# PRUEBA TÉCNICA DEVOPS - TCS Ecuador

**Nombre:** Cristian Fernando Idrobo Montalvo
**Repositorio:** https://github.com/cfidrobo/PruebaTCS_Idrobo_Cristian

## 1. Introducción

Se detalla el proceso seguido para dockerizar, desplegar en un proyecto en Node.js usando Terraform y configurar un pipeline CI/CD en Github Actions.

## 2. Problema Inicial

El reto inicial consiste en tomar un microservicio en Node.js, dockerizarlo, automatizar su integración y despliegue continuo mediante GitHub Actions, y desplegarla en un clúster Kubernetes local (Minikube) para pruebas y luego crear una infraestructura en un proveedor en Google Cloud Platform usando Terraform y GKE

## 3. Funcionamiento del Microservicio

El microservicio fue desarrollado utilizando Node.js con el framework Express.js, y cumple con los requisitos de seguridad, estructura de endpoint y validación de datos.

### 3.1. Archivo: app.js

Es el responsable de levantar el servidor Express, configurar dos middlewares personalizados en los cuales tenemos

- Uno para validar la API Key (2f5ae96c-b558-4c7b-a590-a501ae1c3f6c)
- Otro para validar el JWT (eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0cngiOjE3NTIyOTYxODUzNzksImlhdCI6MTc1MjI5NjE4NX0.VYaZn6iAqIZj8vgdITDYfL826WXz2ibn_sC3yksQe8A)

Nota: con el siguiente comando se genera el JWT:

```
node -e "console.log(require('jsonwebtoken').sign({ trx: Date.now() }, 'clave'))"
```

De igual manera define un middleware que restringe todos los métodos HTTP excepto POST sobre el endpoint /DevOps, y también se implementa la lógica y construye la respuesta.

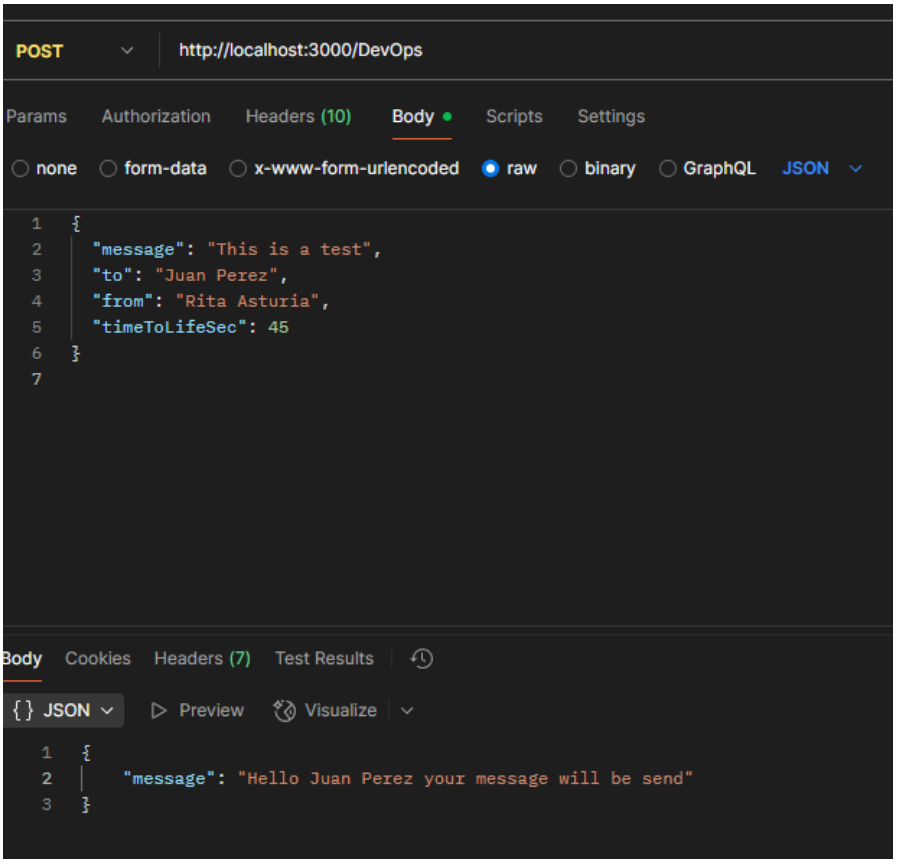
### 3.2. Archivo: app.test.js

Se utiliza el framework Jest junto con Supertest para realizar pruebas automáticas de integración sobre el endpoint /DevOps.
Las pruebas verifican: Solicitud exitosa con headers y payloads válidos, el rechazo de error en API Key, el JWT faltante, rechazo del payload, respuesta de error al usar otros métodos.
Esto con el fin de asegurar una alta cobertura de código

### 3.3. Ejecución de comandos de prueba:

```
npm install
npm test
npm start
```

### 3.4. Pruebas de microservicio con Postman




4. **Contenerización del Microservicio**

El microservicio se dockerizo utilizando una imagen multietapa basada en node:18-alpine, lo que permite optimizar el tamaño como la seguridad de la imagen.

