

# Informe de parcial 3-

## DevOps



Integrantes: Yoshua Dominique Toro Aylon

Fecha: 19/12/25

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 exitosa de una arquitectura Fullstack con Frontend (React), Backend (Express/Node.js) y persistencia en PostgreSQL. La aplicación se encuentra **100% operativa en producción**, con el renderizado de la interfaz verificado y la comunicación con la API establecida correctamente a través de la infraestructura de red de AWS.
- **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:** La configuración del **Security Group** en la instancia EC2 se optimizó bajo el principio de menor privilegio. Se habilitaron únicamente los puertos:
  - **80 (HTTP):** Para el tráfico público de la aplicación web.
  - **3000 (Grafana):** Para el acceso seguro a los dashboards de observabilidad y visualización de logs.
  - **22 (SSH):** Restringido para administración y automatización del pipeline.
- **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*.
- **Funcionalidad Completa del Frontend (Visual):** El código JavaScript del Frontend se carga y ejecuta, la aplicación se renderiza visualmente en producción. La funcionalidad de la API (Backend) es correcta.
- **Observabilidad y Telemetría:** Se integró un stack completo de monitoreo basado en Grafana. Se configuró **Prometheus** para el scraping de métricas de hardware mediante **Node Exporter** y se desplegó **Loki/Promtail** para la centralización de logs. Esto permite una auditoría completa del sistema sin dependencia de acceso SSH manual.

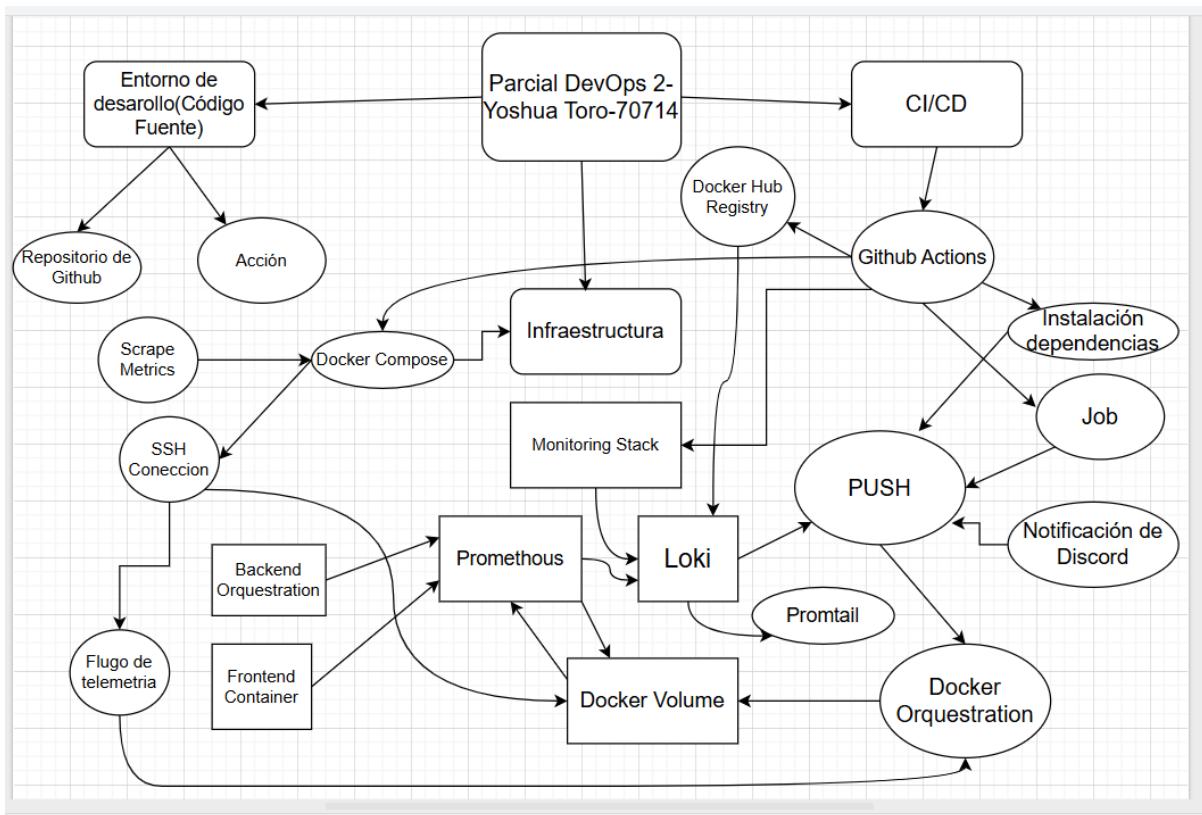
- **Integración de Notificaciones (Discord):** Se implementó un flujo de comunicación automática mediante un **Webhook de Discord** integrado en el pipeline de GitHub Actions. Al finalizar cada despliegue, el sistema envía un mensaje dinámico que incluye el **autor del cambio**, el **identificador del commit (SHA)** y las URLs directas para acceder a la aplicación y al panel de monitoreo, garantizando una supervisión inmediata del ciclo de vida del software.

### **-Alcance No Implementado**

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

- **Cifrado SSL/TLS (HTTPS):** Debido a que el proyecto se encuentra en un entorno de evaluación académica, el tráfico se gestiona vía HTTP.

