



UNIVERSIDAD DE GRANADA

SWAP

Implementación de Kubernetes con Minikube y Docker
para la gestión de servidores web escalables

Trabajo final
Nº wiki: equipo 13

Nuria Manzano Mata
Ignacio Escalona Blanca
Sufian Embark Aomar

Profesor: José Manuel Soto Hidalgo



Horas dedicadas entre todo el equipo: 90 horas

ÍNDICE

1. Introducción.....	2
2. Antecedentes o preliminares.....	2
2.1. Contenedores y Docker.....	2
2.2. Kubernetes.....	3
2.3. Minikube.....	4
3. Objetivos.....	4
3.1. Objetivo general.....	4
3.2. Objetivos específicos.....	4
4. Desarrollo con Kubernetes.....	4
4.1. Desarrollo con Linux.....	5
4.2. Desarrollo con macOS.....	12
5. Desarrollo sin Kubernetes.....	19
6. Resultados obtenidos y análisis.....	22
7. Conclusiones.....	28
8. Bibliografía.....	30

1. Introducción

En la actualidad, la orquestación de contenedores se ha convertido en un componente fundamental para el desarrollo, despliegue y escalabilidad de aplicaciones modernas. Kubernetes, como plataforma líder en este ámbito, ofrece funcionalidades avanzadas para gestionar cargas de trabajo contenerizadas, proporcionando escalabilidad automática, balanceo de carga y tolerancia a fallos. El presente trabajo se centra en el análisis y evaluación del uso de Kubernetes para la orquestación de contenedores Docker, utilizando Minikube para simular un entorno local que emula un despliegue en la nube.

La motivación principal de este estudio radica en comprender cómo Kubernetes puede mejorar la gestión de aplicaciones distribuidas, facilitando su administración y asegurando la continuidad del servicio ante posibles fallos. Además, se pretende evaluar los recursos consumidos, la facilidad de uso y los tiempos de respuesta, comparando el enfoque basado en Kubernetes frente a un despliegue tradicional sin orquestación.

Este trabajo está estructurado para presentar primero un marco teórico sobre orquestación de contenedores y Kubernetes, seguido por una implementación práctica con Minikube, en donde presentamos la implementación en dos sistemas operativos, Linux y MacOs. Posteriormente, se realiza un análisis comparativo que incluye pruebas de rendimiento y resiliencia. Finalmente, se extraen conclusiones basadas en los resultados obtenidos y se sugieren posibles líneas futuras de investigación y mejora.

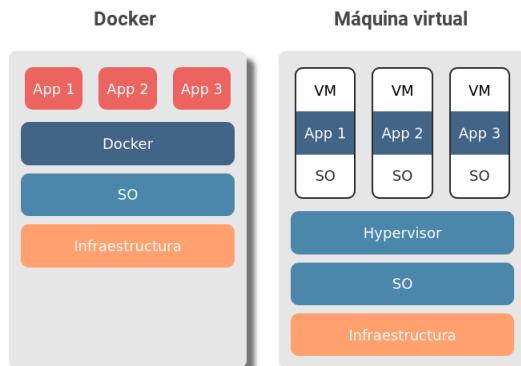
2. Antecedentes o preliminares

Para poder afrontar este trabajo sobre el uso de Kubernetes para la orquestación de contenedores Docker es necesario comprender cómo funcionan los contenedores Docker y Kubernetes.

2.1. Contenedores y Docker

Docker permite a los desarrolladores crear, ejecutar y gestionar contenedores. Estos contenedores serán los encargados de encapsular una aplicación junto a todas sus dependencias con el objetivo de que estos contenedores se puedan ejecutar de manera consistente y predecible en el mayor número de entornos posibles.

