KUBERNETES CON LOAD BALANCER EN AWS

Hamza Akdi

20/12/2025

Índice:

Introducción

Kubernetes es un sistema que gestiona automáticamente aplicaciones en múltiples máquinas. Sirve para desplegar, escalar, supervisar y recuperar aplicaciones sin intervención manual.

K3d es una versión ligera de Kubernetes (50 MB vs 500 MB). Funciona increíblemente bien en WSL2 sin Docker ni complicaciones.

POD es la unidad mínima de Kubernetes, un contenedor ejecutándose. Tiene IP propia, es efímero (desaparece cuando muere), aislado.

Deployment es un controlador que crea y mantiene múltiples pods idénticos, su responsabilidad es que, si un pod muere, crea uno nuevo. Mantiene la cantidad especificada.

Service es un intermediario que da una IP estable para acceder a pods. Los pods tienen IPs que cambian. Service proporciona una IP que nunca cambia.

• **Load Balancer** es un tipo de Service que distribuye solicitudes entre múltiples pods. El LoadBalancer elige qué Pod atender (round robin, menos conexiones, etc.) va a ser atendido el cliente. Si un pod cae, otros atienden. La carga se distribuye.

• **Namespace** es un espacio aislado dentro de Kubernetes (como carpetas). Separar desarrollo/producción, equipos, versiones diferentes.

• **ConfigMap** almacena configuración (URLs, puertos, etc.). Cambias config sin recompilar ni redeploy.

• **Secret** es un ConfigMap pero para datos sensibles (contraseñas, tokens). Está cifrado, no es visible en logs.

• **Labels** son etiquetas para identificar objetos (app: web-app, version: 1.0).

• **Selectors** son formas de filtrar objetos por sus labels.

• **Escalado horizontal** es aumentar número de pods (de 3 a 5). La carga se distribuye entre más máquinas, Kubernetes lo hace automáticamente.

• **Escalado vertical** es aumentar recursos de una máquina (2GB → 8GB RAM), tiene límites.

• **Session Affinity** mantiene al cliente en el mismo pod si ya está conectado.

• **Port-forward** es un túnel que expone puertos de Kubernetes en tu máquina local. Su uso es solo para desarrollo/testing.

• **Túnel SSH** es una conexión SSH que redirige puertos desde máquina remota a local. EC2 accede a Kubernetes sin ruta de red directa.

• **Ingress** es un objeto que gestiona acceso externo a servicios (dominios, HTTPS, routing). Su ventaja sobre port-forward es que tiene nombres reales (miapp.com), HTTPS, y su uso es profesional.

• **Persistent Volume** es almacenamiento que persiste más allá de la vida del pod. Si un pod muere, sus datos desaparecen. PV los guarda en almacenamiento externo.

• **StateFulset** es como Deployment pero para aplicaciones con estado (BD, colas, etc.). Los pods tienen identidad. El orden importa.

• **Job** ejecuta una tarea UNA VEZ y termina (no indefinido como Deployment). Se usa en migraciones, backups, procesamiento por lotes.

• **CronJob** es un Job que se ejecuta en un horario específico.

• **Yaml** es el formato para describir objetos de Kubernetes de forma legible.

Arquitectura

1. Cliente
2. PORT-FORWARD / INGRESS
3. SERVICE (IP estable)
4. LOAD BALANCER (elige pod)
5. DEPLOYMENT (3 replicas)
6. PODS (tu aplicación)

Flujo de una petición

1. Cliente solicita http://localhost:8080
2. Port-forward escucha, redirige a Service
3. Service elige qué pod (LoadBalancer)
4. Pod ejecuta tu código (Flask, Node, Java, etc.)
5. Respuesta vuelve al cliente.

Si un pod muere, Deployment crea uno nuevo automáticamente.

Objetivos

- Instalar Kubernetes local directamente en WSL2 (sin Docker, sin dependencias)
- Desplegar múltiples replicas de una aplicación en Kubernetes
- Configurar un Load Balancer interno para distribuir tráfico
- Conectar tu cluster local de Kubernetes con instancias EC2 en AWS
- Monitorear el tráfico y verificar el balanceo de carga
- Escalar aplicaciones dinámicamente en Kubernetes

Requisitos Previos

• Windows 11 con WSL2 habilitado

• Ubuntu 22.04 o superior en WSL2

• kubectl instalado • k3s instalado (Kubernetes ligero sin Docker)

• Cuenta AWS Free Tier

• Acceso SSH a instancias EC2

• Terminal en Windows o WSL

• Al menos 2 GB RAM disponibles

Requisitos de AWS

• Acceso a EC2 (crear/gestionar instancias)

• Acceso a Security Groups

• Acceso a Key Pairs

1. Preparación del entorno local en WSL2

En esta fase preparamos el sistema operativo Ubuntu en WSL2 para ejecutar **Kubernetes** sin necesidad de Docker, utilizando **k3s por su ligereza.** Es perfecto para desarrollo y testing.

1.1 Instalación de k3s y kubectl

Se ha instalado k3s deshabilitando systemd para mejorar la compatibilidad con WSL2 y se ha configurado kubectl para la gestión del clúster.

Proceso realizado:

1. Actualización del sistema.
2. Instalación del binario de k3s.
3. Configuración de permisos para el archivo k3s.yaml.

Instalación k3s SIN systemd.

Ejecutamos el instalador configurando el modo de permisos 644 para que nuestro usuario pueda leer la configuración del clúster sin problemas de permisos constantes.

Comando: curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" sh -

```
hamza@A6Alumno08:~$ curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" sh -
[INFO]  Finding release for channel stable
[INFO]  Using v1.34.3+k3s1 as release
[INFO]  Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.34.3+k3s1/sha256s
um-amd64.txt
[INFO]  Downloading binary https://github.com/k3s-io/k3s/releases/download/v1.34.3+k3s1/k3s
[INFO]  Verifying binary download
[INFO]  Installing k3s to /usr/local/bin/k3s
[INFO]  Skipping installation of SELinux RPM
[INFO]  Creating /usr/local/bin/kubectl symlink to k3s
[INFO]  Creating /usr/local/bin/crictl symlink to k3s
[INFO]  Creating /usr/local/bin/ctr symlink to k3s
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s.service
[INFO]  systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system
/k3s.service.
[INFO]  Host iptables-save/iptables-restore tools not found
[INFO]  Host ip6tables-save/ip6tables-restore tools not found
[INFO]  systemd: Starting k3s
```

Verificación k3s instalado

Comando: sudo k3s --version

```
hamza@A6Alumno08:~$ sudo k3s --version
k3s version v1.34.3+k3s1 (48ffa7b6)
go version go1.24.11
```

Iniciación de k3s

Esperaremos 10 segundos a que inicie.