**4.1. Dockerfile**

- **Etapa 1 builder:**

Esta etapa instala todas las dependencias, crea un usuario sin privilegios, ejecuta pruebas automáticas y revisión de código estático, sirve para la validación y preparación del entorno.

- **Etapa 2 producción:**

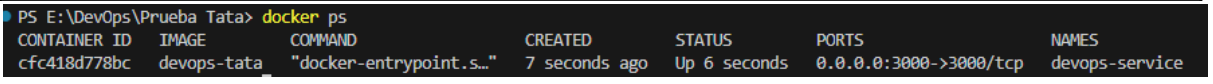
En esta etapa igual utilizamos la imagen de node:18-alpine, se instala las dependencias de producción omitiendo las de dev, se usa el usuario sin privilegios, se expone el puerto y la ruta del comando de inicio.

```
# —— ETAPA 1: Build & Test ————————————————————————————————

FROM node:18-alpine AS builder

# Crear usuario sin privilegios
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /usr/src/app

# Copiar archivos para instalar dependencias
COPY package*.json ./

# Instala todas las dependencias (incluyendo devDependencies)
```

```
RUN npm ci

# Copiar el resto del código fuente
COPY . .

# Ejecutar pruebas automáticas
RUN npm test

# Ejecutar revisión estática (linter)
RUN npm run lint || true

# ——— ETAPA 2: Producción
————————————————————————————————————————————————————————————————————
FROM node:18-alpine

# Crear usuario sin privilegios
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /usr/src/app

# Solo dependencias de producción
COPY package*.json ./
RUN npm ci --omit=dev

# Copiar el código y cambiar propietario
COPY --from=builder /usr/src/app ./
RUN chown -R appuser:appgroup /usr/src/app

# Usar usuario sin privilegios
USER appuser

# Variables de entorno
ENV NODE_ENV=production
ENV PORT=3000
EXPOSE ${PORT}

# Comando de inicio
CMD ["node", "src/app.js"]
```

**Comandos para pruebas:**

```
docker build -t devops-tata .    #Dockeriza
docker run -p 3000:3000 --env-file .env --name devops-service devops-tata        #Levanta el contenedor localmente
```




### 5. Pipeline CI/CD con GitHub Actions

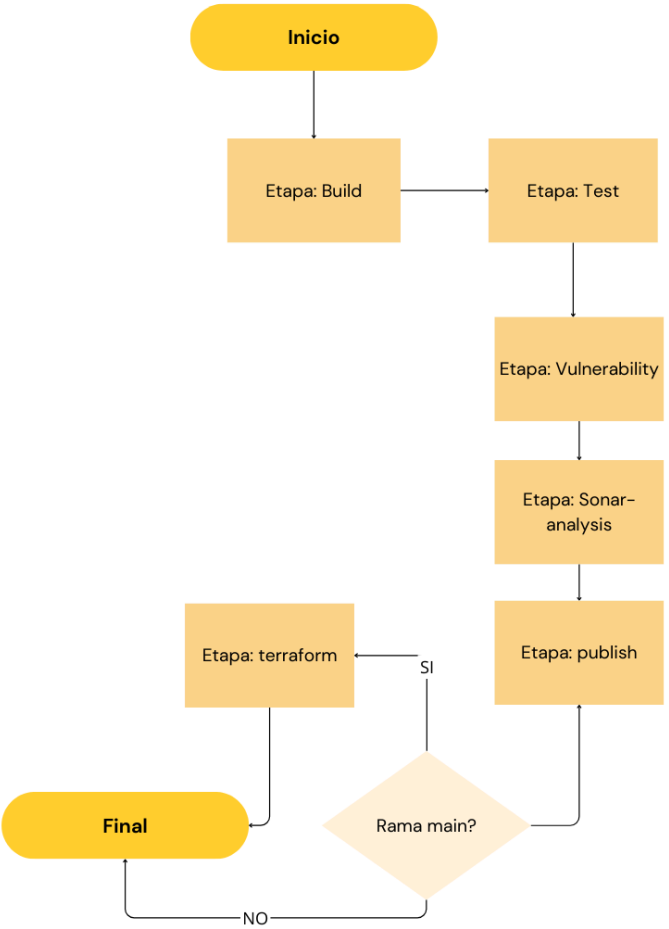
El pipeline de este proyecto está estructurado en 6 etapas “stages” principales que se ejecutan de forma secuencial para garantizar un flujo de trabajo robusto y controlado. Estas

etapas son:

| Etapa | Descripción |
|---|---|
| build | Instala de forma limpia las dependencias (npm ci) y ejecuta el paso de build si existe (npm run build). Prepara el artefacto de la aplicación para las siguientes fases. |
| test | Ejecuta las pruebas unitarias con Jest bajo un entorno de test (NODE_ENV: test), asegurando que el código compila y que los casos de prueba pasan antes de avanzar. |
| vulnerability | Realiza un escaneo automático de vulnerabilidades en las dependencias (npm audit --audit-level=moderate) y guarda el reporte en JSON como artefacto descargable para revisión. |
| sonar-analysis | Integra SonarCloud para análisis estático de código y medición de cobertura. Usa sonar-scanner con los parámetros de organización, proyecto y credenciales definidas en secretos. |
| publish | Construye la imagen Docker (multitag: :sha8 y :latest) y la publica en GitHub Container Registry. Garantiza que cada build genere un tag único (SHA) y un tag mutable (latest). |
| terraform | Auténtica con GCP, inicializa Terraform, genera un plan e implementa automáticamente los recursos si el commit proviene de la rama main. Despliega en GKE el microservicio usando la imagen publicada, junto con el balanceador de carga, secretos y configuración definida por IaC. |

**Nota:** Antes de incluir terraform al flujo del CI/CD, se realizó pruebas locales con Kubernetes usando Mini Kube para el despliegue de la aplicación y verificar su correcto funcionamiento.