Los contenedores Docker tienen cierta similitud con las máquinas virtuales:



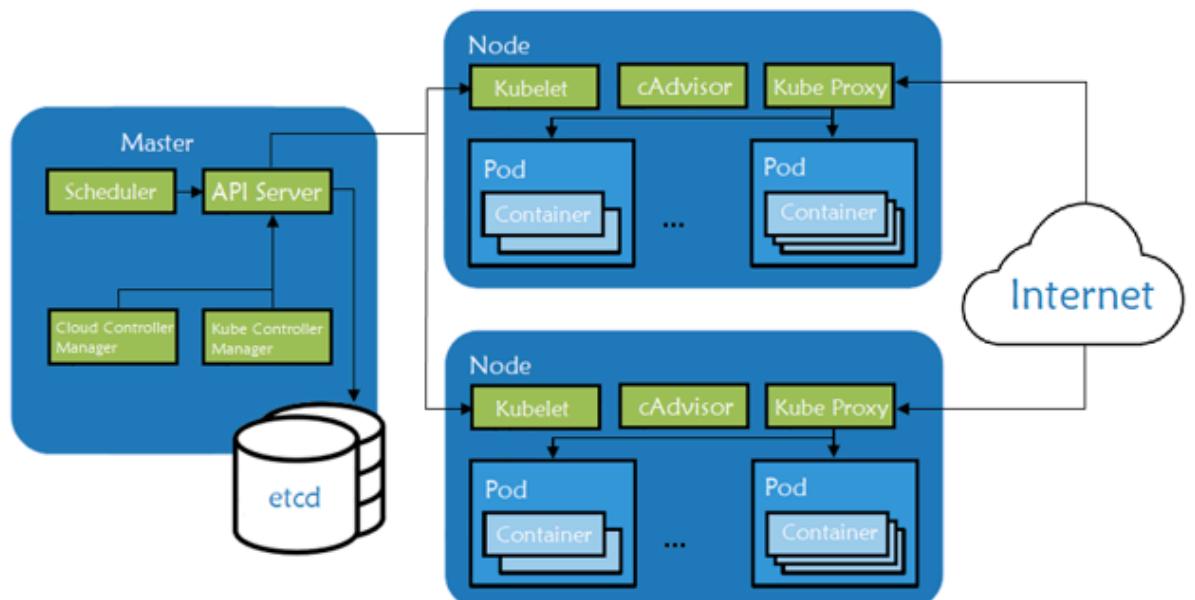
Los contenedores permiten empaquetar una aplicación junto con todas sus dependencias en una única unidad portable y aislada. A diferencia de las máquinas virtuales, los contenedores comparten el mismo núcleo del sistema operativo, lo que los hace más ligeros y rápidos. Docker es la plataforma de contenedorización más utilizada actualmente. Permite crear imágenes a partir de un `Dockerfile`, lanzar contenedores con esas imágenes y gestionar su ciclo de vida mediante comandos simples. En este trabajo, se utiliza Docker para construir y ejecutar una aplicación web en contenedor.

2.2. Kubernetes

Es una plataforma open source de orquestación de contenedores la cual permite automatizar muchos de los procesos necesarios en la gestión e implementación de aplicaciones alojadas en contenedores.

Arquitectura de Kubernetes:

- Clústeres: conjunto de máquinas físicas o virtuales que ejecutan aplicaciones contenerizadas
- Nodos:
 - MasterNodes: se encargan de gestionar el clúster.
 - Api Server: proporciona la API REST para interactuar con el clúster
 - Etcd: BBDD para coordinación de clústeres.
 - Scheduler: asigna pods a nodes.
 - Controller manager: controladores que regulan el estado del clúster.
 - WorkerNodes: se encargan de ejecutar las aplicaciones en contenedores.
 - Kubelet: se ejecuta en cada node y se asegura de que los contenedores y pods estén funcionando correctamente.
 - Kube-proxy: se encarga de recibir el tráfico necesario para los pods.
 - Container Runtime: entorno de ejecución de contenedores (Docker en nuestro caso).
- Pods: son la unidad más pequeña de kubernetes, representan un conjunto de uno o más contenedores.



Por otro lado, la configuración de recursos en Kubernetes se realiza mediante archivos YAML, que describen de forma declarativa cómo debe comportarse cada componente. Esto permite automatizar despliegues, versionar configuraciones y facilitar la gestión del ciclo de vida de las aplicaciones.

2.3. Minikube

Es una herramienta que nos permite crear clústeres de Kubernetes en 1 máquina local y de esta manera poder simular un entorno real en una sola máquina.

3. Objetivos

3.1. Objetivo general

El objetivo general de este trabajo es desplegar y comparar dos arquitecturas para un servicio web, una usando Kubernetes y Docker, y otra basada en Docker tradicional, sin usar Kubernetes; evaluando su comportamiento y características.