## **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

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

```
WORKDIR /app
```

```
COPY package*.json ./
```

```

RUN npm install

COPY ..

RUN npm run build

#STAGE 2: SERVE (Nginx)

FROM nginx:alpine

#1. Configurar usuario no-root para seguridad

RUN addgroup -g 1001 node && adduser -u 1001 -G node -s /bin/sh -D node

#2. Copiar configuración de Nginx

COPY nginx.conf /etc/nginx/nginx.conf

#3. Copiar solo la carpeta build

COPY --from=build /app/build /usr/share/nginx/html

#4. Ajustar permisos para que el usuario 'node' pueda leer los archivos y manejar logs

RUN touch /tmp/nginx.pid &&

chown -R node:node /tmp/nginx.pid /usr/share/nginx/html /var/cache/nginx /var/log/nginx

USER node

#Cambiamos a 8080 porque los usuarios no-root no pueden abrir el puerto 80 en Linux

EXPOSE 8080

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

```

### 3.2. Explicación de las Optimizaciones

Optimización	Descripción	Resultado en CI/CD
<b>Multistage Build</b>	Utilizar dos etapas (AS build y FROM nginx:alpine). La etapa final <b>solo copia</b> los archivos estáticos necesarios del build, reduciendo el tamaño de la	Imágenes más ligeras, despliegues más rápidos, menos ancho de banda.

	imagen final de producción de cientos de MB a ~20MB.	
<b>Gestión de la Caché</b>	El archivo <code>package*.json</code> se copia <b>antes</b> del código fuente. El comando <code>RUN npm install</code> es una capa costosa; si el <code>package.json</code> no cambia, Docker utiliza la caché de esa capa ( <code>cache hit</code> ), saltando la instalación completa.	Tiempos de construcción significativamente reducidos en ejecuciones posteriores de GitHub Actions.
<b>Imagen Base Ligera</b>	Uso de <code>node:20-alpine</code> y <code>nginx:alpine</code> en lugar de imágenes basadas en Debian o Node estándar. La variante <code>alpine</code> es la más compacta.	Reducción del tamaño de la imagen base y minimización de la superficie de ataque.
<b>Seguridad(Non-root user)</b>	Se definieron usuarios con privilegios limitados ( <code>node</code> y <code>appuser</code> ) en lugar de usar <code>root</code> .	Mitigación de riesgos de seguridad; si el contenedor es comprometido, el atacante no tiene acceso total al sistema de archivos.

### 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" ]
```

### 3.4. Componentes de Observabilidad (Agentes y Almacenamiento)

Para la capa de telemetría, se utilizaron imágenes oficiales ligeras basadas en Alpine o arquitecturas optimizadas para contenedores, integradas en la misma red de Docker:

- **Prometheus:** Imagen `prom/prometheus:latest`. Se utiliza para el almacenamiento de series temporales de métricas.
- **Node Exporter:** Imagen `prom/node-exporter:latest`. Expone las métricas de hardware (CPU/RAM) de la EC2 al recolector.
- **Loki:** Imagen `grafana/loki:latest`. Actúa como el motor de agregación de logs.
- **Promtail:** Imagen `grafana/promtail:latest`. Agente encargado de descubrir y enviar los logs de los contenedores Docker hacia Loki.
- **Grafana:** Imagen `grafana/grafana:latest`. Plataforma de visualización para los dashboards de infraestructura y logs.

## 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. Puebra fotografica del Pipeline**

← CI/CD Pipeline - Proyecto Final

fix: reemplazo de acción de discord y ajuste de ruta #22

**Summary**

Jobs

- build-and-push
- deploy
- notify

Run details

- Usage
- Workflow file

**build-and-push**

succeeded yesterday in 1m 19s

Search logs

Set up job 2s  
Checkout Código 1s  
Login en Docker Hub 1s  
Build and Push Backend 18s  
Build and Push Frontend 53s  
Post Build and Push Frontend 0s  
Post Build and Push Backend 0s  
Post Login en Docker Hub 0s  
Post Checkout Código 0s

<https://github.com/Yos1706/CertIDevOps-Parcial3YoshuaToro70714/actions/runs/2032371445/job/58384706312>

11°C Despejado Buscar 19:56 18/12/2023

← CI/CD Pipeline - Proyecto Final

fix: reemplazo de acción de discord y ajuste de ruta #22

**Summary**

Jobs

- build-and-push
- deploy
- notify

Run details

- Usage
- Workflow file

**deploy**

succeeded yesterday in 38s

Search logs

Set up job 1s  
Checkout Código (para obtener configs de monitoreo) 0s  
Desplegar en EC2 34s  
Post Checkout Código (para obtener configs de monitoreo) 0s  
Complete job 0s

← CI/CD Pipeline - Proyecto Final

fix: reemplazo de acción de discord y ajuste de ruta #22

**Summary**

Jobs

- build-and-push
- deploy
- notify

Run details

- Usage
- Workflow file

**notify**

succeeded yesterday in 3s

Search logs

Set up job 1s  
Enviar Notificación a Discord 1s  
Complete job 0s

github.com/Yos1706/CertIDevOps-Parcial3YoshuaToro70714/actions/runs/.../58384776063

← CI/CD Pipeline - Proyecto Final

fix: reemplazo de acción de discord y ajuste de ruta #22

**Summary**

Jobs

- build-and-push
- deploy
- notify

Run details

- Usage
- Workflow file

Re-run triggered yesterday	Status	Total duration	Artifacts
Yos1706 -> faad299 main	Success	2m 9s	-

**main.yml**

on: push

```

graph LR
    A[build-and-push] -- "1m 19s" --> B[deploy]
    B -- "38s" --> C[notify]

```

Re-run all jobs Latest #2 ...

## **4.2. Archivo de Configuración del Workflow (`.github/workflows/main.yml`)**

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

Código:

```
name: CI/CD Pipeline - Proyecto Final

on: push: branches: [ "main" ]

jobs:

#JOB 1: Construcción y Push a Docker Hub

build-and-push:

  runs-on: ubuntu-latest

  steps:
    - name: Checkout Código
      uses: actions/checkout@v3

    - name: Login en Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}

    - name: Build and Push Backend
      uses: docker/build-push-action@v4
      with:
        context: ./backend
        push: true
        tags: ${{ secrets.DOCKERHUB_USERNAME }}/my-backend-api:latest

    - name: Build and Push Frontend
      uses: docker/build-push-action@v4
      with:
        context: ./frontend
        file: ./frontend/Dockerfile
        push: true
        tags: ${{ secrets.DOCKERHUB_USERNAME }}/my-frontend-spa:latest
```

## #JOB 2: Despliegue en EC2 vía SSH