**5.1. Diagrama de flujo del Pipeline CI/CD**




**5.2. Etapas del Pipeline:**

Después de hacer un push con el microservicio configuramos lo siguiente, se deben incluir los secretos en Github, incluyendo las credenciales de GCP, SonarCloud, y el token de Github (GHCR).


- **Etapa BUILD**

En esta etapa se instala dependencias exactamente según package-lock.json y ejecuta el script de build si existe, su propósito es preparar artefactos (transpiled code, bundles) para

pruebas y análisis, y como requisito previo se necesita el package-lock.json, una vez pasado el job el código compilado se queda en el workspace de actions.

- **Etapa TEST**

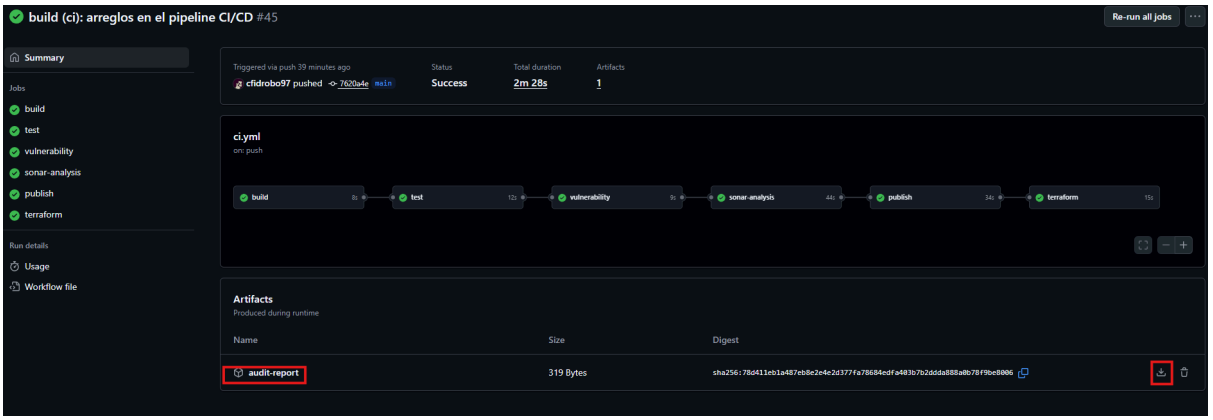
En esta etapa se ejecutan las pruebas unitarias con Jest en modo “test”, para asegurar que el código cumple los casos de prueba antes de continuar.
Como requisito previo necesitamos la configuración de Jest en package.json, como resultado de tests si es exitoso refleja reportes de cobertura en la consola.

```
"scripts": {
  "start": "node src/app.js",
  "test": "jest --coverage"
},
```

```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 app.js   |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        1.211 s
Ran all test suites.
```

- **Etapa VULNERABILITY SCAN**

En esta etapa se escanea vulnerabilidades de dependencias con npm audit y guarda un JSON, con el propósito de detectar posibles riesgos de seguridad en librerías antes de la publicación de la imagen, como requisito tener npm instalado, y su salida será un artefacto audit-report.json descargable desde la UI de Actions.