3.2. Objetivos específicos

- Desplegar un servicio web escalable utilizando Kubernetes con Minikube, configurando los recursos necesarios para su correcto funcionamiento.
- Implementar el mismo servicio web usando únicamente Docker, con una arquitectura de múltiples contenedores y balanceador de carga.
- Realizar pruebas de rendimiento sobre ambos despliegues, incluyendo mediciones de latencia, throughput y uso de recursos.
- Analizar las ventajas y limitaciones de cada enfoque en función de los resultados obtenidos y la experiencia de despliegue y gestión.
- Determinar recomendaciones para la elección entre Kubernetes y Docker en función del contexto, escalabilidad, complejidad y requisitos del proyecto.

4. Desarrollo con Kubernetes

En cuanto al desarrollo del trabajo, lo dividiremos en dos subapartados. Primero, abordaremos el despliegue en un entorno con sistema operativo Linux y, a continuación, realizaremos el mismo proceso adaptado a macOS. De esta forma, se podrá entender con mayor claridad los pasos a seguir según el sistema operativo que se utilice.

En ambos casos, partiremos de una base común de archivos, aunque el enfoque será distinto en cada apartado, lo cual se explicará en detalle. Los directorios creados para cada entorno son, `php-kube-linux` para la parte de Linux y `php-kube-macOS` para macOS. Cada sección especificará y detallará los archivos que contiene.

Intentaremos ser claros y concisos tanto en las explicaciones como en las capturas de pantalla utilizadas. No obstante, si es necesario, se recomienda revisar directamente los archivos para comprobar su contenido completo.

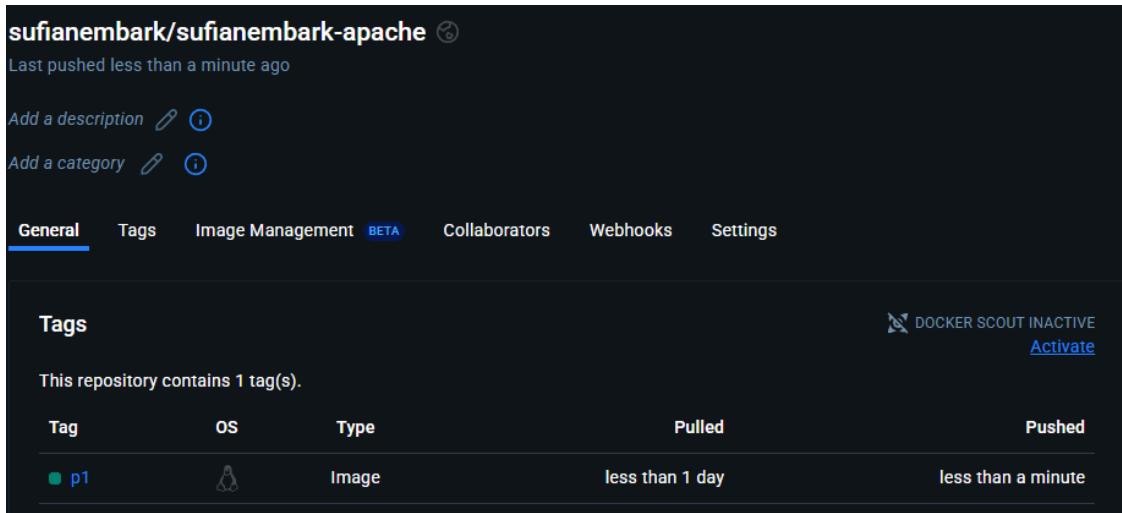
4.1. Desarrollo con Linux

Para el desarrollo de Linux, comenzaremos iniciando minikube con el comando `minikube start`. De esta manera, inicializamos un clúster local de kubernetes único por defecto, o bien reactivamos el clúster que ya tuviésemos definido. Este clúster se ejecuta dentro de una máquina virtual o contenedor, según el driver que utilice Minikube.

El nodo creado actúa **simultáneamente** como nodo **maestro (control-plane)** y nodo **trabajador (worker)**. Aunque técnicamente podríamos definir múltiples nodos en Minikube, para nuestro propósito de centrarnos en la escalabilidad con Kubernetes, utilizaremos únicamente este nodo único, escalando horizontalmente los pods que se ejecutan en él.