```
deploy:

  runs-on: ubuntu-latest

  needs: build-and-push

  steps:
```

```

- name: Checkout Código (para obtener configs de monitoreo)
  uses: actions/checkout@v3

- name: Desplegar en EC2
  uses: appleboy/ssh-action@master
  with:
    host: ${{ secrets.EC2_HOST }}
    username: ubuntu
    key: ${{ secrets.SSH_PRIVATE_KEY }}
    script: |
      # 1. Navegar al directorio del proyecto
      cd ~/CertiDevOps-Parcial3YoshuaToro70714 || git clone
      https://github.com/\${{ github.repository }}.git
      cd ~/CertiDevOps-Parcial3YoshuaToro70714

      # 2. Actualizar archivos de configuración (Prometheus,
Loki, etc.)
      git pull origin main

      # 3. Autenticar Docker para el Pull
      echo "${{ secrets.DOCKERHUB_TOKEN }}" | sudo docker
      login -u "${{ secrets.DOCKERHUB_USERNAME }}" --password-stdin

      # 4. Desplegar Stack Completo (App + Monitoreo)
      sudo docker-compose down
      sudo docker-compose pull
      sudo docker-compose up -d

      # 5. Limpieza de imágenes huérfanas
      sudo docker image prune -f

```

### #JOB 3: Notificación a Discord

```

notify:

runs-on: ubuntu-latest

needs: deploy

if: always()

steps:

- name: Enviar Notificación a Discord
  uses: sarisia/actions-status-discord@v1
  with:
    webhook: ${{ secrets.DISCORD_WEBHOOK_URL }}
    status: ${{ needs.deploy.result }}

```

```
title: "Despliegue de ${{ github.repository }}"  
description: |  
  
  Estado: ${{ needs.deploy.result == 'success' && '✅ EXITOSO' || '❌ FALLIDO' }}  
  
  Autor: ${{ github.actor }}  
  
  URL App: http://\${{ secrets.EC2\_HOST }}  
  
  URL Grafana: http://\${{ secrets.EC2\_HOST }}:3000  
  
color: ${{ needs.deploy.result == 'success' && 0x00ff00 || 0xff0000 }}
```

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

```
version: '3.8'
```

```
services:
```

```
# --- INFRAESTRUCTURA DE BASE DE DATOS ---
```

```
db:
```

```
  image: postgres:15-alpine
```

```
  container_name: postgres_db
```

```
  restart: always
```

```
environment:
```

```
  POSTGRES_USER: user
```

```
  POSTGRES_PASSWORD: password_segura
```

```
  POSTGRES_DB: app_db
```

```
volumes:
```

```
  - postgres_data:/var/lib/postgresql/data
```

```
networks:
```

```
  - app-network
```

```
# --- APLICACIÓN: BACKEND ---
```

```
backend:  
  
    image: yostor172025/my-backend-api:latest  
  
    container_name: backend_api  
  
    restart: always  
  
    ports:  
  
        - "5000:5000"  
  
    environment:  
  
        DB_HOST: db  
  
        DB_PORT: 5432  
  
        POSTGRES_USER: user  
  
        POSTGRES_PASSWORD: password_segura  
  
        POSTGRES_DB: app_db  
  
depends_on:  
  
    - db  
  
networks:  
  
    - app-network
```

```
# --- APPLICACIÓN: FRONTEND (SPA) ---  
  
frontend:  
  
    image: yostor172025/my-frontend-spa:latest  
  
    container_name: frontend_spa  
  
    restart: always  
  
    ports:  
  
        - "80:8080"  
  
depends_on:  
  