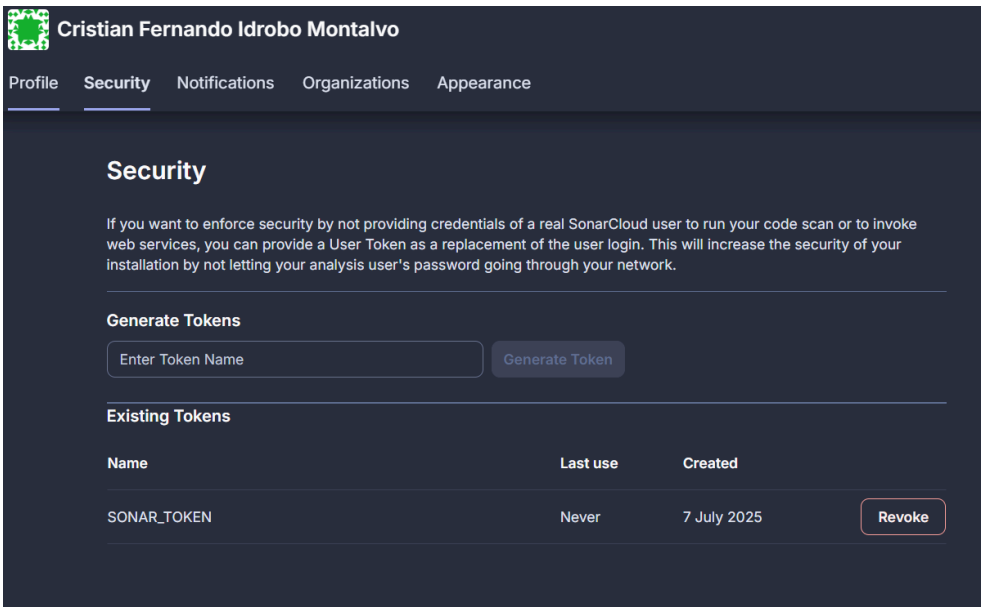



- **Etapa SONAR ANALYSIS**

En esta etapa se ejecuta un análisis estático de código y cobertura en SonarCloud, con el propósito de verificar calidad del código, “code smells”, “bugs” y cobertura, según el Quality Gate configurado, como requisitos previos necesitamos:

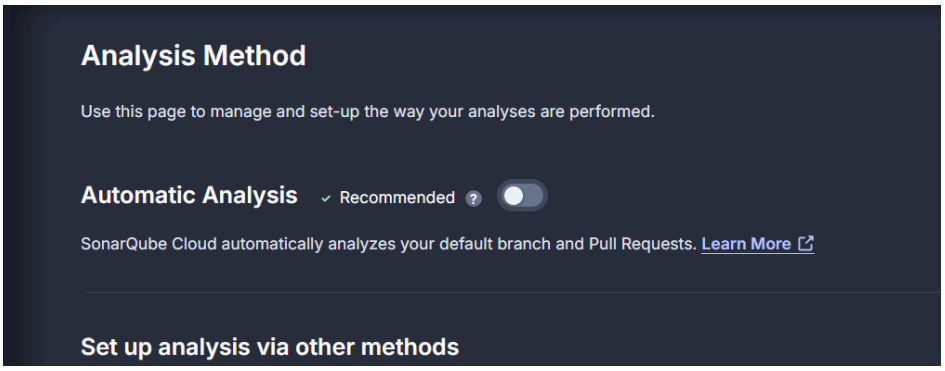
**Configurar en GitHub Secrets:**

- SONAR_TOKEN (token de SonarCloud)

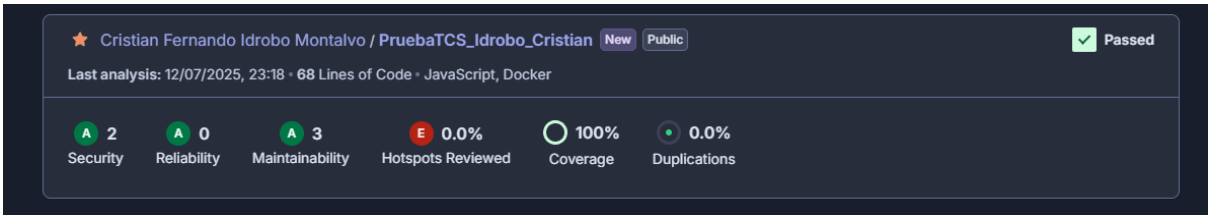



- SONAR_ORG (organización en SonarCloud)
- SONAR_PROJECT_KEY (clave de proyecto)

Igualmente se debe deshabilitar la opción de método automático de análisis para que siempre se analice cuando el pipeline se ejecute.



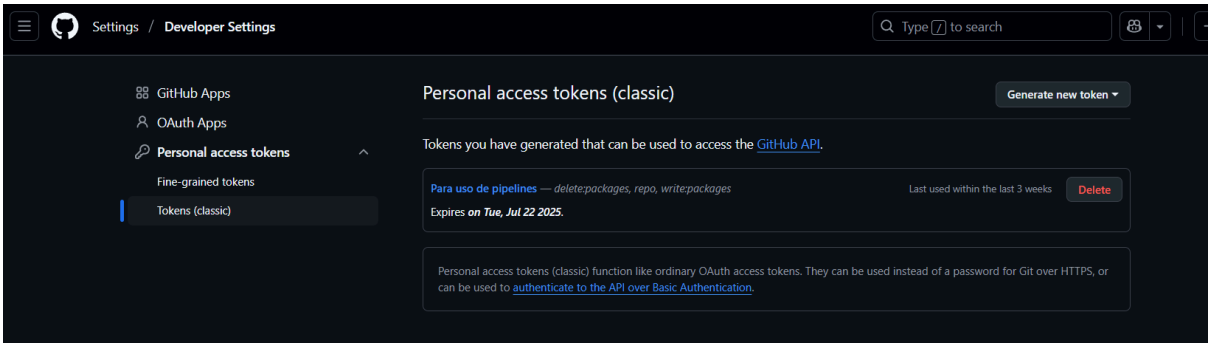

Como salida se puede observar los dashboards de análisis en SonarCloud.



