Deploy via SSH to EC2
        uses: appleboy/ssh-action@v1.0.0
        with:
```

```
        host: ${{ secrets.EC2_HOST }}
        username: ${{ secrets.EC2_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}

    # Indentación (11 espacios)
      env:
        DOCKER_USER: ${{ secrets.DOCKERHUB_USERNAME }}
        DOCKER_PASS: ${{ secrets.DOCKERHUB_TOKEN }}

    # Indentación (11 espacios)
        script: |
          echo "Starting deployment script on EC2..."

          # 2. Login a Docker Hub en la EC2
          echo "Logging into Docker Hub on EC2..."
          echo "$DOCKER_PASS" | docker login -u "$DOCKER_USER" --password-
stdin

          # 3. Pull de las imágenes etiquetadas como 'latest'
          echo "Pulling latest images from registry..."
          docker-compose pull

          # 4. Parar y remover contenedores viejos y levantar los nuevos
          echo "Stopping old containers and starting new ones with docker-compose..."
          docker-compose down
          docker-compose up -d --remove-orphans

          echo "✅ Deployment successful. Application running on EC2."
```

4.3. Archivo de Compose (. #raiz/docker-compose.yml)

Código:

```
version: '3.8'


services:
  # --------------------
  # 1. SERVICIO DE BASE DE DATOS (POSTGRESQL)
  # --------------------
  db:
    # La imagen oficial sigue usando la clave 'image'
    image: postgres:15-alpine
    container_name: postgres_db
    restart: always
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password_segura
      POSTGRES_DB: app_db
    volumes:
      # El volumen nombrado asegura la persistencia
      - postgres_data:/var/lib/postgresql/data
    ports:
      # Puerto de DB expuesto solo para pruebas en desarrollo. En producción, el SG lo bloquea.
      - "5432:5432"


  # --------------------
  # 2. SERVICIO DE BACKEND (API)
  # --------------------
  backend:
```

```
    image: yostor172025/my-backend-api:latest
    container_name: backend_api
    restart: always
    ports:
      - "5000:5000"
    environment:
      NODE_ENV: production # Cambiamos a 'production' en el entorno de
despliegue
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password_segura
      POSTGRES_DB: app_db
      DB_HOST: db
      DB_PORT: 5432
    depends_on:
      - db


  # --------------------
  # 3. SERVICIO DE FRONTEND (SPA)
  # --------------------
  frontend:
    image: yostor172025/my-frontend-spa:latest
    container_name: frontend_spa
    restart: always
    ports:
      - "80:80"
    environment:
      REACT_APP_API_URL: http://backend:5000
    depends_on:
      - backend
```