Comando: sudo k3s server & sleep 10

```
hamza@A6Alumno08:~$ sudo k3s server & sleep 10
[1] 7781
INFO[0000] Starting k3s v1.34.3+k3s1 (48ffa7b6)
INFO[0000] Configuring sqlite3 database connection pooling: maxIdleConns=2, maxOpenConns=0, c
onnMaxLifetime=0s
INFO[0000] Configuring database table schema and indexes, this may take a moment...
INFO[0000] Database tables and indexes are up to date
INFO[0000] Kine available at unix://kine.sock
INFO[0000] Reconciling bootstrap data between datastore and disk
INFO[0000] Password verified locally for node a6alumno08
INFO[0000] certificate CN=a6alumno08 signed by CN=k3s-server-ca@1768294041: notBefore=2026-01
-13 08:47:21 +0000 UTC notAfter=2027-01-13 08:56:12 +0000 UTC
INFO[0000] Module overlay was already loaded
INFO[0000] Module nf_conntrack was already loaded
INFO[0000] Module br_netfilter was already loaded
```

Verificación de que está corriendo

Comando: sudo k3s kubectl get nodes

```
hamza@A6Alumno08:~$ sudo k3s kubectl get nodes
NAME         STATUS   ROLES           AGE     VERSION
a6alumno08   Ready    control-plane   8m17s   v1.34.3+k3s1
```

1.2 Instalación de kubectl localmente

Descargar kubectl

Aunque **k3s** incluye **kubectl**, instalamos la versión **standalone** para mayor comodidad y configuramos el acceso copiando el archivo k3s.yaml a la carpeta **$HOME/.kube/config**. Esto permite administrar el clúster sin usar sudo constantemente.

Comandos:

curl -LO "https://dl.k8s.io/release/$(curl -L –s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

sudo chmod +x kubectl

sudo mv kubectl /usr/local/bin/

```
hamza@A6Alumno08:~$ curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
curl: (2) no URL specified
curl: try 'curl --help' or 'curl --manual' for more information
-bash: https://dl.k8s.io/release/stable.txt: No such file or directory
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0    629      0 --:--:-- --:--:-- --:--:--   630
100   238  100   238    0     0    529      0 --:--:-- --:--:-- --:--:--   529
hamza@A6Alumno08:~$ sudo chmod +x kubectl
hamza@A6Alumno08:~$ sudo mv kubectl /usr/local/bin/
```

Verificación:

Comandos: kubectl version --client

```
hamza@A6Alumno08:~$ kubectl version --client
/usr/local/bin/kubectl: line 1: syntax error near unexpected token '<'
/usr/local/bin/kubectl: line 1: '<?xml version='1.0' encoding='UTF-8'?><Error><Code>NoSuchKey
</Code><Message>The specified key does not exist.</Message><Details>No such object: 767373bbd
cb8270361b96548387bf2a9ad0d48758c35/release/bin/linux/amd64/kubectl</Details></Error>'
```

Configuración de kubeconfig para acceder sin sudo

Configuramos el acceso al clúster copiando el archivo de configuración a la carpeta home del usuario. Esto permite que el comando kubectl sepa a qué clúster debe conectarse.

Comandos: mkdir -p $HOME/.kube

sudo cp /etc/rancher/k3s/k3s.yaml $HOME/.kube/config

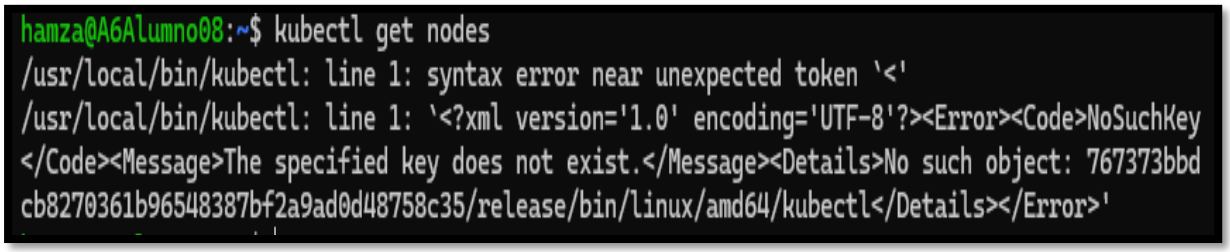
sudo chown $(id -u):$(id -g) $HOME/.kube/config

sudo chmod 600 $HOME/.kube/config

```
hamza@A6Alumno08:~$  mkdir -p $HOME/.kube
hamza@A6Alumno08:~$ sudo cp /etc/rancher/k3s/k3s.yaml $HOME/.kube/config
hamza@A6Alumno08:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
hamza@A6Alumno08:~$  sudo chmod 600 $HOME/.kube/config
```

Verificación de que funciona sin sudo:

Comando: kubectl get nodes

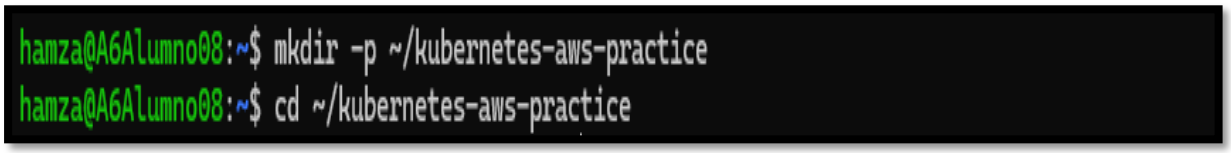


1.3 Crear directorio de trabajo

Se crea la estructura de carpetas **~/kubernetes-aws-practice** para organizar los ficheros del proyecto.

Comandos: mkdir -p ~/kubernetes-aws-practice

cd ~/kubernetes-aws-practice



2. Creación de aplicación simple para Kubernetes

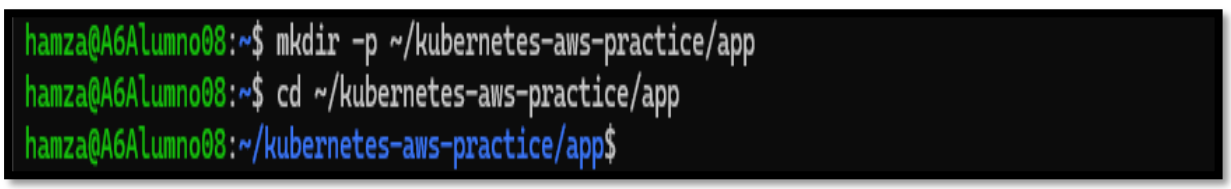
Desarrollamos una aplicación web ligera usando Python y Flask. La clave aquí es que la aplicación consulta su propio nombre de Pod (hostname). Esto nos permitirá ver visualmente cómo el balanceador nos envía a diferentes contenedores en cada clic.

2.1 Creación de carpeta para la aplicación

Comandos:

mkdir -p ~/kubernetes-aws-practice/app

cd ~/kubernetes-aws-practice/app



2.2 – Creación de aplicación Python con Flask

El script de Python utiliza variables de entorno inyectadas por Kubernetes (POD_NAME, POD_NAMESPACE) y expone endpoints / (web) y /pod-info (API JSON).