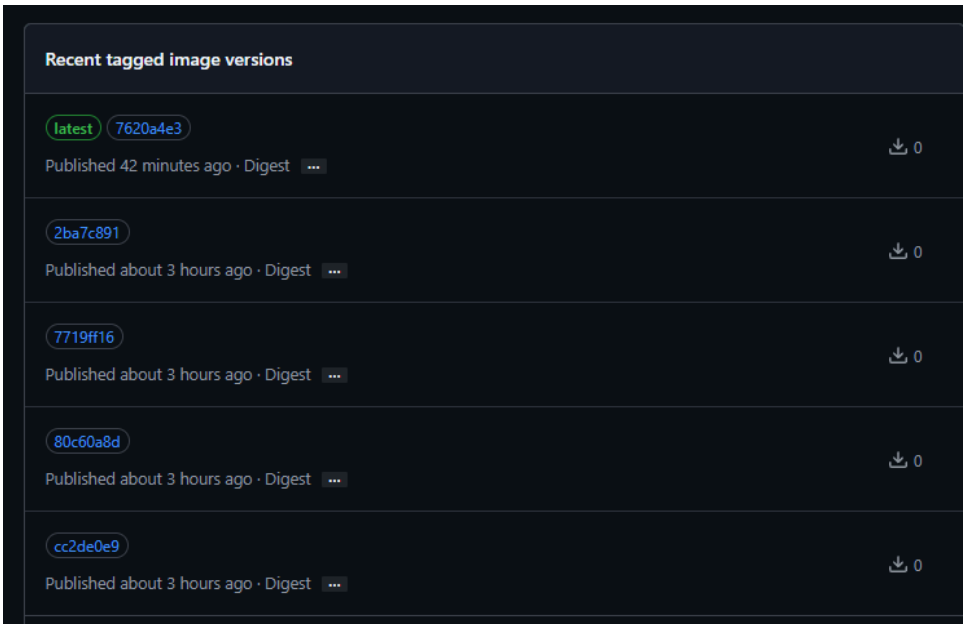

- **Etapa PUBLISH**

En esta etapa se configura Docker Buildx, se loguea en GitHub Container Registry, Determina un tag único (SHA de 8 caracteres) para los commits, se construye la imagen y push con un tag (sha8) y (latest), para tener un histórico de imágenes que por commits y una latest haciendo referencia al último push que está bien.

Su propósito es generar y publicar la imagen Docker para su posterior despliegue, como requisitos previos debemos configurar un GHCR_PAT el cual es el personal access token con permisos de escritura de paquetes.




Como salida las imágenes estarán disponibles en los paquetes del repositorio.



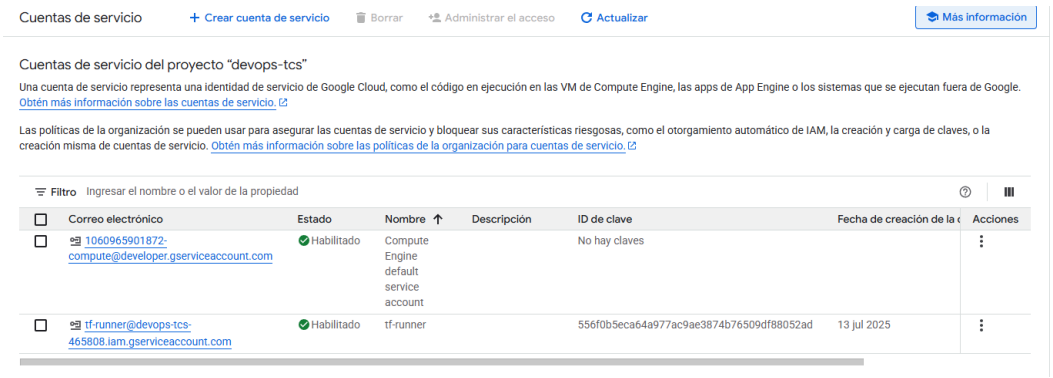

- **Etapa TERRAFORM**

En esta etapa se realiza el despliegue automatizado de la infraestructura y aplicación en Google Kubernetes Engine (GKE) mediante IaC usando Terraform.
Su propósito es automatizar el aprovisionamiento de infraestructura y el despliegue del microservicio ya empacado como imagen Docker en un clúster de Kubernetes administrado por GCP, usando terraform plan y terraform apply.

Como requisitos previos:

- Debe completar la etapa publish y estar la imagen creada en packages.
- Crear y configurar un bucket de Cloud Storage para almacenar el estado de terraform.


- Una cuenta de servicio y asignar roles porque Github Action lo utiliza para ejecutar comandos de Terraform.




- Se debe configurar los secretos los cuales son:
    - GCP_CREDENTIALS: JSON con credenciales de servicio GCP.
    - GCP_PROJECT: ID del proyecto en Google Cloud.
    - GCP_REGION: Región donde se desplegará GKE.
- Incluir en la carpeta terraform los archivos correspondientes:
    - backend.tf: Define el backend remoto para almacenar el estado de Terraform en un bucket de GCS.
    - provider.tf: Configura el proveedor de Google Cloud y obtiene la autenticación del cliente actual.
    - variables.tf: Define las variables requeridas para el despliegue (project, región, image).
    - gke.tf: Crea el clúster GKE (google_container_cluster) y un pool de nodos (node_pool).
    - k8s.tf: Despliega todos los recursos de Kubernetes (Deployment, Service, ConfigMap, Secret, Ingress).