Para configurar los recursos del nodo, especificamos 4 CPUs y 4096 MB de memoria RAM, garantizando así suficiente capacidad para nuestras pruebas de escalado horizontal de pods. Primero eliminamos el nodo por defecto con `minikube delete` y luego iniciamos de nuevo con `minikube start --cpus=4 --memory=4096`

Una vez inicializado el clúster, necesitamos tener la imagen disponible para Minikube. La construimos localmente con `docker build -t sufianembark-apache:p1 .` . Despues, subimos la imagen a un DockerHub para que sea accesible desde Minikube. Para ello, iniciamos sesión con `docker login` etiquetamos la imagen con `docker tag sufianembark-apache:p1 sufianembark/sufianembark-apache:p1` y la subimos la imagen con el comando `docker push sufianembark/sufianembark-apache:p1`



La imagen que serviremos será un apache con un `index.php` que calcule fibonacci a un número dado:

DockerFile:

```

1 # Usa una imagen base oficial con Apache y PHP
2 FROM php:8.2-apache
3
4 # Copia el index.php al directorio web de Apache
5 COPY index.php /var/www/html/
6
7 # Da permisos apropiados
8 RUN chown -R www-data:www-data /var/www/html

```

index.php:

```

<?php
function fibonacci($n) {
    if ($n <= 1) return $n;
    return fibonacci($n - 1) + fibonacci($n - 2);
}

// Lee un parámetro para aumentar la carga
$load = isset($_GET['load']) ? (int)$_GET['load'] : 10;

$result = fibonacci($load);

echo "Fibonacci($load) = $result";
?>

```

Más adelante, usaremos un script llamado `autoloader.sh` que simula carga progresiva sobre el servidor para observar el comportamiento del autoscaler.

autoloader.sh:

```

1 #!/bin/bash
2
3 load=100 # carga inicial (ejemplo: número de iteraciones)
4
5 while true; do
6     echo "Generando carga con $load iteraciones..."
7
8     # Llamada a la web pasando el parámetro load.
9     # Más tarde, al lanzar el servicio obtendremos el puerto
10    curl "http://192.168.49.2:<PUERTO>/?load=$load"
11
12    # Incrementa la carga para la siguiente iteración
13    load=$((load + 100))
14
15    # Espera 2 segundos antes de la próxima carga
16    sleep 2
17 done

```

La ip la obtenemos con el comando `minikube ip`

Antes de lanzar el servicio, **necesitamos habilitar metrics-server** para que kubernetes pueda medir el uso de CPU/Memoria. Lo hacemos con el comando minikube addons enable metrics-server.

También **necesitamos modificar el archivo de configuracion de metrics-server** para que **ignore errores de certificados TLS** al intentar hablar con los kubelets de los nodos. Esto es debido a que Minikube genera certificados TLS **autofirmados** que hacen fallar a metrics-server por **seguridad**.

Para ello usamos el comando kubectl edit deployment metrics-server -n kube-system y añadiremos el parámetro --kubelet-insecure-tls en el apartado args

Ahora sí, creamos el archivo de lanzamiento php-apache-deployment.yaml y lo desplegamos con kubectl apply -f php-apache-deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: php-apache
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        run: php-apache
10   template:
11     metadata:
12       labels:
13         run: php-apache
14     spec:
15       containers:
16         - name: php-apache
17           image: sufianembark/sufianembark-apache:p1
18           ports:
19             - containerPort: 80
20           resources:
21             requests:
22               cpu: 200m
23             limits:
24               cpu: 500m
```

Por último, **exponemos el servicio** para poder acceder desde fuera del clúster con el comando kubectl expose deployment php-apache --type=NodePort --name=php-apache y **creamos el autoescalador** sobre ese servicio con el comando kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10. Esto crea un HPA que mantendrá el uso de CPU alrededor del 50% escalando entre 1 y 10 pods.