Comandos:

cat > app.py << 'EOF' #!/usr/bin/env python3 from flask import Flask, jsonify, send_from_directory import os import socket from datetime import datetime import sys app = Flask(__name__) # Variables de entorno inyectadas por Kubernetes POD_NAME = os.getenv('POD_NAME', 'Unknown Pod') POD_NAMESPACE = os.getenv('POD_NAMESPACE', 'default') @app.route('/') def index(): return send_from_directory('.', 'index.html') @app.route('/pod-info') def pod_info(): return jsonify({ 'pod_name': POD_NAME, 'namespace': POD_NAMESPACE, 'hostname': socket.gethostname(), 'timestamp': datetime.now().isoformat() }) @app.route('/health') def health(): return jsonify({'status': 'healthy', 'pod': POD_NAME}), 200 if __name__ == '__main__': print(f"[{POD_NAME}] Iniciando servidor Flask...", file=sys.stderr) app.run(host='0.0.0.0', port=5000, debug=False) EOF sudo chmod +x app.py

```
hamza@A6Alumno08:~$ cat > app.py << 'EOF'
#!/usr/bin/env python3
from flask import Flask, jsonify, send_from_directory
import os
import socket
from datetime import datetime
import sys
app = Flask(__name__)
# Variables de entorno inyectadas por Kubernetes
POD_NAME = os.getenv('POD_NAME', 'Unknown Pod')
POD_NAMESPACE = os.getenv('POD_NAMESPACE', 'default')
@app.route('/')
def index():
 return send_from_directory('.', 'index.html')
@app.route('/pod-info')
def pod_info():
 return jsonify({
 'pod_name': POD_NAME,
 'namespace': POD_NAMESPACE,
 'hostname': socket.gethostname(),
 'timestamp': datetime.now().isoformat()
 })
@app.route('/health')
def health():
 return jsonify({'status': 'healthy', 'pod': POD_NAME}), 200
if __name__ == '__main__':
sudo chmod +x app.py.0', port=5000, debug=False).", file=sys.stderr)
```

2.3 Crear archivo HTML

Se crea un archivo HTML con estilos CSS que consume la **API /pod-info** para mostrar dinámicamente qué pod está respondiendo.

```
hamza@A6Alumno08:~/kubernetes-aws-practice/app$ cat > index.html << 'EOF'
<!DOCTYPE html>
<html>
<head>
 <title>Kubernetes Load Balancer</title>
 <style>
 body {
 font-family: Arial, sans-serif;
 display: flex;
 justify-content: center;
 align-items: center;
 min-height: 100vh;
 margin: 0;
 background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
 }
 .container {
 background: white;
 padding: 50px;
 border-radius: 10px;
 box-shadow: 0 10px 25px rgba(0,0,0,0.2);
 text-align: center;
 max-width: 500px;
 }
 h1 {
 color: #667eea;
 margin: 0 0 30px 0;
 }
 .info {
 background: #f0f0f0;
 padding: 20px;
EOFtml>pt>rror('Error:', e);d-name').textContent = ' Error';name;lizar Ahora</button>
```

Creación de fichero requirements.txt:

```
hamza@A6Alumno08:~/kubernetes-aws-practice/app$ cat > requirements.txt << 'EOF'
Flask==3.0.0
Werkzeug==3.0.0
EOF
```

2.4 Crear Dockerfile ultraligero

Se define un Dockerfile ultraligero basado en python:3.11-slim para empaquetar la aplicación (aunque en este entorno k3s lo gestiona directamente).

Nota, este Dockerfile es solo para construcción local. k3s lo ejecutará directamente:

```
hamza@A6Alumno08:~/kubernetes-aws-practice/app$ cat > Dockerfile << 'EOF'
FROM python:3.11-slim
WORKDIR /app
# Copiar archivos
COPY requirements.txt .
COPY app.py .
COPY index.html .
# Instalar dependencias
RUN pip install --no-cache-dir -r requirements.txt
# Ejecutar app
CMD ["python", "app.py"]
EOF
```

3. Configuración de Kubernetes (MANIFESTS YAML)

Kubernetes funciona de forma declarativa: le decimos cómo queremos que sea el estado final mediante archivos YAML.

- **Namespace:** Creamos una "isla" lógica para no mezclar este proyecto con otros servicios del sistema.
- **ConfigMap:** Una técnica avanzada para inyectar archivos (como requirements.txt) dentro de los contenedores sin tener que reconstruir la imagen de Docker constantemente.
- **Deployment:** Aquí definimos las **3 réplicas**. Es la garantía de que nuestra web no se caerá.
- **Service:** Definimos el puerto 80 como entrada y el puerto 5000 (donde escucha Flask) como destino.

3.1 Creación de Namespace

Creamos una "isla" lógica llamada **load-balancer-demo** para no mezclar este proyecto con otros servicios del sistema.

Comando:

cd ~/kubernetes-aws-practice

cat > namespace.yaml << 'EOF' apiVersion: v1 kind: Namespace metadata: name: load-balancer-demo labels: name: load-balancer-demo EOF

```
hamza@A6Alumno08:~/kubernetes-aws-practice/app$ cd ~/kubernetes-aws-practice
hamza@A6Alumno08:~/kubernetes-aws-practice$ cat > namespace.yaml << 'EOF'
apiVersion: v1
kind: Namespace
metadata:
 name: load-balancer-demo
 labels:
 name: load-balancer-demo
EOF
```

Aplicar configuración:

Comando: kubectl apply -f namespace.yaml

```
hamza@A6Alumno08:~/kubernetes-aws-practice$ kubectl apply -f namespace.yaml
namespace/load-balancer-demo created
```

Verificación:

Comando: kubectl get namespaces

```
hamza@A6Alumno08:~/kubernetes-aws-practice$ kubectl get namespaces
NAME                STATUS   AGE
default             Active   78m
kube-node-lease     Active   78m
kube-public         Active   78m
kube-system         Active   78m
load-balancer-demo  Active   8s
```

3.2 Crear ConfigMap con archivos de la app

Utilizamos un **ConfigMap** (app-files) para inyectar los archivos **app.py, index.html y requirements.txt** dentro de los contenedores. Esto es una técnica avanzada que evita tener que reconstruir imágenes de Docker cada vez que cambiamos una línea de código.

Comando: cat > configmap.yaml << 'EOF' apiVersion: v1 kind: ConfigMap metadata: name: app-files namespace: load-balancer-demo data: requirements.txt: | Flask==3.0.0 Werkzeug==3.0.0

```
hamza@A6Alumno08:~$ cat > configmap.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
 name: app-files
 namespace: load-balancer-demo
data:
 requirements.txt: |
 Flask==3.0.0
 Werkzeug==3.0.0

 app.py: |
 #!/usr/bin/env python3
 from flask import Flask, jsonify, send_from_directory
 import os
 import socket
 from datetime import datetime
 import sys
 app = Flask(__name__)
 POD_NAME = os.getenv('POD_NAME', 'Unknown Pod')
 POD_NAMESPACE = os.getenv('POD_NAMESPACE', 'default')
 @app.route('/')
 def index():
 return send_from_directory('.', 'index.html')
 @app.route('/pod-info')
```

Aplicar y verificar:

```
hamza@A6Alumno08:~$ kubectl apply -f configmap.yaml
configmap/app-files created
hamza@A6Alumno08:~$ kubectl get configmap -n load-balancer-demo
NAME               DATA   AGE
app-files          3      5s
kube-root-ca.crt   1      9m14s
```

3.3 Crear Deployment con 3 replicas

El Deployment es la garantía de que nuestra web no se caerá. Definimos **3 réplicas**, asegurando alta disponibilidad.