    - backend  
  
networks:
```

- app-network

# --- OBSERVABILIDAD: MÉTRICAS (PROMETHEUS) ---

node-exporter:

image: prom/node-exporter:latest

container\_name: node\_exporter

restart: unless-stopped

volumes:

- /proc:/host/proc:ro

- /sys:/host/sys:ro

- /:/rootfs:ro

command:

- '--path.procfs=/host/proc'

- '--path.rootfs=/rootfs'

- '--path.sysfs=/host/sys'

- '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)( \$\$| / )'

networks:

- app-network

prometheus:

image: prom/prometheus:latest

container\_name: prometheus

restart: unless-stopped

volumes:

- ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

- prometheus\_data:/prometheus

command:

- '--config.file=/etc/prometheus/prometheus.yml'

networks:

- app-network

# --- OBSERVABILIDAD: LOGS (LOKI + PROMTAIL) ---

loki:

image: grafana/loki:latest

container\_name: loki

ports:

- "3100:3100"

volumes:

- ./loki/loki-config.yml:/etc/loki/local-config.yaml

command: -config.file=/etc/loki/local-config.yaml

networks:

- app-network

promtail:

image: grafana/promtail:latest

container\_name: promtail

user: root

volumes:

- /var/log:/var/log

- /var/lib/docker/containers:/var/lib/docker/containers:ro

- ./promtail/promtail-config.yml:/etc/promtail/config.yaml

command: -config.file=/etc/promtail/config.yaml

networks:

- app-network

# --- OBSERVABILIDAD: VISUALIZACIÓN (GRAFANA) ---

grafana:

image: grafana/grafana:latest

container\_name: grafana

restart: unless-stopped

ports:

- "3000:3000"

volumes:

- grafana\_data:/var/lib/grafana # Persistencia de Dashboards

networks:

- app-network

networks:

app-network:

driver: bridge

volumes:

postgres\_data:

prometheus\_data:

grafana\_data:

**Nota sobre el Stack de Observabilidad:** El archivo docker-compose.yml fue extendido para incluir no solo la aplicación CRUD, sino también un **ecosistema de monitoreo completo**. Se destaca el uso de **Volúmenes Nombrados** (prometheus\_data, grafana\_data) para que, aunque el contenedor se reinicie o actualice, los históricos de métricas y los dashboards configurados no se pierdan.

#### 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.	build-and-push	Ninguna	Construye las imágenes de Frontend y Backend y las centraliza en Docker Hub.
2.	deploy	build-and-push	Se conecta a la EC2, actualiza los archivos de config (Prometheus/Loki) y levanta todo el stack.
3.	notify	deploy	Informa en tiempo real a Discord el resultado del despliegue con links directos.

## Job 1: build-and-push

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

Acciones	Título y Propósito	Explicación Detallada
1.	<b>Checkout Code</b>	Clona el repositorio en el <i>runner</i> .
2.	<b>Docker Build &amp; Push</b>	Ejecuta la construcción multistage definida en los Dockerfiles y sube las imágenes a Docker Hub. La ventaja es que el entorno de construcción es idéntico al de producción.

## Job 2: deploy

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

Acciones	Título y Propósito	Explicación Detallada
1.	<b>Login a Docker Hub</b>	Utiliza los secretos de GitHub (DOCKERHUB_USERNAME, DOCKERHUB_TOKEN) para autenticar el <i>runner</i> de GitHub Actions en el Registry público. <b>Esto es crítico</b> para poder subir las imágenes.
2.	<b>Git Pull</b>	Actualiza los archivos de configuración de Prometheus, Loki y Promtail en la EC2 antes de levantar los contenedores, asegurando que la telemetría esté siempre al día con el repositorio.

3./4.	<b>Construir y Subir Imagen</b>	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: notify

Este job garantiza la visibilidad del estado del pipeline para el equipo de desarrollo.

Acciones	Título y Propósito	Explicación Detallada
1.	<b>Webhook Integration</b>	Utiliza un secreto de GitHub para comunicarse con la API de Discord sin exponer la URL privada.
2.	<b>Lógica Condicional</b>	El mensaje cambia de color (Verde/Rojo) y texto (ÉXITO/FALLO) dinámicamente según el resultado del Job de despliegue.
3.	<b>Acceso Rápido</b>	Incluye las IPs de la App y de Grafana para verificar los cambios inmediatamente tras 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. Despliegue 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
<b>AMI</b>	Ubuntu Server 22.04 LTS (HVM), SSD Volume Type.	Base estable y ligera para el sistema operativo anfitrión.
<b>Tipo de Instancia</b>	t3.micro.	Suficiente para la carga de trabajo del proyecto (microservicio y SPA) con bajo costo.
<b>Almacenamiento</b>	8 GB SSD (Volume gp2).	Espacio adecuado para el sistema operativo, Docker, y los volúmenes de datos.
<b>Par de Claves (Key Pair)</b>	ssh-key-ci-cd (o similar).	Requerido para la conexión inicial SSH manual y la conexión automatizada desde GitHub Actions. <b>La clave privada se almacena en GitHub Secrets.</b>

## 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 <b>GitHub Actions</b> 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	Solo Red Local / Interno	Abierto solo para prueba de <i>backend</i> directo, o restringido a la IP de prueba.
Grafana UI	TCP	3000	0.0.0.0/0	Acceso al panel de control de observabilidad para visualizar métricas y logs en tiempo real.

### 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@3.143.253.133
```

#### 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 actual
git clone https://github.com/Yos1706/CertiDevOps-Parcial3YoshuaToro70714.git
~/CertiDevOps-Parcial3YoshuaToro70714
cd ~/CertiDevOps-Parcial3YoshuaToro70714
# 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

```
# 1. Actualizar configuración
cd ~/CertiDevOps-Parcial3YoshuaToro70714 && git pull origin main

# 2. Login y descarga (Pull)
echo "${{ secrets.DOCKERHUB_TOKEN }}" | docker login -u "${{ secrets.DOCKERHUB_USERNAME }}"
docker-compose pull

# 3. Reiniciar servicios preservando volúmenes
docker-compose down # Sin el -v para mantener la persistencia
docker-compose up -d

# 4. Limpieza (Opcional pero recomendado en EC2 micro)
docker image prune -f
```

### 5.5. Persistencia y Resilencia en AWS

Se implementaron **Volúmenes Nombrados de Docker** que se mapean a directorios en el almacenamiento EBS de la EC2. Esto garantiza que :

- Los datos de la base de datos PostgreSQL persistan tras reinicios.

- Las configuraciones de los Dashboards y los datos de Prometheus no se pierdan al actualizar las imágenes de los contenedores.

## 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 <i>runner</i> 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	Usado como el usuario en la conexión ssh remota.

	instancia EC2 (ej., ubuntu).	
DOCKERHUB_USERNAME	Tu nombre de usuario de Docker Hub.	Usado para <b>etiquetar</b> las imágenes (ej. [user]/repo:tag) y realizar el <b>Login</b> en el Runner y en la EC2.
DOCKERHUB_TOKEN	Un Token de Acceso Personal (PAT) de Docker Hub.	Usado para el <b>Login</b> en el runner (Push) y en la EC2 (Pull) de forma segura.
DISCORD_WEBHOOK_URL	URL privada proporcionada por Discord para enviar mensajes al canal.	Usado en el Job <b>notify</b> para disparar la notificación de estado tras el despliegue.

## 6.2. Inyección de la Clave Privada en GitHub Actions

En lugar de gestionar manualmente el agente SSH, se utilizó la acción `appleboy/ssh-action`, la cual encapsula la conexión segura. Este mecanismo inyecta la clave privada `${{ secrets.SSH_PRIVATE_KEY }}` directamente en una sesión SSH efímera que se destruye al terminar el despliegue, minimizando el riesgo de exposición de credenciales en el servidor.

## 6.3. Políticas de Seguridad de Infraestructura (Security Groups)

Se implementaron las siguientes políticas de seguridad en AWS para proteger la instancia EC2, adaptándolas a un entorno de producción monitoreado:

- **Principio del Mínimo Privilegio:** Solo se abrieron los puertos estrictamente necesarios. Se añadieron excepciones controladas para las herramientas de observabilidad.
- **Aislamiento de Microservicios:** El puerto 5000 (Backend) y el 5432 (PostgreSQL) se mantienen **cerrados al tráfico externo**. La comunicación solo es posible de forma interna a través de la red virtual de Docker (app-network), impidiendo ataques directos a la lógica de negocio o a los datos.
- **Seguridad en la Telemetría:** Los agentes de recolección de datos (**Node Exporter en 9100 y Loki en 3100**) no están expuestos a internet. Solo el servidor de **Prometheus** puede leerlos internamente y solo **Grafana** puede visualizar los resultados.