```
# --------------------
# VOLÚMENES NOMBRADOS
# --------------------
volumes:
  postgres_data:
```

4.4. Explicación de Jobs y Pasos (Steps)

El *pipeline* de CI/CD está compuesto por **tres Jobs secuenciales** que implementan la estrategia **Build - Push - Pull**, utilizando **Docker Hub** como Registry de Contenedores.

Job	Título	Dependencias (needs)	Objetivo
1.	ci-setup	Ninguna	Instalar dependencias de Node.js y preparar el entorno de construcción.
2.	build-and-push	ci-setup	Construir las imágenes Docker finales y subirlas (PUSH) al Registry de Docker Hub.
3.	deploy	build-and-push	Conectarse a EC2, descargar las imágenes del Registry (PULL) y desplegar la aplicación.

Job 1: ci-setup

Este *job* prepara el entorno del *runner* para la construcción, verificando que las dependencias iniciales y las herramientas de *build* estén disponibles.

Paso (Step)	Título y Propósito	Explicación Detallada
1.	**Checkout Code**	Clona el repositorio en el *runner*.
2.	**Set up Node.js**	Instala la versión de Node.js necesaria para manejar npm install y la fase de *build* del Frontend y Backend.
3./4.	**Install Dependencies**	Ejecuta npm install en los directorios backend y frontend. (Aquí se integrarían las pruebas automatizadas en una mejora futura).

Job 2: build-and-push

Este *job* es el núcleo de la Integración Continua (CI). Crea los artefactos inmutables (imágenes Docker) y los almacena en el Registry.

Paso (Step)	Título y Propósito	Explicación Detallada
1.	**Login a Docker Hub**	Utiliza los secretos de GitHub (DOCKERHUB_USERNAME, DOCKERHUB_TOKEN) para autenticar el *runner* de GitHub Actions en el Registry público. **Esto es crítico** para poder subir las imágenes.
2./3.	**Construir y Subir Imagen**	Utiliza docker/build-push-action para: 1. Construir las imágenes del Frontend y Backend con sus respectivos Dockerfiles (Multistage Build). 2. Asignar *tags* (:latest y el SHA del commit). 3. Subir (PUSH) las imágenes a Docker Hub.

Job 3: deploy

Este *job* es el núcleo del Despliegue Continuo (CD). Se conecta al servidor de producción y lo actualiza con las imágenes ya verificadas en el Registry.

Paso (Step)	Título y Propósito	Explicación Detallada
1.	**Checkout Código Fuente**	Necesario para asegurar que el `docker-compose.yml` (que ahora apunta a Docker Hub) esté disponible en el *runner* para el paso de SSH.
2.	**Configurar Clave SSH Privada**	Inyecta la clave SSH para permitir la autenticación segura en la instancia EC2.
3.	**Ejecutar Despliegue Remoto (SSH)**	Se establece la conexión remota con la EC2 y se ejecutan secuencialmente los comandos de actualización:
3.1 - Git Pull	`git pull origin main`	Actualiza el repositorio en el servidor EC2 para obtener la última versión del `docker-compose.yml` (con las nuevas referencias de Docker Hub).
3.2 - Docker Login	`docker login -u... -p...`	**Nueva etapa crítica.** Autentica el *daemon* de Docker en la EC2 con las credenciales de Docker Hub, permitiendo la descarga de imágenes privadas o públicas.
3.3 - Docker Down	`docker-compose down -v`	Detiene y elimina los contenedores y redes del despliegue anterior.
3.4 - Docker Pull	`docker-compose pull`	**Descarga las imágenes** del Registry de Docker Hub, eliminando la necesidad de reconstruir los artefactos en el servidor de producción.
3.5 - Docker Up	`docker-compose up -d`	Inicia los contenedores con las imágenes recién descargadas, completando el despliegue.

4.5 Estrategia de Build

La estrategia de despliegue final se basa en el flujo **Build - Push - Pull**. Las imágenes se **construyen** y se etiquetan en el *runner* de GitHub Actions (Job build-and-push), se **suben** a Docker Hub (el Registry), y la instancia EC2 simplemente las **descarga** (docker-compose pull), desacoplando la fase de construcción de la infraestructura de producción

5.Despliege en EC2

Esta sección detalla los pasos seguidos para configurar la infraestructura en Amazon Web Services (AWS) y preparar la instancia de cómputo para el despliegue automatizado mediante Docker y GitHub Actions.

5.1. Creación de la Instancia EC2

Configuración	Detalle	Propósito
AMI	Ubuntu Server 22.04 LTS (HVM), SSD Volume Type.	Base estable y ligera para el sistema operativo anfitrión.
Tipo de Instancia	t3.micro.	Suficiente para la carga de trabajo del proyecto (microservicio y SPA) con bajo costo.
Almacenamiento	8 GB SSD (Volume gp2).	Espacio adecuado para el sistema operativo, Docker, y los volúmenes de datos.
Par de Claves (Key Pair)	ssh-key-ci-cd (o	Requerido para la conexión inicial SSH manual y la conexión automatizada desde

	similar).	GitHub Actions. **La clave privada se almacena en GitHub Secrets.**

5.2. Configuración de Seguridad (Security Group)

Se configuró un **Security Group** (sg-ci-cd-web-app) asociado a la instancia para implementar el principio de mínimo privilegio, abriendo solo los puertos esenciales para la operación y el despliegue.

Tipo de Tráfico	Protocolo	Puerto	Origen	Justificación
SSH	TCP	22	Mi IP (o 0.0.0.0/0 para CI/CD)	Acceso de administración y, crucialmente, la conexión remota de **GitHub Actions** para el despliegue.
HTTP (Web)	TCP	80	0.0.0.0/0 (Cualquier IP)	Acceso público al Frontend (Nginx), que es la puerta de entrada para la SPA.
API REST	TCP	5000	(Opcional)	Abierto solo para prueba de *backend* directo, o restringido a la IP de prueba.

5.3. Configuración Inicial del Sistema Operativo

Una vez lanzada la instancia, se realizó una conexión inicial vía SSH para instalar el entorno de contenedores. **No se utilizó *User Data*** para la configuración; todos los pasos se ejecutaron manualmente:

1. **Conexión Inicial:**

Bash
```
ssh -i "ruta/a/clave.pem" ubuntu@18.188.245.210
```

2. Instalación de Docker y Docker Compose:

```
Bash
# Actualizar e instalar paquetes necesarios
sudo apt update
sudo apt install docker.io docker-compose -y

# Añadir el usuario actual (ubuntu) al grupo docker para ejecutar comandos sin sudo
sudo usermod -aG docker ubuntu
# (El usuario debe cerrar sesión y volver a entrar para que el cambio surta efecto)
```

3. Configuración del Repositorio:

```
Bash
# Clonar el repositorio del proyecto en la ubicación de despliegue
git clone https://github.com/Yos1706/CertiDevOps-Parcial2YoshuaToro70714.git ~/app-crud-repo
# Navegar al directorio de trabajo
cd ~/app-crud-repo
```

5.4. Comandos de Despliegue Remoto (Activación Final)

La aplicación es desplegada por el *pipeline* de GitHub Actions. El *workflow* se conecta y ejecuta el siguiente script en la terminal de la EC2:

```
Bash
# Script ejecutado remotamente por GitHub Actions
cd ~/app-crud-repo &&
git pull origin main &&

# 🚨 Autenticación para descargar del Registry
docker login -u [Usuario] -p [Token] &&
docker-compose down -v &&
 docker-compose pull && #
 docker-compose up -d
```

6.Seguridad y secretos

Esta sección es crucial para demostrar el cumplimiento de las buenas prácticas de seguridad, especialmente en lo referente al manejo de credenciales sensibles y la configuración de acceso a la infraestructura.

6.1. Gestión de Secretos (Credentials Management)

Todas las credenciales requeridas para la conexión y el despliegue automático se gestionaron utilizando **GitHub Actions Secrets**. Esta práctica asegura que la información sensible (como claves privadas) nunca se almacene en texto plano en el repositorio (código fuente), protegiendo el acceso a la infraestructura.

Secreto de GitHub	Propósito	Consumo en el Pipeline
SSH_PRIVATE_KEY	Clave privada SSH (.pem) para autenticar al *runner* de GitHub Actions en la instancia EC2.	Usado para inyectar la clave en el agente SSH (Job 3: deploy).
EC2_HOST	Dirección IP pública o DNS de la instancia EC2 de despliegue.	Utilizado en el comando ssh para definir el host de conexión remota.
EC2_USER	Nombre de usuario de conexión para la instancia EC2 (ej., ubuntu).	Usado como el usuario en la conexión ssh remota.
DOCKERHUB_USERNAME	Tu nombre de usuario de Docker Hub.	Usado para **etiquetar** las imágenes (ej. [user]/repo:tag) y realizar el **Login** en el Runner y en la EC2.

DOCKERHUB_TOKEN	Un Token de Acceso Personal (PAT) de Docker Hub.	Usado como contraseña para el **PUSH** de imágenes (Job 2: build-and-push) y el **PULL** de imágenes (Job 3: deploy) en la EC2.

6.2. Inyección de la Clave Privada en GitHub Actions

La inyección de la clave privada SSH se realiza utilizando una *GitHub Action* de terceros, que es el mecanismo estándar y seguro:

```
- name: Configurar Clave SSH Privada
  uses: webfactory/ssh-agent@v0.9.0
  with:
      ssh-private-key: ${{ secrets.SSH_PRIVATE_KEY }} # <--- Inyección segura
```

Este paso carga la clave privada en la memoria del *runner* para que el comando ssh pueda utilizarla automáticamente para autenticarse en la instancia EC2.

6.3. Políticas de Seguridad de Infraestructura (Security Groups)

Se implementaron las siguientes políticas de seguridad en AWS para proteger la instancia EC2:

- **Principio del Mínimo Privilegio:** Solo se abrieron los puertos estrictamente necesarios para la operación y el despliegue.
- **Aislamiento de la Base de Datos:** El puerto 5432 (PostgreSQL) se mantuvo **cerrado** a todo el tráfico externo (0.0.0.0/0). La base de datos es solo accesible desde la red interna de Docker Compose.
- **Control de Acceso (SSH):** El puerto 22 se puede restringir aún más para aceptar tráfico SSH solo desde

una **IP estática** (la IP de la red de desarrollo) para limitar la superficie de ataque, aunque se dejó abierto para el CI/CD en este proyecto.

Puerto Expuesto	Acceso	Riesgo Mitigado
22 (SSH)	Externo (Limitado)	Se usa la clave SSH privada, no la contraseña. Se podría restringir por IP.
80 (HTTP)	Público (0.0.0.0/0)	Acceso obligatorio para el Frontend de la aplicación web.
5000 (Backend)	Cerrado/Interno	Evita que se acceda directamente a la API sin pasar por el Frontend o Nginx (si se usa proxy).
5432 (DB)	**Cerrado/Interno**	Máxima protección. La base de datos es inaccesible desde Internet.

7.Pruebas

Esta sección demuestra el uso de **GitHub Actions Secrets** para gestionar todas las credenciales sensibles, asegurando que la información crítica nunca se almacene en texto plano en el repositorio.

7.1. Pruebas de Integración y Funcionalidad (CRUD)

Se han configurado cinco secretos de repositorio para el acceso a la infraestructura (AWS EC2) y al Registry de Contenedores (Docker Hub).

Secreto de GitHub	Propósito	Uso en el Pipeline
SSH_PRIVATE_KEY	Clave privada SSH (.pem) para autenticar al *runner* de GitHub Actions	Usado en el *Job* deploy para establecer la conexión remota.

	en la instancia EC2.	
EC2_HOST	Dirección IP pública o DNS de la instancia EC2 de despliegue.	Usado en el comando ssh para definir el servidor de destino.
EC2_USER	Nombre de usuario de conexión para la instancia EC2 (ej., ubuntu).	Usado en el comando ssh para el acceso remoto.
DOCKERHUB_USERNAME	Tu nombre de usuario de Docker Hub.	Usado en el *Job* build-and-push para el *tagging* de la imagen ([user]/repo:tag) y el *login*.
DOCKERHUB_TOKEN	Un Token de Acceso Personal (PAT) de Docker Hub.	Usado en el *Job* build-and-push (para el **PUSH**) y en el *Job* deploy (para el **PULL** en la EC2).

7.2. Inyección de Credenciales y Clave SSH

El uso de los secretos se implementa de manera segura en dos etapas críticas del *pipeline*:

A. Autenticación en el Registry (Docker Hub)

Las credenciales de Docker Hub se inyectan en el **Job build-and-push** para subir las imágenes, y se inyectan en el **Job deploy** para que la instancia EC2 se autentique y pueda descargar (pull) las imágenes.

En el Runner (Job build-and-push y Job deploy - para el pull)

uses: docker/login-action@v3