```
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl apply -f php-apache-deployment.yaml
deployment.apps/php-apache created
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl expose deployment php-apache --type=NodePort --name=php-apache
service/php-apache exposed
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

Obtenemos el puerto con `kubectl get svc php-apache` (En este caso 31764)

```
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get svc php-apache
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
php-apache   NodePort    10.96.154.106    <none>        80:31764/TCP   87s
```

Ahora asignamos la ip y el puerto correcto en el script de carga y lo ejecutamos con `./autoloader.sh`. Por último, observamos al escalador actuar ejecutando `kubectl get hpa` para ver el número de réplicas existentes en cada momento. Observamos que conforme pasa el tiempo, la carga de la cpu media va aumentando, y por consiguiente, el autoescalador va aumentando el número de réplicas para estabilizar la carga de la cpu.

```
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ ./autoloader.sh
Generando carga con 30 iteraciones...
Fibonacci(30) = 832040
Generando carga con 40 iteraciones...
Fibonacci(40) = 102334155
Generando carga con 50 iteraciones...
<br />
<b>Fatal error</b>: Maximum execution time of 30 seconds exceeded in <b>/var/www/html/index.php</b> on line <b>3</b><br />
Generando carga con 60 iteraciones...
<br />
<b>Fatal error</b>: Maximum execution time of 30 seconds exceeded in <b>/var/www/html/index.php</b> on line <b>3</b><br />
Generando carga con 70 iteraciones...
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 0%/50%  1           10          5           49m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 40%/50%  1           10          1           50m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 40%/50%  1           10          1           50m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 40%/50%  1           10          1           51m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 226%/50% 1           10          1           51m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 226%/50% 1           10          5           51m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 26%/50%  1           10          5           52m
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache  Deployment/php-apache  cpu: 26%/50%  1           10          5           52m
```

Al ser una carga muy elevada, el cambio es brusco, pero jugando con los parámetros, podemos simular una carga más parecida a la realidad con una escalada más progresiva. Iniciamos una primera vez el script con los nuevos parámetros, provocando una **escalada del servicio, hasta estabilizarse en 3 réplicas**:

```
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 9%/50%   1            10           2            5m43s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 49%/50%   1            10           2            6m31s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 49%/50%   1            10           2            6m44s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 60%/50%   1            10           2            7m10s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 60%/50%   1            10           3            7m34s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 60%/50%   1            10           3            7m56s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 47%/50%   1            10           3            8m25s
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 40%/50%   1            10           3            9m11s
```

Luego iniciamos de nuevo el script en otro terminal y notamos como la carga vuelve a ascender, **hasta estabilizarse en 5 réplicas**:

```
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 52%/50%   1            10           3            19m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 52%/50%   1            10           3            21m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 69%/50%   1            10           3            22m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 69%/50%   1            10           5            22m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 69%/50%   1            10           5            22m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 69%/50%   1            10           5            23m
sufian@DESKTOP-TBVNSSL:~$ kubectl get hpa
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
php-apache Deployment/php-apache  cpu: 47%/50%   1            10           5            23m
```

Finalmente cerramos los script de carga y observamos como, tras un tiempo, **kubernetes acaba dejando una sola réplica de nuevo**:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 36%/50%	1	10	5	24m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 36%/50%	1	10	5	24m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 36%/50%	1	10	5	25m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	5	25m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	5	25m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	5	26m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	5	27m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	5	28m
sufian@DESKTOP-TBVNSSL:~\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	cpu: 0%/50%	1	10	1	32m

Como hemos podido observar, **el sistema de auto escalado horizontal en Kubernetes ha demostrado ser efectivo para gestionar cargas variables de trabajo**. A través del uso del Horizontal Pod Autoscaler (HPA), observamos cómo el clúster responde de forma dinámica al aumento de uso de CPU, creando nuevas réplicas de forma progresiva hasta estabilizar la carga. Es importante destacar que el valor de CPU monitorizado representa una media de todas las réplicas activas, lo que permite una decisión de escalado más equilibrada. Kubernetes requiere al menos 30 segundos con la CPU por encima del umbral configurado (50%) para escalar, y un periodo de gracia de aproximadamente 5 minutos para reducir réplicas cuando la carga disminuye. Este comportamiento confirma la capacidad de Kubernetes para balancear carga eficientemente entre réplicas nuevas, aunque también evidencia que el ajuste de parámetros como el umbral de CPU y los períodos de evaluación puede tener un impacto significativo en la agilidad del escalado.

En conjunto, los resultados validan que Kubernetes, incluso en un entorno simulado con Minikube, es capaz de ofrecer una respuesta robusta y adaptable ante variaciones de demanda, lo que lo convierte en una herramienta adecuada para sistemas que requieren elasticidad y continuidad de servicio.