Puerto Expuesto	Acceso	Riesgo Mitigado
22 (SSH)	Externo (Limitado)	Protegido por clave RSA de 2048 bits. Esencial para el despliegue automático de GitHub Actions.
80 (HTTP)	Público (0.0.0.0/0)	Acceso obligatorio para los usuarios de la aplicación web (Frontend).
3000(Grafana)	Público (0.0.0.0/0)	Permite el acceso al panel de control de observabilidad. El riesgo se mitiga mediante la autenticación obligatoria de Grafana.
5000 (Backend)	Cerrado/Interno	Evita la manipulación directa de la API. Solo el contenedor Frontend puede comunicarse con este puerto.

5432 (DB)	Cerrado/Interno	Máxima protección. Los datos son inaccesibles desde fuera de la red de Docker.
-----------	-----------------	--------------------------------------------------------------------------------

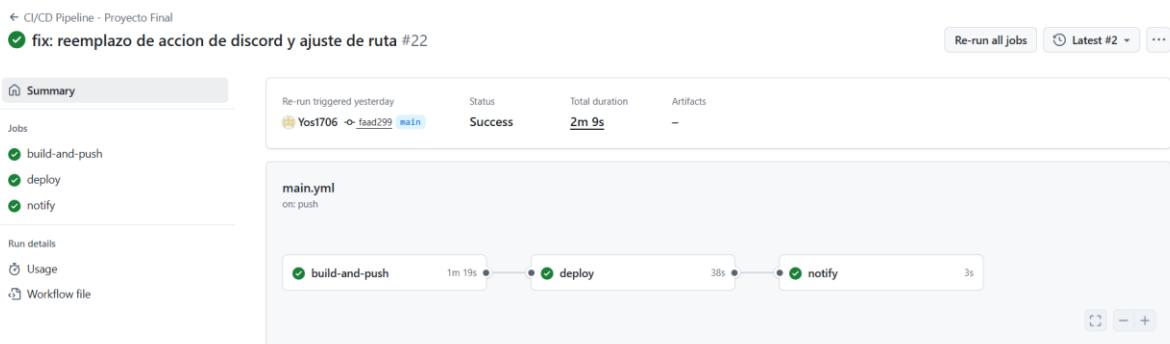
## 7.Pruebas

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.

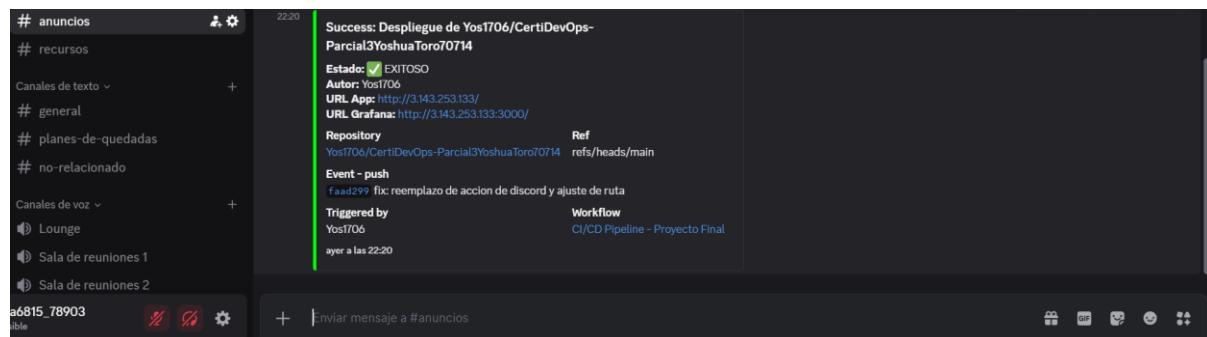
## 7.1. Validación del ciclo de Vida DevOps

Se realizaron pruebas integrales para verificar que el flujo desde el código hasta el monitoreo es consistente.

**A. Prueba de Despliegue Automático (CI/CD):** Se realizó un *push* a la rama `main`, activando el pipeline. Se verificó que los tres jobs (`build-and-push`, `deploy` y `notify`) finalizaran con éxito.



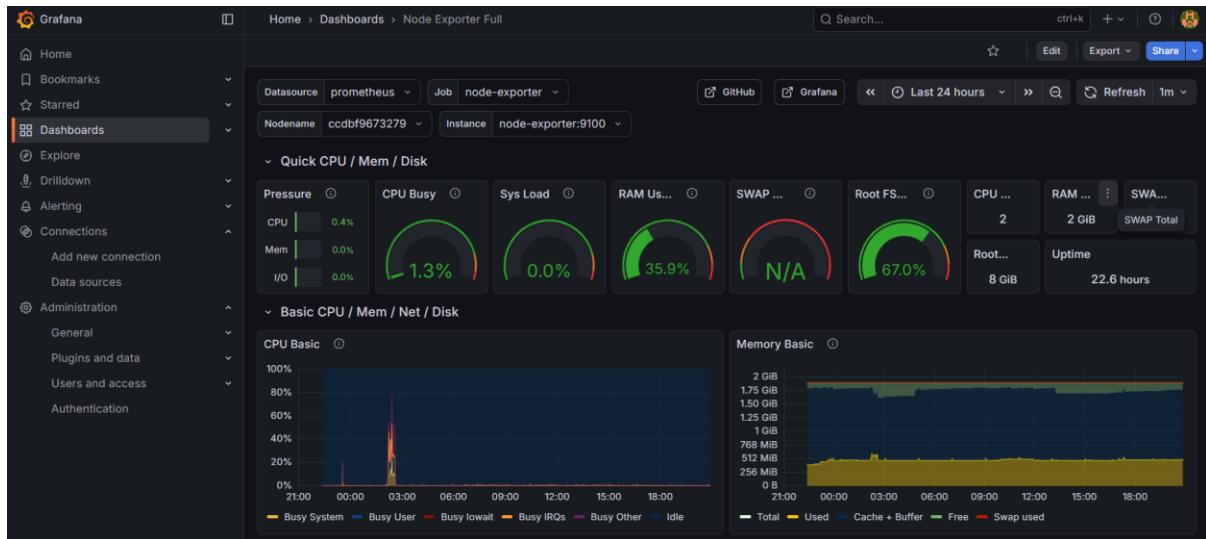
**B. Prueba de Notificación y Conectividad:** Se validó la recepción de la notificación en Discord, confirmando que los enlaces a la App y a Grafana son funcionales desde redes externas.