En esta etapa se automatiza el despliegue de toda la infraestructura y del microservicio en Google Kubernetes Engine (GKE) mediante Terraform. Primero se clona el repositorio y se instala Terraform. Luego, utilizando credenciales seguras (GCP_CREDENTIALS), se autentica en GCP, se inicializa el backend remoto en Google Cloud Storage y se genera un plan de despliegue (terraform plan) que, si la rama es main, se aplica automáticamente (terraform apply).

Durante este proceso se crean y configuran los siguientes recursos:

- Clúster GKE y Node Pool: Se despliega un clúster gestionado con un pool de 2 nodos e2-medium para alojar los pods.
- Dirección IP estática: Se reserva una IP regional para exponer el servicio externamente. (35.193.20.145)
- ConfigMap y Secret: Se definen variables de entorno y credenciales necesarias para la ejecución segura del microservicio.
- Deployment y Service: Se despliega la app en 2 réplicas con sondas de salud, y se expone por medio de un LoadBalancer.
- Ingress HTTP: Se utiliza un controlador GCE que enruta el tráfico hacia el servicio usando la IP fija.

### 5.3. Workflow de GitHub Actions

```yaml
name: CI Pipeline

on:
  push:

env:
  REGISTRY: ghcr.io
  IMAGE_OWNER: ${{ github.repository_owner }}
  IMAGE_REPO: ${{ github.event.repository.name }}

jobs:
  # 1) BUILD: instalar deps y (opcional) build
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Install dependencies
        run: npm ci
      - name: Build (if defined)
        run: npm run build || echo "No build step"

  # 2) TEST: correr unit tests
  test:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Install deps
        run: npm ci
      - name: Run tests
        env:
          NODE_ENV: test
        run: npm test

  # 3) VULNERABILITY SCAN: npm audit
  vulnerability:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Audit dependencies and save report
        run: |
          npm ci
          npm audit --audit-level=moderate --json > audit-report.json || true
```

```
    - uses: actions/upload-artifact@v4
      with:
        name: audit-report
        path: audit-report.json

# 4) SONAR ANALYSIS: SonarCloud
sonar-analysis:
  needs: vulnerability
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
    - name: Install deps
      run: npm ci
    - name: Run tests with coverage
      env:
        NODE_ENV: test
      run: npm test
    - name: SonarCloud Scan
      env:
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
      run: |
        npx sonar-scanner \
          -Dsonar.organization=${{ secrets.SONAR_ORG }} \
          -Dsonar.projectKey=${{ secrets.SONAR_PROJECT_KEY }} \
          -Dsonar.host.url=https://sonarcloud.io \
          -Dsonar.login=$SONAR_TOKEN \
          -Dsonar.sources=. \
          -Dsonar.tests=. \
          -Dsonar.test.inclusions="**/*.test.js" \
          -Dsonar.javascript.lcov.reportPaths=coverage/lcov.info


# 5) BUILD & PUSH: image tagged con SHA corta y 'latest'

publish:
  needs: sonar-analysis
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - name: Setup Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Login to GHCR
      uses: docker/login-action@v3
      with:
        registry: ghcr.io
        username: ${{ github.actor }}
        password: ${{ secrets.GHCR_PAT }}
    - name: Determine image tag
      id: tag
      run: echo "sha=${GITHUB_SHA::8}" >> $GITHUB_OUTPUT
```

```
    - name: Build & push to GHCR
      run: |
        REPO_LOWER=$(echo "${IMAGE_REPO}" | tr '[:upper:]' '[:lower:]')
        IMAGE=ghcr.io/${IMAGE_OWNER}/$REPO_LOWER
        docker build -t $IMAGE:${{ steps.tag.outputs.sha }} -t $IMAGE:latest .
        docker push $IMAGE:${{ steps.tag.outputs.sha }}
        docker push $IMAGE:latest

  # 6) Terraform
  terraform:
    needs: publish
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

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

      - name: Authenticate GCP
        uses: google-github-actions/auth@v1
        with:
          credentials_json: ${{ secrets.GCP_CREDENTIALS }}

      - name: Terraform Init
        run: terraform -chdir=terraform init

      - name: Terraform Plan
        run: |
          terraform -chdir=terraform plan \
            -var="project=${{ secrets.GCP_PROJECT }}" \
            -var="region=${{ secrets.GCP_REGION }}" \
            -var="image=ghcr.io/${IMAGE_OWNER}/${IMAGE_REPO,,}:latest" \
            -out=tfplan

      - name: Terraform Apply
        if: github.ref_name == 'main'
        run: terraform -chdir=terraform apply -auto-approve tfplan
```

## 6. Pasos previos antes de ejecutar el pipeline en Github

### 6.1. Autenticarse en CLI de gcloud

```
gcloud auth login
```

### 6.2. Habilitar el container registry al proyecto

Esto configura el proyecto activo y habilita las APIs necesarias

```
gcloud config set project devops-tcs-465808
gcloud services enable containerregistry.googleapis.com
gcloud projects add-iam-policy-binding devops-tcs-465808
--member="serviceAccount:tf-runner@devops-tcs-465808.iam.gserviceaccount.com"
--role="roles/storage.admin"
```

```
gcloud services list --enabled | Select-String "containerregistry.googleapis.com"
```