Esperamos a que los **pods** pasen al estado *Running* y estén listos (Ready) mediante el **comando kubectl wait.**

Aplicar:

Comando: deployment kubectl apply -f deployment.yaml

```
hamza@A6Alumno08:~$ kubectl apply -f deployment.yaml
deployment.apps/web-app unchanged
```

Verificación de que los pods se están creando:

Comando: kubectl get pods -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl get pods -n load-balancer-demo
NAME                          READY   STATUS    RESTARTS   AGE
web-app-65967466dd-77ghb      1/1     Running   0          3m
web-app-65967466dd-n9ddl      1/1     Running   0          3m
web-app-65967466dd-qcsw5      1/1     Running   0          3m
```

Esperaremos a que estén ready (puede tardar 30-60 segundos)

Comando: echo "Esperando a que los pods estén listos..." kubectl wait --for=condition=ready pod -l app=web-app -n load-balancer-demo --timeout=120s

```
hamza@A6Alumno08:~$ echo "Esperando a que los pods estén listos..." kubectl wait --for
=condition=ready pod -l app=web-app -n load-balancer-demo --timeout=120
Esperando a que los pods estén listos... kubectl wait --for=condition=ready pod -l app
=web-app -n load-balancer-demo --timeout=120
```

Verificación:

Comando: kubectl get pods -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl get pods -n load-balancer-demo
NAME                          READY   STATUS    RESTARTS   AGE
web-app-65967466dd-77ghb      1/1     Running   0          4m44s
web-app-65967466dd-n9ddl      1/1     Running   0          4m44s
web-app-65967466dd-qcsw5      1/1     Running   0          4m44s
```

3.4 Crear Service (Load Balancer)

Definimos el objeto **Service** de tipo **LoadBalancer**. Mapeamos **el puerto 80** (entrada del servicio) al **puerto 5000** (donde escucha Flask).

Comando: cat > service.yaml << 'EOF' apiVersion: v1 kind: Service metadata: name: web-app-service namespace: load-balancer-demo labels: app: web-app spec: type: LoadBalancer selector: app: web-app ports: - protocol: TCP port: 80 targetPort: 5000 name: http sessionAffinity: None EOF

```
hamza@A6Alumno08:~$ cat > service.yaml << 'EOF'
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
  namespace: load-balancer-demo
  labels:
    app: web-app
spec:
  type: LoadBalancer
  selector:
    app: web-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
    name: http
  sessionAffinity: None
EOF
```

Aplicar service:

Comando: kubectl apply -f service.yaml

```
hamza@A6Alumno08:~$ kubectl apply -f service.yaml
service/web-app-service created
```

Verificar el service:

Comando: kubectl get svc -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl get svc -n load-balancer-demo
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
web-app-service   LoadBalancer   10.43.87.143   <pending>     80:32125/TCP   31s
```

Obtención de IP del service:

Información del servicio mostrando la Cluster-IP y los puertos expuestos.

Comando: kubectl get svc -n load-balancer-demo web-app-service -o wide

```
hamza@A6Alumno08:~$ kubectl get svc -n load-balancer-demo web-app-service -o wide
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE   SEL
ECTOR
web-app-service   LoadBalancer   10.43.87.143   <pending>     80:32125/TCP   56s   app
=web-app
```

4. Verificar balanceo de carga local

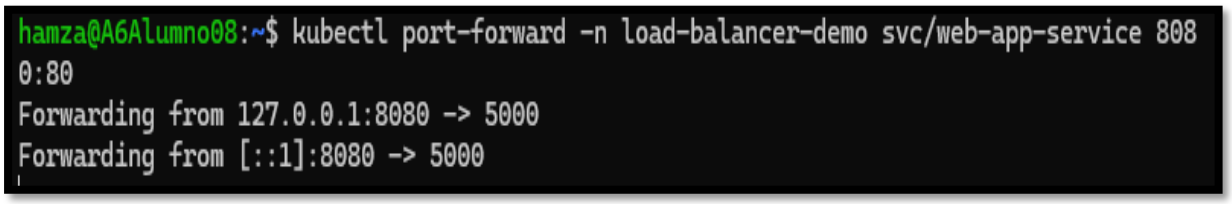
Antes de subir a la nube, probamos el "puente" interno. El comando port-forward mapea el puerto 8080 de tu Windows al **puerto 80 del servicio de Kubernetes.**

4.1 Levantar la aplicación

Port-forward (recomendado para WSL2)

El servicio no es accesible directamente desde el navegador sin un túnel. Ejecutamos **kubectl port-forward** para mapear el puerto 8080 local al 80 del servicio. El comando se queda "colgado", indicando que el túnel está activo.

Comando: kubectl port-forward -n load-balancer-demo svc/web-app-service 8080:80



4.2 Probar balanceo con curl

Ejecutamos **un script bash** que realiza 10 peticiones seguidas con curl. Se observa cómo el campo **pod_name** varía, confirmando que diferentes réplicas atienden las peticiones.

Script: for i in {1..10}; do echo "Petición $i:" curl -s http://localhost:8080/pod-info | python3 -m json.tool | grep pod_name sleep 1 done # Deberías ver el mismo pod respondiendo varias veces lo siguiente: # "pod_name": "web-app-xxxxx-11111"