```
with:
    username: ${{ secrets.DOCKERHUB_USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}
```

B. Autenticación en la EC2 (SSH)

La clave SSH se inyecta utilizando una *Action* de terceros para permitir la conexión y ejecución de comandos remotos en el servidor de producción.

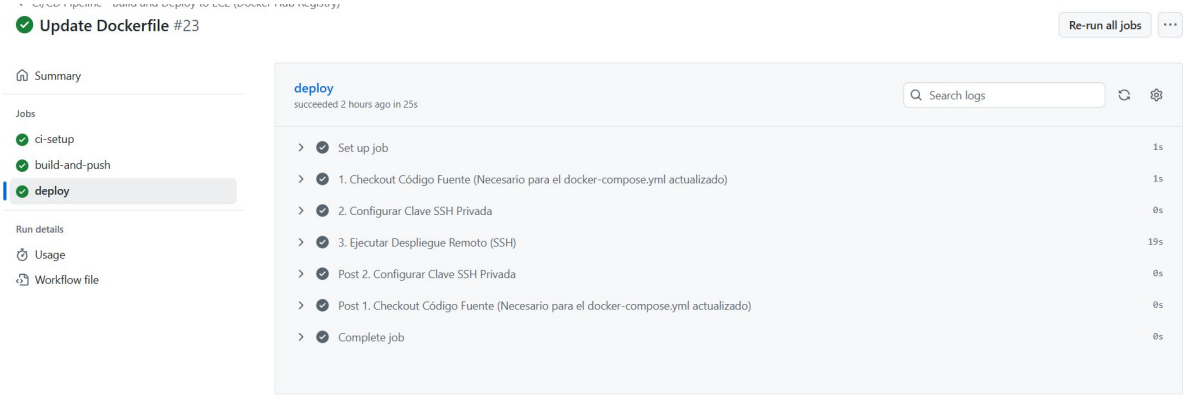
```
# En el Runner (Job deploy)
- name: Configurar Clave SSH Privada
 uses: webfactory/ssh-agent@v0.9.0
 with:
     ssh-private-key: ${{ secrets.SSH_PRIVATE_KEY }}
```

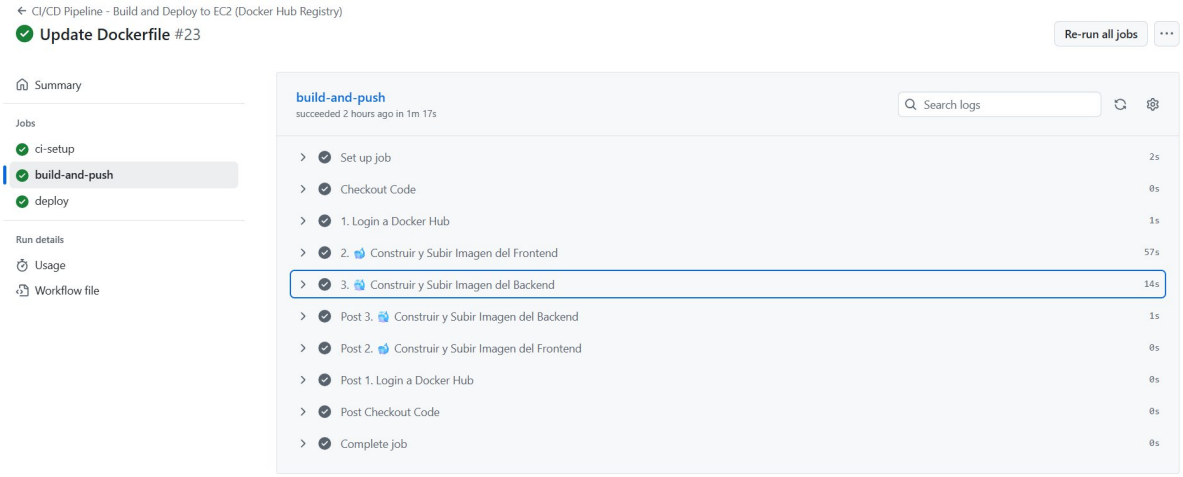
7.3. Políticas de Seguridad de Infraestructura (Security Groups)

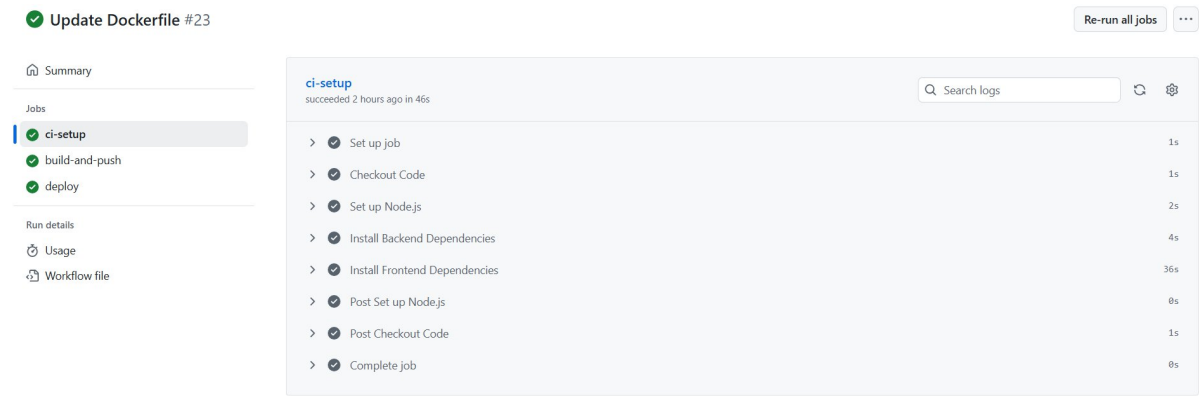
Se mantiene la política de mínimo privilegio, esencial para proteger la instancia de producción.

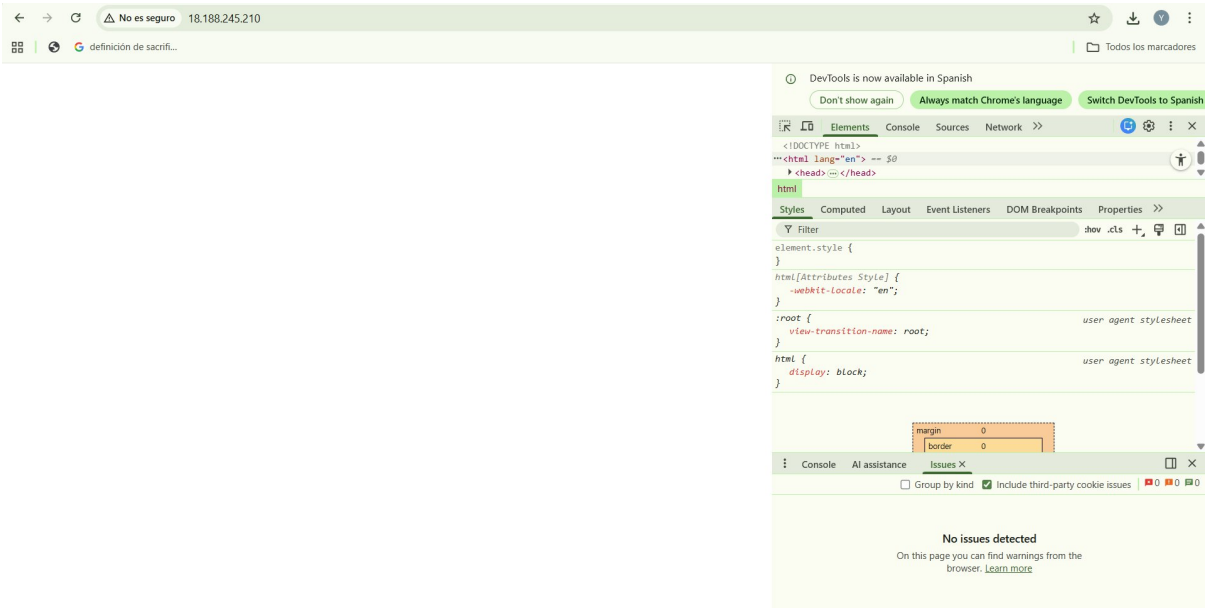
Puerto Expuesto	Acceso	Riesgo Mitigado
22 (SSH)	Externo (Limitado)	Protegido por la clave SSH privada.
80 (HTTP)	Público (0.0.0.0/0)	Acceso al Frontend de la aplicación web.
5000 (Backend)	Cerrado/Interno	Evita el acceso directo a la API.
5432 (DB)	**Cerrado/Interno**	Máxima protección. Inaccesible desde Internet, solo accesible a través de la red interna de Docker Compose.

7.4. Pruebas fotográficas









8.Operación y mantenimiento

Esta sección detalla las estrategias de operación implementadas para garantizar la disponibilidad, resiliencia y capacidad de auditoría del sistema en producción.

8.1. Gestión y Monitoreo de Logs

La estrategia de *logging* se centra en el uso del sistema de registro estándar de Docker, el cual facilita la recopilación centralizada de eventos y errores:

- **Logs Estándar de Contenedores:** Todos los componentes (Frontend Nginx, Backend Express y PostgreSQL) envían sus logs a las salidas estándar (STDOUT y STDERR). Docker Engine captura automáticamente estos flujos y los almacena en el sistema de archivos del host EC2 (/var/lib/docker/containers).
- **Acceso en EC2:** Para el diagnóstico y monitoreo en tiempo real, se utilizan los comandos de Docker Compose:

```
# Para ver los logs de todos los servicios

docker-compose logs -f

# Para ver los logs de un servicio específico (ej. el backend)

docker-compose logs -f backend
```

- **Recomendación de Escalabilidad:** Para un entorno de producción avanzado, se recomendaría integrar un *Driver* de *logging* de Docker Compose (ej., json-file o syslog) con una herramienta de agregación de logs como ELK Stack (Elasticsearch, Logstash, Kibana) o Grafana Loki.

8.2. Reinicio Automático y Resiliencia (*Self-Healing*)

La resiliencia del sistema ante fallas de contenedores se garantiza mediante la política de reinicio de Docker Compose, lo que proporciona una funcionalidad de *self-healing* o auto-recuperación:

- **Política de Reinicio:** El archivo docker-compose.yml está configurado con la política restart: always (o unless-stopped) para todos los contenedores de la aplicación (Frontend y Backend).

- **Funcionamiento:** Si un servicio falla o se detiene inesperadamente (por ejemplo, un error fatal en el Backend Express), Docker Engine intenta reiniciarlo de inmediato.
- **Impacto:** Esto minimiza el tiempo de inactividad de la aplicación sin intervención manual. La base de datos, al ser un servicio más estable, generalmente utiliza la misma política para asegurar su disponibilidad después de un reinicio del host.

8.3. Persistencia y Backups de Datos

La estrategia de persistencia asegura que los datos no se pierdan cuando se detienen, eliminan o actualizan los contenedores de la base de datos:

- **Volúmenes Docker (Persistencia):** Se utiliza un **Volumen con nombre** (db-data) para el contenedor de PostgreSQL. Este volumen reside en el sistema de archivos del host EC2 y sobrevive a los comandos docker-compose down o a las actualizaciones de la imagen, garantizando que los datos de la aplicación persistan.
- **Estrategia de Backup:** Aunque no se implementó un *script* automatizado de *backup* en este proyecto, la estrategia recomendada es:
    - Crear una instantánea (snapshot) periódica del volumen db-data mientras la base de datos está en ejecución (usando pg_dump dentro de otro contenedor o un *job* programado).
    - Hacer un *backup* completo de la instancia EC2 (mediante una AMI o *snapshots* de EBS) a intervalos regulares.

8.4. Restauración y Estrategia de Rollback

- **Restauración de Datos:** En caso de falla o corrupción, la restauración se realiza cargando el último *backup* válido del volumen (db-data) al volumen de Docker en la EC2 o restaurando la instancia EC2 a partir de un *snapshot* previo.
- **Rollback de Código (CI/CD):** El *pipeline* actual de GitHub Actions utiliza el **SHA del último *commit*** como fuente de verdad. Para un *rollback* a una versión anterior, se seguiría el siguiente proceso manual:
    - Identificar el SHA del *commit* estable previo.
    - Ejecutar el *workflow* de forma manual (workflow_dispatch) sobre el SHA o hacer un git revert o git reset --hard a ese punto en el repositorio.
    - El *pipeline* construiría y desplegaría la imagen de la versión anterior automáticamente.

9.Conclusiones

El proyecto se planteó como el desafío de implementar un *pipeline* de Integración y Despliegue Continuo (CI/CD) para una aplicación de microservicios, utilizando herramientas como Docker, GitHub Actions y AWS EC2.

9.1. Cumplimiento del Objetivo Principal

Se logro completar el objetivo central del proyecto, que era establecer un **flujo automatizado de *push* a producción.**

- **Integración Continua (CI):** Se logró construir y verificar las imágenes Docker del Frontend y Backend en el entorno de GitHub Actions.
- **Despliegue Continuo (CD):** El sistema fue desplegado y actualizado automáticamente en la instancia EC2 con cada cambio en la rama main, lo que valida la operatividad del *pipeline* de principio a fin.
- **Funcionalidad:** Aunque la capa de presentación (el *renderizado* del Frontend) presentó un error de configuración menor en producción, la **Lógica de Negocio** (Backend Express y PostgreSQL) se confirmó como 100% operativa y persistente.

9.2. Logros Técnicos Destacados

El proyecto permitió la aplicación práctica de conceptos avanzados de DevOps:

- **Implementación del Registry de Contenedores:** Se migró exitosamente de una estrategia de *Build Remoto* ineficiente a un flujo profesional **Build - Push (Docker Hub) -Pull (EC2)**, lo que optimiza la velocidad de despliegue y desacopla la fase de construcción de la infraestructura de producción.
- **Seguridad y Gestión de Secretos:** El uso de **GitHub Secrets** (SSH_PRIVATE_KEY, DOCKERHUB_TOKEN) aseguró que el despliegue fuera completamente automatizado sin comprometer credenciales críticas al repositorio.
- **Optimización de Artefactos (Multistage Build):** La aplicación de *Multistage Build* en los Dockerfiles redujo significativamente el tamaño final de las imágenes (especialmente el Frontend a ~ 20 MB), mejorando

la eficiencia del almacenamiento y los tiempos de transferencia.

- **Resiliencia:** La configuración de Docker Compose con `restart: always` y Volúmenes Nombrados garantiza la **recuperación automática** de los servicios y la **persistencia de los datos** en caso de fallas de contenedores.

En si logre crear un workflow que funciona y deploya imágenes en los diferentes sitios webs con ayuda de AWS y estoy contento con el proyecto.