### 6.3. Otorgar el rol iam.serviceAccountUser al Service Account

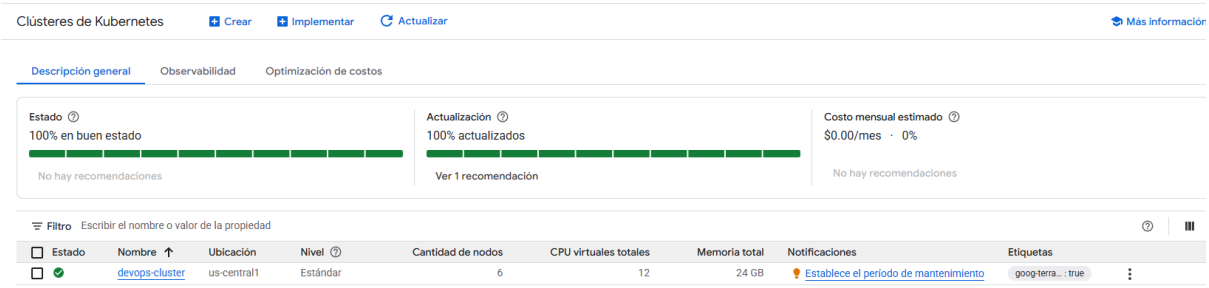
Este comando otorga a la cuenta de servicio, acceso total al bucket para manejar el estado de Terraform.

```
gcloud iam service-accounts add-iam-policy-binding
1060965901872-compute@developer.gserviceaccount.com
--member="serviceAccount:tf-runner@devops-tcs-465808.iam.gserviceaccount.com"
--role="roles/iam.serviceAccountUser"  --project=devops-tcs-465808
```

## 7. Pruebas de Funcionamiento

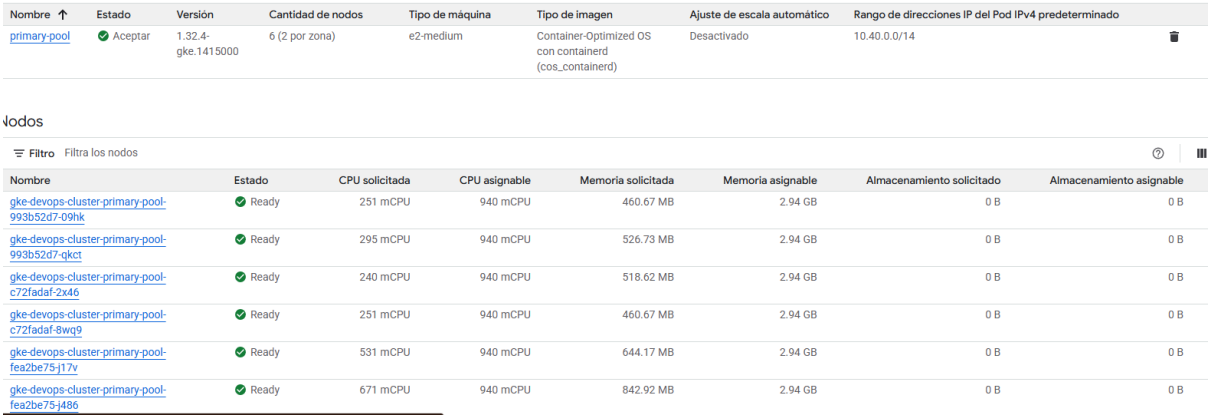
Tras completar el despliegue del microservicio en Google Kubernetes Engine (GKE) mediante Terraform, se realizaron pruebas para validar su correcto funcionamiento y accesibilidad a través del balanceador de carga HTTP.

### 7.1. Validación del clúster GKE




Se verificó que el clúster devops-cluster se encuentra en estado 100% en buen estado y actualizado en la región us-central1, lo que garantiza una infraestructura saludable y lista para recibir tráfico.