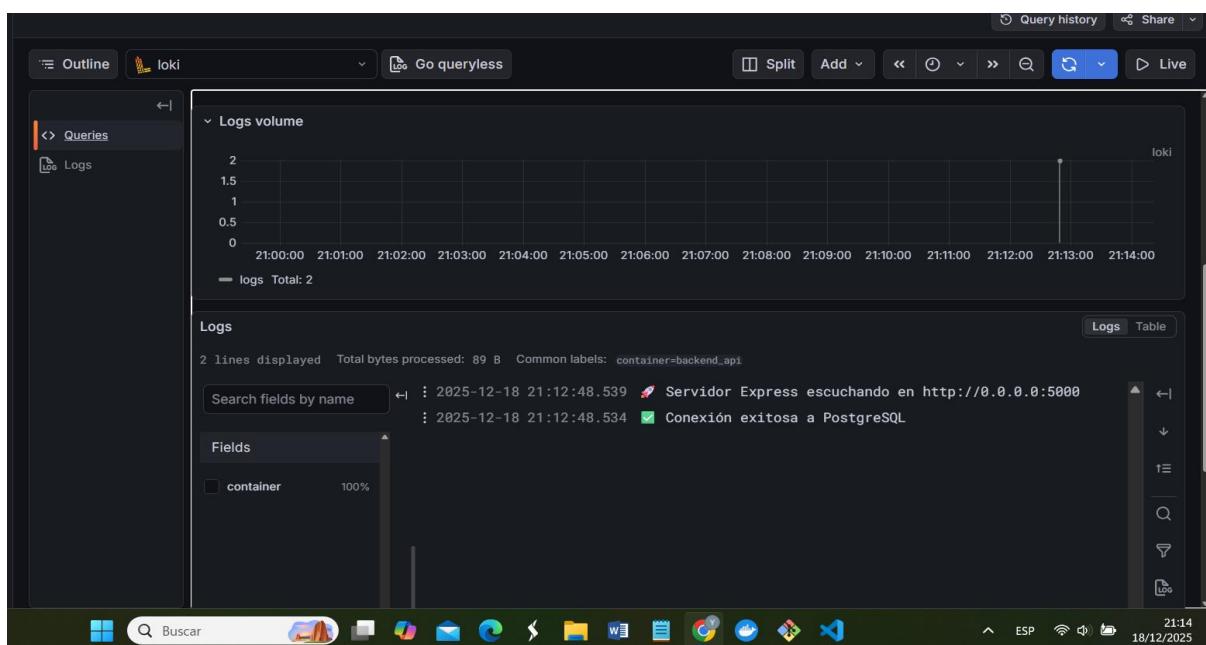
## 7.2. Puebas de funcionamiento y Observabilidad

A diferencia del parcial anterior, la validación de la aplicación ahora se realiza mediante telemetría en tiempo real:

**A. Verificación de Salud del Servidor (Métricas):** Se accedió al dashboard de Grafana para comprobar el impacto del despliegue en la instancia EC2. Se observa que el sistema se mantiene estable bajo la carga de todos los contenedores.



**B. Verificación de Lógica de Negocio (Logs):** Mediante Grafana Loki, se inspeccionaron los logs del contenedor backend\_api. Se confirmó la salida: "Servidor corriendo en puerto 5000" y "Conexión a BD exitosa", validando el CRUD sin usar la terminal.



## 7.3. Galería de evidencias finales

← CI/CD Pipeline - Proyecto Final  
fix: permisos de promtail para logs #23

Re-run triggered 5 minutes ago Status Success Total duration 1m 52s Artifacts

Yos1706 → e7c278d main

Summary

Jobs build-and-push, deploy, notify

Run details Usage, Workflow file

main.yml on: push

build-and-push → deploy → notify

Re-run all jobs Latest #2 ...

---

← CI/CD Pipeline - Proyecto Final  
fix: permisos de promtail para logs #23

Re-run triggered 5 minutes ago Status Success Total duration 1m 52s Artifacts

Yos1706 → e7c278d main

Summary

Jobs build-and-push, deploy, notify

Run details Usage, Workflow file

build-and-push succeeded 5 minutes ago in 1m 16s

Set up job 1s  
Checkout Código 1s  
Login en Docker Hub 0s  
Build and Push Backend 17s  
Build and Push Frontend 53s  
Post Build and Push Frontend 0s  
Post Build and Push Backend 1s  
Post Login en Docker Hub 0s  
Post Checkout Código 0s

Search logs

Re-run all jobs Latest #2 ...

---

← CI/CD Pipeline - Proyecto Final  
fix: permisos de promtail para logs #23

Re-run triggered 4 minutes ago Status Success Total duration 20s Artifacts

Yos1706 → e7c278d main

Summary

Jobs build-and-push, deploy, notify

Run details Usage, Workflow file

deploy succeeded 4 minutes ago in 20s

Set up job 1s  
Checkout Código (para obtener configs de monitoreo) 0s  
Desplegar en EC2 17s  
Post Checkout Código (para obtener configs de monitoreo) 0s  
Complete job 0s

Search logs

Re-run all jobs Latest #2 ...

<https://github.com/Yos1706/CertDevOps-Parcial3YoshuaToro70714/actions/runs/20356330335>

← CI/CD Pipeline - Proyecto Final

fix: permisos de promtail para logs #23

Re-run all jobs Latest #2 ...