```
^Chamza@A6Alumno08:~for i in {1..10}; dodo
 echo "Petición $i:"
 curl -s http://localhost:8080/pod-info | python3 -m json.tool | grep pod_name
 sleep 1
done
# Deberías ver el mismo pod respondiendo varias veces lo siguiente:
# "pod_name": "web-app-xxxxx-11111"
Petición 1:
Expecting value: line 1 column 1 (char 0)
Petición 2:
Expecting value: line 1 column 1 (char 0)
Petición 3:
Expecting value: line 1 column 1 (char 0)
Petición 4:
Expecting value: line 1 column 1 (char 0)
Petición 5:
Expecting value: line 1 column 1 (char 0)
Petición 6:
Expecting value: line 1 column 1 (char 0)
Petición 7:
```

Verificación en el navegador:

Abriremos **http://localhost:8080** en nuestro navegador y actualizaremos varias veces. Veremos que cambia el pod que atiende.

Para que funcione **localhost:8080**, te falta ejecutar un paso crítico que conecta tu máquina con el interior de **Kubernetes.** Por sí solo, el servicio no es accesible desde tu navegador sin un "túnel".

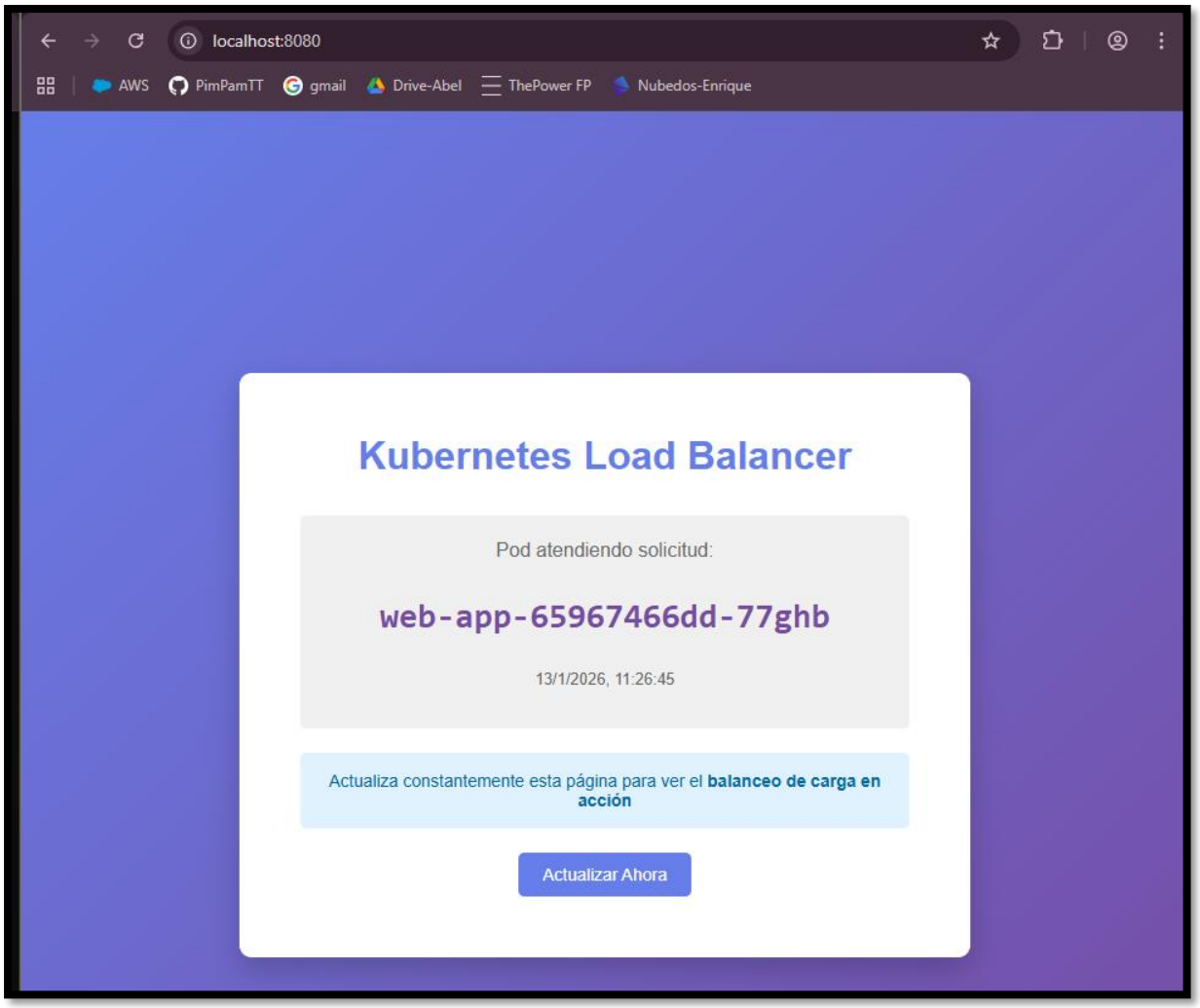
Debemos ejecutar el comando **port-forward.**

Abriremos nuestra terminal y activaremos el Port-Forward ejecutando este comando exacto:

kubectl port-forward -n load-balancer-demo svc/web-app-service 8080:80

El comando se queda "congelado". Eso significa que el túnel está abierto y funcionando.

```
hamza@A6Alumno08:~$ kubectl port-forward -n load-balancer-demo svc/web-app-service 808
0:80
Forwarding from 127.0.0.1:8080 -> 5000
Forwarding from [::1]:8080 -> 5000
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```



5. Configuración en AWS

En la consola de AWS, desplegamos una instancia **EC2 Ubuntu**. Lo más importante aquí es el **Security Group**: debemos abrir el puerto **22 (SSH)** para el túnel y opcionalmente el **8888** si queremos probar desde fuera.

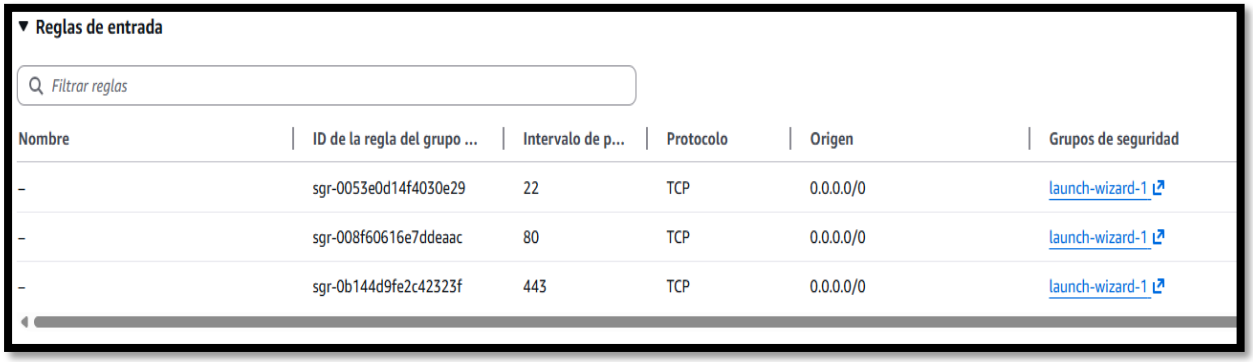
Crear Security Group:

En AWS Console:

1. Accede a EC2: Security Groups
2. Haz clic en "Create security group"
3. Nombre: kubernetes-aws-sg
4. Descripción: Security group para Kubernetes con AWS

Agregar reglas de entrada:

Tipo	Protocolo	Puerto	Origen
SSH	TCP	22	0.0.0.0/0
HTTP	TCP	80	0.0.0.0/0
HTTPS	TCP	443	0.0.0.0/0



Crear instancia EC2

En AWS Console:

1. EC2 → Instances → Launch instances
2. Nombre: kubernetes-test-server
3. AMI: Ubuntu 24.04 LTS
4. Tipo: t3.micro (Free Tier)
5. Key pair: Tu clave SSH
6. VPC: Default
7. Security group: kubernetes-aws-sg
8. Storage: 8 GiB, gp3
9. Launch instance

Conectar a EC2

Obtener IP pública desde AWS Console (ej: 54.123.45.67)

Conectar via SSH:

Comando: ssh -i ~/.ssh/tu-clave-aws.pem ubuntu@0.0.0.0

Una vez conectados, actualizaremos nuestra máquina.

En EC2, instalaremos las siguientes herramientas:

Comando: sudo apt update sudo apt install -y curl wget python3 python3-pip git

```
ubuntu@ip-172-31-18-182:~$ sudo apt install -y curl wget python3 python3-pip git
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
curl is already the newest version (8.5.0-2ubuntu10.6).
wget is already the newest version (1.21.4-1ubuntu4.1).
python3 is already the newest version (3.12.3-0ubuntu2.1).
python3 set to manually installed.
python3-pip is already the newest version (24.0+dfsg-1ubuntu1.3).
git is already the newest version (1:2.43.0-1ubuntu7.3).
0 upgraded, 0 newly installed, 0 to remove and 73 not upgraded.
```

Creación de directorio de trabajo:

Comando: mkdir -p ~/kubernetes-test

```
ubuntu@ip-172-31-18-182:~$ mkdir -p ~/kubernetes-test
```

6. Conexión de Kubernetes local con AWS EC2

Nos conectamos vía SSH a la instancia para instalar las herramientas necesarias (python3, pip, curl) y preparar el entorno de pruebas.