### 7.2. Verificación del balanceador de carga



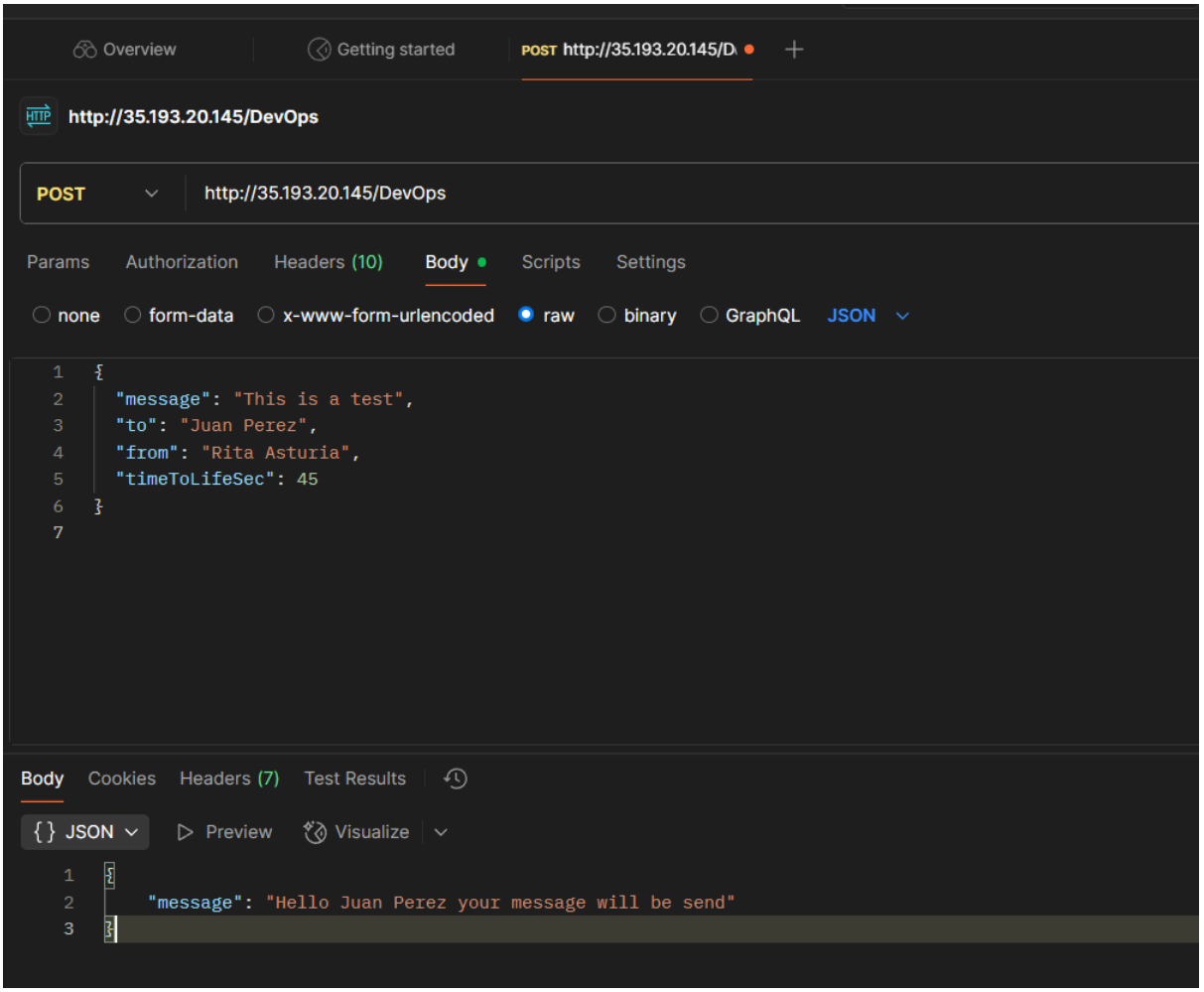

Se validó que el balanceador de carga asignado al Ingress está activo, utilizando la IP pública 35.193.20.145, y que enruta correctamente el tráfico hacia las instancias de los nodos del pool desplegado por Terraform.

### 7.5. Verificación del estado de los pods

```
PS E:\DevOps\PruebaTCS_Idrobo_Cristian> kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
gke-devops-cluster-primary-pool-993b52d7-09hk   Ready    <none>   13h   v1.32.4-gke.1415000
gke-devops-cluster-primary-pool-993b52d7-qkct   Ready    <none>   13h   v1.32.4-gke.1415000
gke-devops-cluster-primary-pool-c72fadaf-2x46   Ready    <none>   13h   v1.32.4-gke.1415000
gke-devops-cluster-primary-pool-c72fadaf-8wq9   Ready    <none>   13h   v1.32.4-gke.1415000
gke-devops-cluster-primary-pool-fea2be75-j17v   Ready    <none>   13h   v1.32.4-gke.1415000
gke-devops-cluster-primary-pool-fea2be75-j486   Ready    <none>   13h   v1.32.4-gke.1415000
PS E:\DevOps\PruebaTCS_Idrobo_Cristian> kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
devops-service-7778c696f6-7kc7h   1/1     Running   0          6h21m
devops-service-7778c696f6-cz5xf   1/1     Running   0          6h21m
PS E:\DevOps\PruebaTCS_Idrobo_Cristian> kubectl get svc
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)        AGE
devops-service-svc  LoadBalancer   34.118.230.166   35.193.20.145   80:32562/TCP   9h
kubernetes          ClusterIP      34.118.224.1     <none>          443/TCP        19h
PS E:\DevOps\PruebaTCS_Idrobo_Cristian>
```

### 7.4. Test de la API desplegada en Postman y Máquina virtual Linux

Se ejecutó una prueba de tipo POST hacia la ruta /DevOps utilizando Postman, enviando un cuerpo JSON. La respuesta recibida fue satisfactoria, indicando que el microservicio está funcionando correctamente y accesible desde el exterior.



**Pruebas máquina virtual**

```
curl -X POST -H "X-Parse-REST-API-Key: 2f5ae96c-b558-4c7b-a590-a501ae1c3f6c" -H "X-JWT-KWY: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0cngiOjE3NTIyOTYxODUzNzksImlhdCI6MTc1MjI5NjE4NX0.VYaZn6iAqIZj8vgdlTDYfL826WXz2ibn_sC3yksQe8A" -H "Content-Type: application/json" -d '{"message":"This is a test","to":"Juan Perez","from":"Rita Asturia","timeToLifeSec":45}' http://35.193.20.145/DevOps
```