A continuación pasaremos a verificar la capacidad de Kubernetes de **mantener la alta disponibilidad ante fallos**, desplegaremos un Deployment llamado `php-apache-3` configurado con 3 réplicas.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: php-apache-3
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: php-apache-3
10   template:
11     metadata:
12       labels:
13         app: php-apache-3
14     spec:
15       containers:
16         - name: php-apache-3
17           image: sufianembark/sufianembark-apache:p1
18         ports:
19           - containerPort: 80
20

```

Tras el despliegue con `kubectl apply -f php-apache-deployment-3.yaml` verificamos el correcto despliegue con `kubectl get pods -l app=php-apache-3 -w`

```

sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl apply -f php-apache-deployment-3.yaml
deployment.apps/php-apache-3 created
sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl get pods -l app=php-apache-3 -w
NAME          READY   STATUS    RESTARTS   AGE
php-apache-3-5d99bdddc-8c28d   1/1    Running   0          3s
php-apache-3-5d99bdddc-hcwm7   1/1    Running   0          3s
php-apache-3-5d99bdddc-n8jj7   1/1    Running   0          3s

```

Para simular un fallo, forzamos la **eliminación manual de uno de los pods**, y observamos como **automáticamente se reemplaza por uno nuevo**:

```

sufian@DESKTOP-TBVNSSL:~/ProyectoSWAP$ kubectl delete pod php-apache-3-5d99bdddc-n8jj7
pod "php-apache-3-5d99bdddc-n8jj7" deleted

```

NAME	READY	STATUS	RESTARTS	AGE
php-apache-3-5d99bdddc-n8jj7	1/1	Terminating	0	34s
php-apache-3-5d99bdddc-qhqnc	0/1	Pending	0	0s
php-apache-3-5d99bdddc-qhqnc	0/1	Pending	0	0s
php-apache-3-5d99bdddc-qhqnc	0/1	ContainerCreating	0	0s
php-apache-3-5d99bdddc-qhqnc	1/1	Running	0	1s
php-apache-3-5d99bdddc-n8jj7	0/1	Completed	0	36s
php-apache-3-5d99bdddc-n8jj7	0/1	Completed	0	36s
php-apache-3-5d99bdddc-n8jj7	0/1	Completed	0	36s

Este comportamiento demuestra cómo **Kubernetes garantiza la resiliencia y la continuidad del servicio**, reemplazando automáticamente cualquier réplica caída para mantener el número deseado de instancias definidas en el Deployment.

4.2. Desarrollo con macOS

El desarrollo con macOS se ha realizado siguiendo los siguientes pasos, aunque he de añadir que los conceptos teóricos explicados anteriormente son exactamente los mismos, por ello no se volverán a repetir y únicamente se explicará como se ha logrado el despliegue y los conceptos exclusivos para este apartado.

En primer lugar, es necesario tener Docker Desktop instalado y corriendo en tu máquina para poder iniciar Minikube sin problema. Una vez que hayamos instalado Minikube brew install minikube, tenemos que asegurarnos de que también tenemos instalado kubectl (al igual que con Linux) brew install kubectl.

IMPORTANTE: Si aparece un mensaje mencionando que está instalado pero no linkeado entonces debemos escribir el comando brew link kubernetes-cli

En este punto, iniciamos minikube minikube start

IMPORTANTE: Si en esta parte se genera un error, probablemente será porque Docker Desktop no está abierto y corriendo.

Una vez realizado todo lo anterior, comenzaremos con la creación de los diferentes archivos de configuración que podemos encontrarlos en /php-kube-macOS

El archivo index.php contiene la app PHP. En este caso, a diferencia del apartado anterior, se ha usado las sesiones del navegador para ir duplicandolas conforme se recargue la página:

```
2025-06-06 12:08:41 ✎ Nurias-MacBook-Pro in ~/Deskt  
op/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/php-kub  
e  
○ → cat index.php  
<?php  
session_start();  
if (!isset($_SESSION['counter'])) {  
    $_SESSION['counter'] = 1;  
} else {  
    $_SESSION['counter'] *= 2;  
}  
$loop = $_SESSION['counter'];  
$sum = 0;  
for ($i = 0; $i < $loop; $i++) {  
    $sum += sqrt($i);  
}  
echo "Iteraciones: $loop - Resultado: $sum";  
?>
```

En el caso del **Dockerfile** se construye la imagen con Apache y PHP, similar al apartado anterior

```
2025-06-06 12:08:49 ✎ Nurias-MacBook-Pro in ~/Deskt  
op/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/php-kub  
e  
○ → cat Dockerfile  
FROM php:8.2-apache  
COPY index.php /var/www/html/  
RUN docker-php-ext-install session
```

En el caso del archivo `deployment.yaml`, se ha completado de forma diferente al apartado anterior, en este caso, se ha preferido incluir directamente la configuración del Deployment, Service y Horizontal Pod Autoscaler (HPA), incluyendo réplicas mínimas y máximas, y configuración de escalado.