6.1 Creación de túnel SSH que expone el LoadBalancer

En una nueva máquina local (WSL2), mantendremos esta terminal abierta, el túnel estará activo mientras esté abierta:

Comando: ssh -i ~/.ssh/nuevaclave.pem \ -N -R 8888:localhost:8080 \ ubuntu@54.160.201.41

```
hamza@A6Alumno08:~$ kubectl port-forward -n load-balancer-demo svc/web-app-service 8080:80
Forwarding from 127.0.0.1:8080 -> 5000
Forwarding from [::1]:8080 -> 5000
```

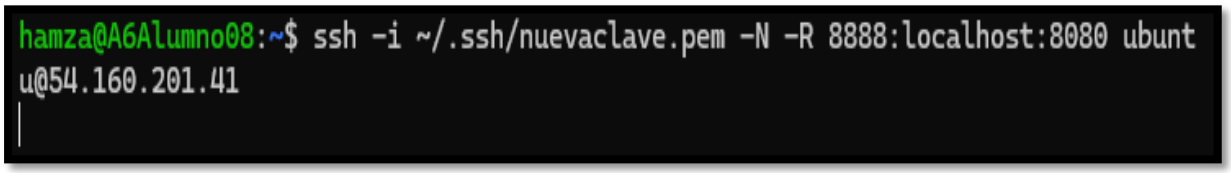
No ejecutar comandos remotos # -R 8888:localhost:8080: Redirige puerto 8888 en EC2 a puerto 8080 local

6.2 En AWS EC2 (otra terminal local)

Conectaremos a la instancia en otra terminal.

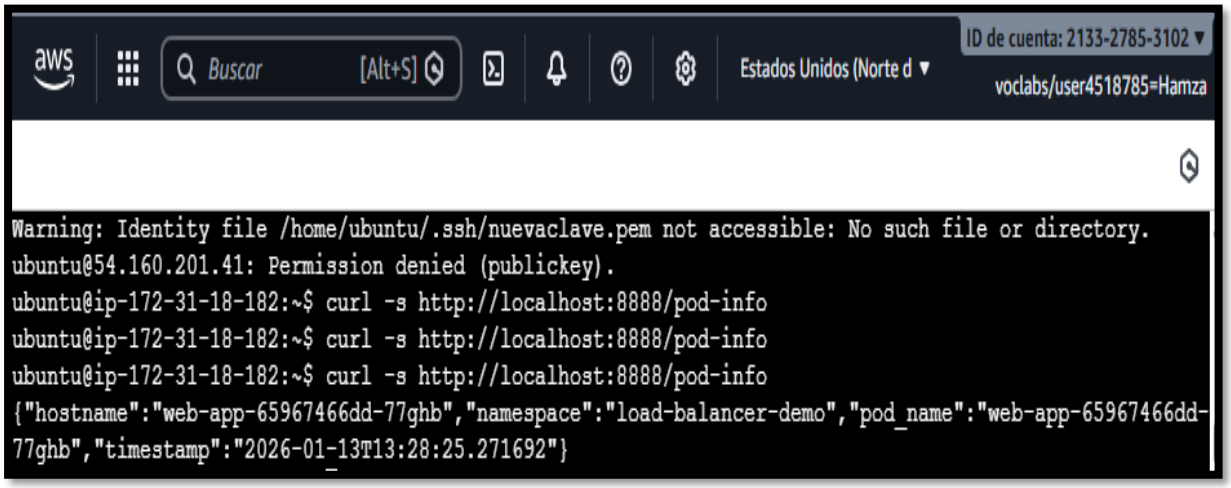
Comando: ssh -i ~/.ssh/nuevaclave.pem ubuntu@54.160.201.41

SSH **no mostrará nada** ni te dará un prompt para escribir. Se quedará el cursor parpadeando, significa que el túnel está abierto y funcionando.



Verificar que el túnel funciona:

Comando: curl -s http://localhost:8888/pod-info



Deberíamos recibir JSON:

 # {"pod_name":"web-app-xxxxx-11111", "namespace":"load-balancer-demo", ...}

6.3 Creación script de prueba en EC2

En la instancia EC2

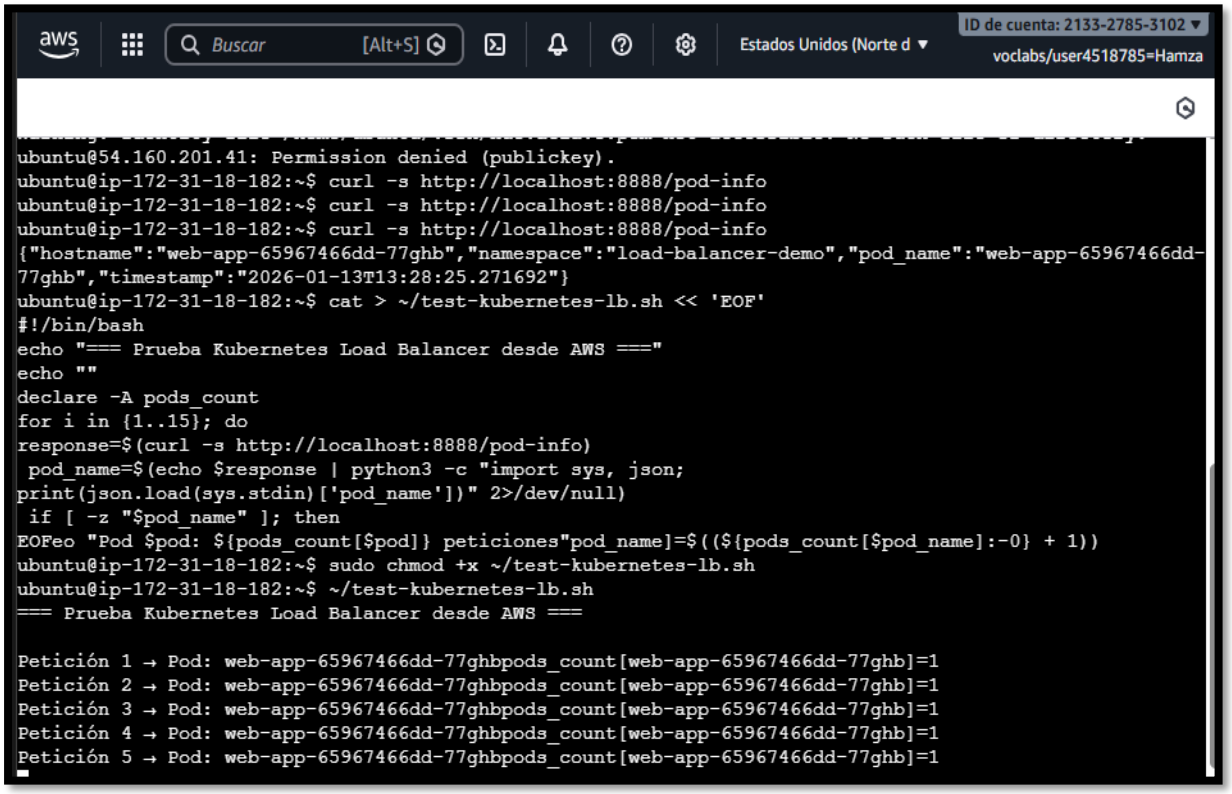
cat > ~/test-kubernetes-lb.sh << 'EOF' #!/bin/bash echo "=== Prueba Kubernetes Load Balancer desde AWS ===" echo "" declare -A pods_count for i in {1..15}; do response=$(curl -s http://localhost:8888/pod-info) pod_name=$(echo $response | python3 -c "import sys, json; print(json.load(sys.stdin)['pod_name'])" 2>/dev/null) if [-z "$pod_name"]; then pod_name="ERROR" fi echo "Petición $i → Pod: $pod_name" pods_count[$pod_name]=$((${pods_count[$pod_name]:-0} + 1)) sleep 1 done echo "" echo "=== Resumen Balanceo ===" for pod in "${!pods_count[@]}"; do echo "Pod $pod: ${pods_count[$pod]} peticiones" done EOF

Permisos:

Comando: sudo chmod +x ~/test-kubernetes-lb.sh

Ejecución de prueba:

Comando: ~/test-kubernetes-lb.sh



7. Escalado dinámico

Demostramos la potencia de Kubernetes aumentando la capacidad de procesamiento al instante. Pasamos de 3 a 5 réplicas. Kubernetes detecta la orden y crea 2 pods nuevos inmediatamente, integrándolos al balanceador de carga sin que el usuario note interrupción alguna.