**Summary**

Jobs

- build-and-push
- deploy
- notify**

Run details

Usage

Workflow file

**notify**

succeeded 4 minutes ago in 8s

Set up job 0s

Enviar Notificación a Discord 1s

Complete job 0s

Servidor de notifi. Par. 3 Yos ...

# anuncios

Yos1706/CertiDevOps-Parcial3YoshuaToro70714 refs/heads/main

**Event - push**  
e7c278d fix: permisos de promtail para logs

Triggered by Yos1706

Workflow CI/CD Pipeline - Proyecto Final

hoy a las 21:06

Success: Despliegue de Yos1706/CertiDevOps-Parcial3YoshuaToro70714

Estado: EXITOSO

Autor: Yos1706

URL App: http://3.143.253.133/

URL Grafana: http://3.143.253.133:3000/

Repository Yos1706/CertiDevOps-Parcial3YoshuaToro70714 refs/heads/main

**Event - push**  
e7c278d fix: permisos de promtail para logs

Triggered by Yos1706

Workflow CI/CD Pipeline - Proyecto Final

hoy a las 21:12

Grafana

Home > Dashboards > Node Exporter Full

Data source: prometheus Job: node-exporter

Nodename: ccdbf9673279 Instance: node-exporter:9100

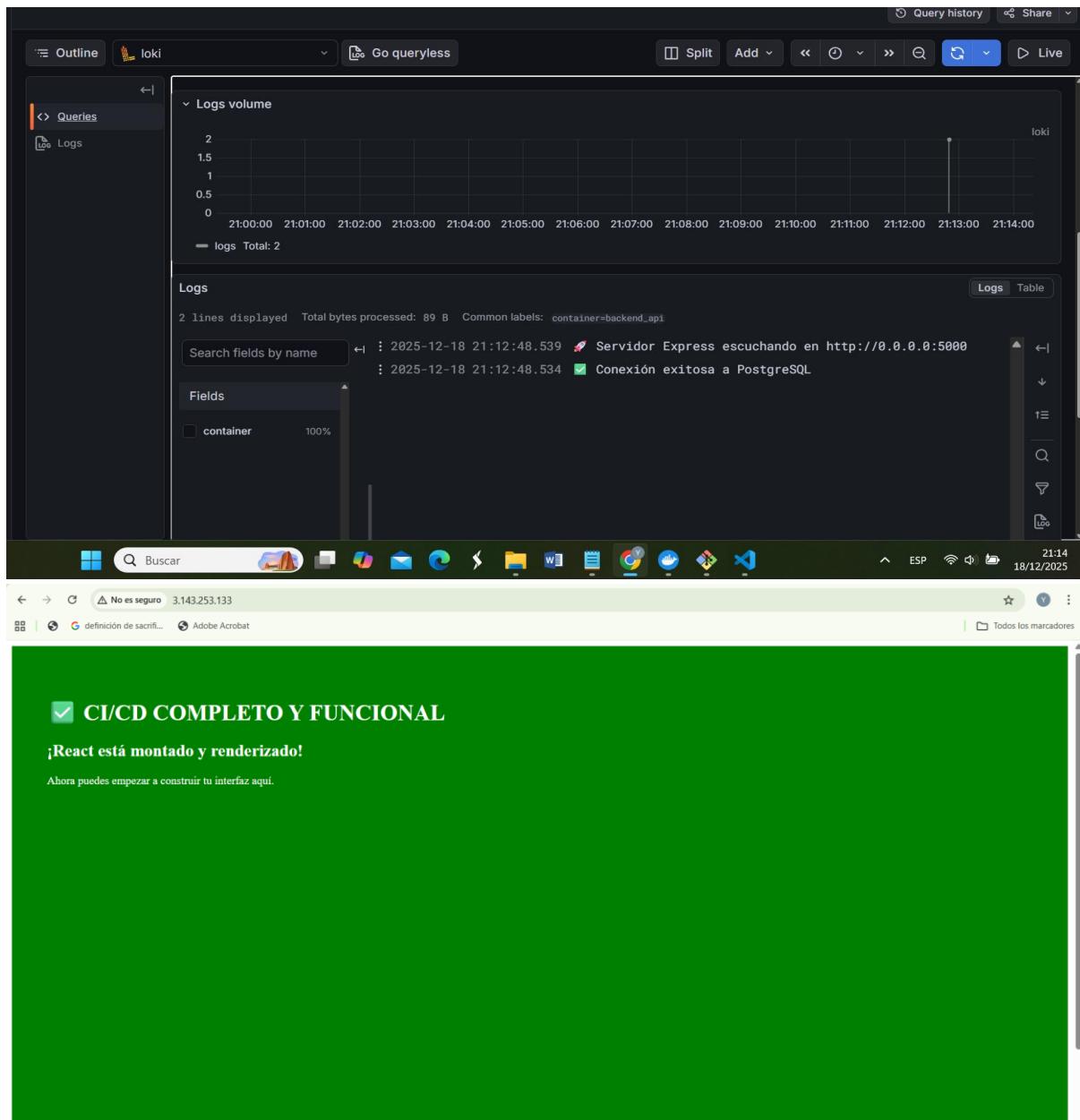
Quick CPU / Mem / Disk

Pressure	CPU Busy	Sys Load	RAM Us...	SWAP ...	Root FS...	CPU ...	RAM ...	SWA...
0.4%	1.3%	0.0%	35.9%	N/A	67.0%	2	2 GiB	SWAP Total
Mem						Root...	8 GiB	
I/O						Uptime	22.6 hours	

Basic CPU / Mem / Net / Disk

CPU Basic

Memory Basic



## 7.4. Validación de Integración y Notificaciones

# 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

Se implementó un sistema de observabilidad centralizado utilizando **Grafana Loki** y **Promtail**. Esta arquitectura permite superar las limitaciones de la terminal SSH:

- **Recolección:** Promtail actúa como agente en la EC2, recolectando los logs de `/var/lib/docker/containers` con privilegios de root para garantizar la lectura de todos los servicios.
- **Retención y Consulta:** Los logs se indexan en Loki, permitiendo realizar consultas históricas y filtrado por etiquetas (ej. `container_name="backend_api"`) desde la interfaz de Grafana Explore.
- **Beneficio:** Facilita el diagnóstico de errores en caliente sin interrumpir la ejecución de los contenedores en producción.