```
2025-06-06 12:09:16 ✘ Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/php-kube
o ➔ cat deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  replicas: 3
  selector:
    matchLabels:
      app: php-apache
  template:
    metadata:
      labels:
        app: php-apache
    spec:
      containers:
        - name: php-apache
          image: equipo13-apache-image:p6
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: "100m"
              memory: "64Mi"
            limits:
              cpu: "500m"
              memory: "128Mi"
  ---
apiVersion: v1
kind: Service
metadata:
  name: php-apache-service
spec:
  type: NodePort
  selector:
    app: php-apache
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
  ---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 3
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

Una vez que tenemos toda la configuración completada, llega el momento de construir la imagen Docker dentro de Minikube con los siguientes comandos:

1. Activamos el entorno Docker de Minikube: `eval $(minikube docker-env)`
2. Construimos la imagen Docker con tag personalizado: `docker build -t equipo13-apache-image:p6` .

```

2025-06-06 11:36:33 ✎ Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/
hp-kube
○ → eval $(minikube docker-env)

2025-06-06 11:37:40 ✎ Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/
[hp-kube
○ → docker build -t equipo13-apache-image:p6 .
[+] Building 43.7s (9/9) FINISHED                                            docker:default
   => [internal] load build definition from Dockerfile                      0.0s
[ => => transferring dockerfile: 159B                                         0.0s
   => [internal] load metadata for docker.io/library/php:8.2-apache          2.5s
   => [auth] library/php:pull token for registry-1.docker.io                  0.0s
   => [internal] load .dockerignore                                           0.1s
   => => transferring context: 2B                                           0.0s
   => [internal] load build context                                         0.1s
   => => transferring context: 351B                                         0.0s
   => [1/3] FROM docker.io/library/php:8.2-apache@sha256:f8816038aecbd2bcd 28.3s
   => => resolve docker.io/library/php:8.2-apache@sha256:f8816038aecbd2bcd2 0.0s
   => => sha256:b4ce612dc732adccbd880a354ee9af6a19b52c126027356 225B / 225B 0.5s
   => => sha256:bebfa42fd224138641540bb485f1eb1920b7ec6b93 3.64kB / 3.64kB 0.0s
   => => sha256:093338982d92caf6c799c270562af3382ca1ce2 104.33MB / 104.33MB 5.5s
   => => sha256:f8816038aecbd2bcd2f29c662687ccb1151f9ee5 10.40kB / 10.40kB 0.0s
   => => sha256:d0886ccbf6e9bd60fa7de105dde7b7d69257fcc3d 11.38kB / 11.38kB 0.0s
   => => sha256:61320b01ae0e798393ef25f2dc72faf43703e60b 28.23MB / 28.23MB 1.1s
   => => sha256:0c2a0ba9eb0c18d2453647f08a87a0a53eb1291ee91aa54 225B / 225B 0.7s
   => => sha256:442abaed77518b7a9c1df49d43ea6fc520469f114 20.12MB / 20.12MB 1.8s
   => => sha256:aa4e51934eeef5c693fa90c8fbfb5dadf695eb1f4eda985 429B / 429B 1.3s
   => => extracting sha256:61320b01ae0e798393ef25f2dc72faf43703e60ba089b0 12.0s
   => => sha256:924949db942b0079f2da66e25e58d59874c70ab6eaab013 480B / 480B 1.7s
   => => sha256:153d2fb08c64974e8da0f4666c1dd95d2286a4aa8 12.28MB / 12.28MB 2.4s
   => => sha256:26f17ce45149dff48a2163f1cf82b7fc16963ceb66150e5 487B / 487B 2.2s
   => => sha256:64a857ca8a6acaf9c76bf75ddaa41134ec2d641c1 11.42MB / 11.42MB 2.9s
   => => sha256:7443480a6bdbe3b3132144656453a2b94ddebd3a9 2.46kB / 2.46kB 2.7s
   => => sha256:d4dfcfe68eba6b10ca65bfcfe9f7dc5ddf263cc5d3e840e 242B / 242B 2.9s
   => => sha256:91e76e37da73c3b7fa0c82623e137511bafb35be8238e42 890B / 890B 3.1s
   => => sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d 32B / 32B 3.2s
   => => extracting sha256:b4ce612dc732adccbd880a354ee9af6a19b52c1260273569 0.0s
   => => extracting sha256:093338982d92caf6c799c270562af3382ca1ce25205c8d1 10.1s
   => => extracting sha256:0c2a0ba9eb0c18d2453647f08a87a0a53eb1291ee91aa542 0.0s
   => => extracting sha256:442abaed77518b7a9c1df49d43ea6fc520469f1141ab82f9 1.5s
   => => extracting sha256:aa4e51934eeef5c693fa90c8fbfb5dadf695eb1f4eda985b 0.0s
   => => extracting sha256:924949db942b0079f2da66e25e58d59874c70ab6eaab0131 0.0s
   => => extracting sha256:153d2fb08c64974e8da0f4666c1dd95d2286a4aa8d033d1b 0.3s
   => => extracting sha256:26f17ce45149dff48a2163f1cf82b7fc16963ceb66150e54 0.0s
   => => extracting sha256:64a857ca8a6acaf9c76bf75ddaa41134ec2d641c1b74d2ad 1.3s
   => => extracting sha256:7443480a6bdbe3b3132144656453a2a29b4ddebd3a9cfa72 0.0s
   => => extracting sha256:d4dfcfe68eba6b10ca65bfcfe9f7dc5ddf263cc5d3e840e8 0.0s
   => => extracting sha256:91e76e37da73c3b7fa0c82623e137511bafb35be8238e42b 0.0s
   => => extracting sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6 0.0s
=> [2/3] COPY index.php /var/www/html/                                         0.0s
=> [3/3] RUN docker-php-ext-install session                                     12.6s
=> exporting to image                                                       0.0s
=> => exporting layers                                                       0.0s
=> => writing image sha256:9627affdb40463a9764cia15d520fa93a57d3ba86d1b7 0.0s
=> => naming to docker.io/library/equipo13-apache-image:p6                 0.0s