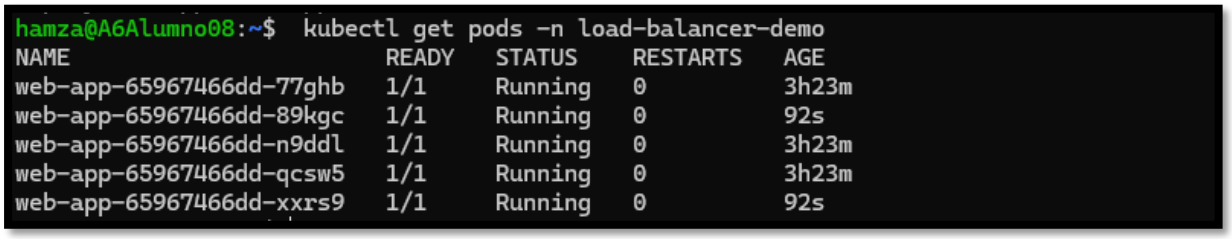
7.1 Escalar a 5 replicas

Ejecutamos **kubectl scale --replicas=5. Kubernetes** detecta la nueva orden y crea 2 pods nuevos inmediatamente, integrándolos al balanceador de carga sin interrupción del servicio.

Comando: kubectl scale deployment web-app -n load-balancer-demo --replicas=5

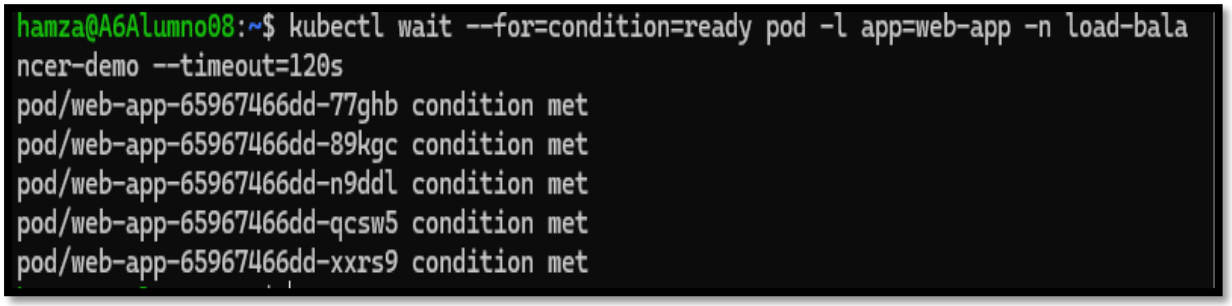
Verificación:

Comando: kubectl get pods -n load-balancer-demo



Esperararemos a que estén ready:

Comando: kubectl wait --for=condition=ready pod -l app=web-app -n load-balancer-demo --timeout=120s



7.2 Probar balanceo desde AWS

Volvemos a ejecutar el script de prueba en AWS. Ahora observamos una mayor variedad en los nombres de los pods que responden, confirmando que la carga se reparte entre las 5 instancias.

En EC2:

Comando: ~/test-kubernetes-lb.sh

```
ubuntu@ip-172-31-18-182:~$ ./test-kubernetes-lb.sh
=== Prueba Kubernetes Load Balancer desde AWS ===

Petición 1 -> Pod: web-app-7c4dd68966-wgth6
Petición 2 -> Pod: web-app-7c4dd68966-wgth6
Petición 3 -> Pod: web-app-7c4dd68966-wgth6
Petición 4 -> Pod: web-app-7c4dd68966-wgth6
Petición 5 -> Pod: web-app-7c4dd68966-wgth6
Petición 6 -> Pod: web-app-7c4dd68966-wgth6
Petición 7 -> Pod: web-app-7c4dd68966-wgth6
Petición 8 -> Pod: web-app-7c4dd68966-wgth6
Petición 9 -> Pod: web-app-7c4dd68966-wgth6
Petición 10 -> Pod: web-app-7c4dd68966-wgth6
Petición 11 -> Pod: web-app-7c4dd68966-wgth6
```

Hay más variedad de pods, pero solo va a aparecer uno.

8. Monitoreo y observabilidad

Un sistema en producción debe ser vigilado. Usamos tres niveles de monitoreo:

1. **Logs:** Ver qué está pasando "dentro" (peticiones HTTP).
2. **Top:** Ver si los pods están consumiendo demasiada memoria o CPU.
3. **Describe:** Ver el historial de eventos por si algún pod ha fallado y por qué.

8.1 Estados de pods

Para ver todos los pods usaremos el siguiente comando:

Comando: kubectl get pods -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl get pods -n load-balancer-demo
NAME                      READY   STATUS    RESTARTS      AGE
web-app-7c4dd68966-ml8th  1/1     Running   1 (48m ago)   19h
web-app-7c4dd68966-pgwmb  1/1     Running   1 (48m ago)   19h
web-app-7c4dd68966-skj6h  1/1     Running   1 (48m ago)   19h
web-app-7c4dd68966-v2c9b  1/1     Running   1 (48m ago)   19h
web-app-7c4dd68966-wgth6  1/1     Running   1 (48m ago)   19h
```

Ver logs de un pod específico:

La forma recomendada, será buscar el nombre exacto.

Comando: kubectl logs -n load-balancer-demo web-app-xxxxx-11111

```
hamza@A6Alumno08:~$ kubectl logs -n load-balancer-demo -l app=web-app
10.42.0.1 - - [14/Jan/2026 09:17:47] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:53] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:56] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:58] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:03] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:06] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:08] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:13] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:16] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:18] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:47] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:53] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:58] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:17:58] "GET /health HTTP/1.1" 200 -
```

Ver logs en tiempo real:

Comando: kubectl logs -n load-balancer-demo -l app=web-app –f

```
hamza@A6Alumno08:~$ kubectl logs -n load-balancer-demo -l app=web-app -f
10.42.0.1 - - [14/Jan/2026 09:18:51] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:56] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:01] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:02] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:07] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:11] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:12] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:17] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:21] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:19:22] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:18:49] "GET /health HTTP/1.1" 200 -
```

8.2 Ver estadísticas

CPU y memoria de pods:

Comando: kubectl top pods -n load-balancer-demo