## 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.

Esta política se extendió también al stack de monitoreo. Los contenedores de **Prometheus**, **Grafana** y **Loki** están configurados con restart: unless-stopped. Esto asegura que, tras un reinicio del servidor físico o una falla crítica del motor de Docker, el sistema no solo recupere la aplicación, sino también la capacidad de monitorearla de inmediato.

### 8.3. Persistencia y Backups de Datos

Se utilizan **Volúmenes Nombrados** para garantizar la persistencia en tres niveles críticos:

- **postgres\_data:** Almacena la base de datos app\_db.
- **prometheus\_data:** Conserva el histórico de métricas de rendimiento.
- **grafana\_data:** Mantiene la configuración de dashboards y alertas creadas por el usuario.

Esta estrategia permite que, al ejecutar un nuevo despliegue mediante el pipeline de CI/CD (Job deploy), se puedan destruir los contenedores antiguos sin perder la integridad de la información ni las métricas históricas.

## **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:
  - o Identificar el SHA del *commit* estable previo.
  - o 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.
  - o El *pipeline* construiría y desplegaría la imagen de la versión anterior automáticamente.

**Capacidad de Auditoría para Rollback:** Gracias al monitoreo de logs implementado, antes de realizar un rollback, el equipo de operaciones puede inspeccionar los logs en Grafana para identificar si el error fue causado por el código o por una saturación de recursos de la instancia EC2, permitiendo tomar una decisión informada sobre qué versión desplegar.

## **9. Estrategia de monitoreo**

La estrategia de monitoreo se diseñó bajo el principio de **Observabilidad Completa**, dividiéndose en dos pilares: métricas de infraestructura (Prometheus) y gestión de eventos (Loki).

## 10.1. Selección de Métricas y Justificación

Se seleccionaron métricas específicas para garantizar la estabilidad de la instancia t3.micro en AWS:

- **Uso de CPU (CPU Busy):** Vital para detectar si los procesos de Node.js o el recolector de logs están saturando el procesador.
- **Utilización de Memoria RAM:** Crítica en instancias con recursos limitados para prevenir errores de *Out of Memory* (OOM) que detengan la base de datos o la API.
- **Disponibilidad de Contenedores:** Monitoreo del estado de salud (UP/DOWN) de los servicios para validar la resiliencia del sistema.

## 10.2. Configuración de Prometheus (`prometheus.yml`)

Prometheus actúa como el motor de métricas, configurado para realizar "scraping" (recolección) de datos en intervalos de 15 segundos.

- **Targets de Monitoreo:** Se configuró el archivo para extraer datos del contenedor node-exporter, el cual traduce las estadísticas del kernel de Linux de la EC2 a un formato legible para Prometheus.
- **Persistencia:** Se utiliza un volumen nombrado (`prometheus_data`) para asegurar que el histórico de rendimiento no se borre al actualizar el contenedor mediante el pipeline de CI/CD.

### **10.3. Configuración de Loki y Gestión de Logs**

Loki se implementó como un sistema de agregación de logs de bajo consumo, optimizado para no sobrecargar el almacenamiento de la EC2.

- **Flujo de Datos:** El agente promtail escanea el directorio `/var/lib/docker/containers`.
- **Privilegios Elevados:** Se configuró el acceso como `user:root` en Promtail para superar las restricciones de seguridad del sistema de archivos de Docker, permitiendo que todos los logs del backend y la base de datos lleguen a Grafana.
- **Visualización:** Los logs se etiquetan por `container_name`, permitiendo que el desarrollador filtre errores específicos del backend sin necesidad de acceder al servidor vía SSH.

## **10. 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.

## 10.1. Cumplimiento del Objetivo Principal

Se logró 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 y Visibilidad**: Se confirmó que la lógica de negocio (Backend Express y PostgreSQL) es 100% operativa. La integración de un stack de observabilidad permitió validar el funcionamiento interno de los contenedores sin depender de acceso manual por terminal.

## 10.2. 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  $\sim 20\text{ MB}$ ), 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.
- **Implementación de Observabilidad Proactiva:** Se estableció un ecosistema de monitoreo basado en **Prometheus, Grafana y Loki**. Esto permite la visualización de métricas de hardware en tiempo real y la centralización de logs de los contenedores, facilitando una gestión de incidentes basada en datos y no en suposiciones.
- **Resolución de Desafíos de Seguridad en Red:** Se aplicó el principio de mínimo privilegio en los **Security Groups** de AWS, manteniendo cerrados los puertos de Base de Datos y API al exterior, exponiendo únicamente la interfaz web y el panel de control de Grafana bajo autenticación.
- **Gestión de Agentes de Telemetría:** Se resolvió el desafío técnico de recolección de logs configurando el agente **Promtail con privilegios de root**, permitiendo la lectura segura de los archivos de log de Docker y su posterior indexación en Loki para auditoría.

En si la parte que más dificulto cuando estaba haciendo este nuevo parcial es que me aparezcan los logs del backend, tuve

que reiniciar el container del promtail y cambiar el tiempo en el que reciben los logs para que se vean y crear el dashboard, ya que intentaba lo que intentaba, no me dejaba importar el dashboard de prometheus hasta que uso el json del sitio y finalmente me salió.

Para mí es interesante como en el dashboard se puede ver el RAM, memoria, CPU, tráfico, disco que usa nuestra instancia gracias al grafana, es lo que más me intereso al hacer este parcial.

En si logre crear un workflow que funciona, crea imágenes, y hace un deploy exitoso en el sitio web con ayuda de AWS, que este monitoreando los logs y el uso del CPU/Memoria con Grafana, Prometeus y Loki y estoy contento con el proyecto.