```
^Chamza@A6Alumno08:~ kubectl top pods -n load-balancer-demomo
NAME                       CPU(cores)   MEMORY(bytes)
web-app-7c4dd68966-ml8th   1m           26Mi
web-app-7c4dd68966-pgwmb   1m           25Mi
web-app-7c4dd68966-skj6h   1m           25Mi
web-app-7c4dd68966-v2c9b   1m           26Mi
web-app-7c4dd68966-wgth6   1m           32Mi
```

Nodos del cluster:

Comando: kubectl top nodes

```
hamza@A6Alumno08:~$ kubectl top nodes
NAME        CPU(cores)   CPU(%)   MEMORY(bytes)   MEMORY(%)
a6alumno08   73m          0%       961Mi           12%
```

Eventos del cluster:

Comando: kubectl get events -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl logs -n load-balancer-demo -l app=web-app
10.42.0.1 - - [14/Jan/2026 09:37:07] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:12] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:16] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:17] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:24] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:27] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:29] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:34] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:37] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:39] "GET /health HTTP/1.1" 200 -
10.42.0.1 - - [14/Jan/2026 09:37:08] "GET /health HTTP/1.1" 200 -
```

8.3 Ver detalles del deployment

Información completa del deployment:

Comando: kubectl describe deployment web-app -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl describe deployment web-app -n load-balancer-demo
Name:                   web-app
Namespace:              load-balancer-demo
CreationTimestamp:      Tue, 13 Jan 2026 11:23:57 +0100
Labels:                 app=web-app
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=web-app
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:       app=web-app
  Annotations:  kubectl.kubernetes.io/restartedAt: 2026-01-13T14:53:21+01:00
  Containers:
   web-app:
    Image:      python:3.11-slim
    Port:       5000/TCP
    Host Port:  0/TCP
```

Historial de cambios:

Comando: kubectl rollout history deployment web-app -n load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl rollout history deployment web-app -n load-balancer-demo
deployment.apps/web-app
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

9. Eliminar recursos

Para finalizar, realizamos la limpieza del entorno para evitar costes y dejar el sistema limpio.

9.1 Limpiar Kubernetes

Eliminar todo en el namespace:

Comando: kubectl delete namespace load-balancer-demo

```
hamza@A6Alumno08:~$ kubectl delete namespace load-balancer-demo
namespace "load-balancer-demo" deleted
```

Verificar:

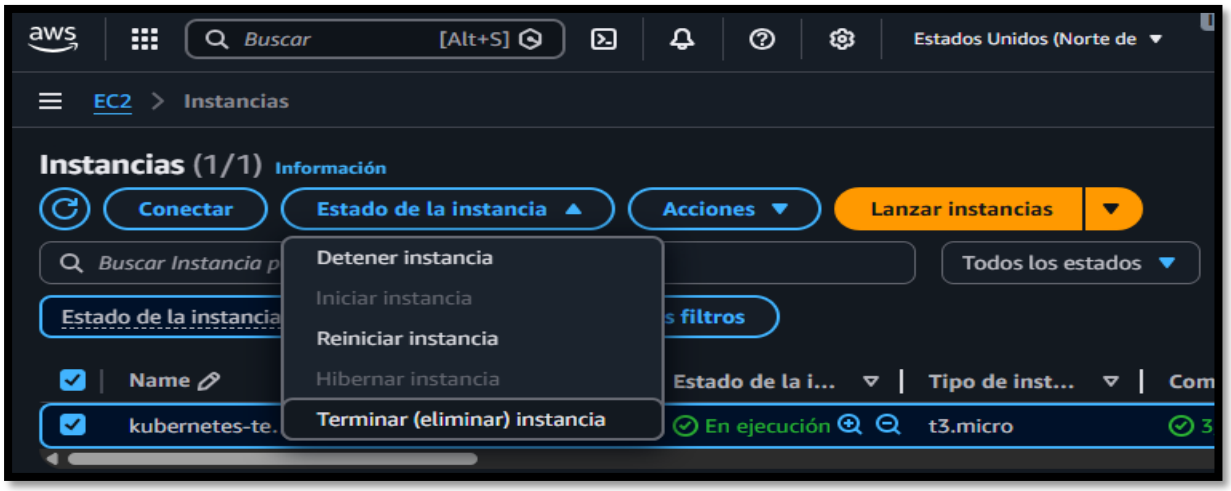
Comando: kubectl get namespaces

```
^Chamza@A6Alumno08:~kubectl get namespaceses
NAME                STATUS        AGE
default             Active        24h
kube-node-lease     Active        24h
kube-public         Active        24h
kube-system         Active        24h
load-balancer-demo  Terminating   23h
```

9.2 Eliminar instancia EC2 En AWS

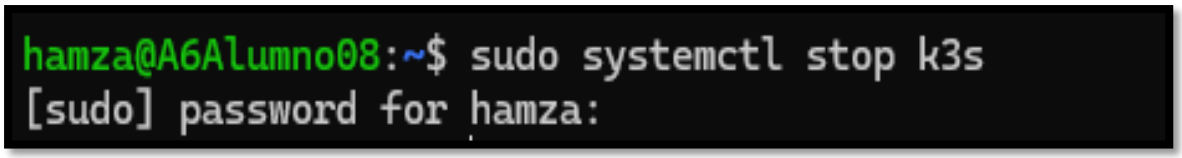
Console:

1. EC2 → Instances
2. Selecciona kubernetes-test-server
3. Instance State → Terminate
4. Confirma



9.3 Detener k3s

Comando: sudo systemctl stop k3s



Para eliminar completamente:

Comando: sudo /usr/local/bin/k3s-uninstall.sh

```
hamza@A6Alumno08:~$ sudo /usr/local/bin/k3s-uninstall.sh
+ id -u
+ [ 0 -eq 0 ]
+ K3S_DATA_DIR=/var/lib/rancher/k3s
+ /usr/local/bin/k3s-killall.sh
+ [ -s /etc/systemd/system/k3s.service ]
+ basename /etc/systemd/system/k3s.service
+ systemctl stop k3s.service
+ [ -x /etc/init.d/k3s* ]
+ killtree 1451 1660 1775 2603 2677
+ kill -9 1451 1711 2414 1660 1722 2561 1775 1960 2537 2603 2683 2882 2962 2677 2759 2911
+ do_unmount_and_remove /run/k3s
+ set +x
sh -c 'umount -f "$0" && rm -rf "$0"' /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io/fd44a31db7c9ca460afbfbe4b
1ba7ad31290c3f2595ba5b0008187340ec329b3/rootfs
sh -c 'umount -f "$0" && rm -rf "$0"' /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io/f383961b8c0d33ca6887c1ea6
641014794612dc2063cc6a45addf7c312d23049/rootfs
sh -c 'umount -f "$0" && rm -rf "$0"' /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io/f2ac03ac718ad3afbe17022c0
cdebafd150e257541e7b59c570919f767b8db86/rootfs
sh -c 'umount -f "$0" && rm -rf "$0"' /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io/cd25d6251b76aa12b77e43f56
f8c1783215006e90f3b594011eaa8b0b809457d/rootfs
sh -c 'umount -f "$0" && rm -rf "$0"' /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io/c241e832993b4ded3b7711476
```

10. Conclusiones

Este proyecto demuestra que la infraestructura moderna no depende de hardware, sino de una buena orquestación. Hemos logrado unir lo local con la nube pública, creando un entorno de alta disponibilidad escalable y monitorizado.