```

□	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
□	● minikube	6deadb21de0a	k8s-minikube/kicbas	52420:22 ↗ Show all ports (5)	22.04%	36 minutes ago	⋮ trash

3. Aplicamos el archivo de despliegue en Kubernetes: `kubectl apply -f deployment.yaml`

```

2025-06-06 11:39:12 ✎ Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUND
O CUATRI/SWAP/TrabajoFinal/php-kube
○ → kubectl apply -f deployment.yaml
deployment.apps/php-apache created
service/php-apache-service created
horizontalpodautoscaler.autoscaling/php-apache-hpa unchanged

```

4. Verificamos que todo se ha creado correctamente:

```

2025-06-06 11:40:06 📈 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUND
o CUATRI/SWAP/TrabajoFinal/php-kube
o ➔ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
php-apache-75cb8bb769-7xsr7  1/1     Running   0          8s
php-apache-75cb8bb769-8x2jz  1/1     Running   0          8s
php-apache-75cb8bb769-rkch5  1/1     Running   0          8s

2025-06-06 11:40:14 📈 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUND
o CUATRI/SWAP/TrabajoFinal/php-kube
o ➔ kubectl get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   3/3      3           3           15s

2025-06-06 11:40:21 📈 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUND
o CUATRI/SWAP/TrabajoFinal/php-kube
o ➔ kubectl get services
NAME              TYPE       CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
kubernetes        ClusterIP   10.96.0.1      <none>          443/TCP
2m37s
php-apache-service   NodePort   10.103.122.108  <none>          80:30080/TCP
CP   24s

```

5. Exponemos el servicio para acceder desde el navegador con el comando `kubectl expose deployment php-apache --type=NodePort --port=80` De esta manera, se abre automáticamente la app PHP en el navegador:

```

2025-06-06 11:40:36 📈 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUND
o CUATRI/SWAP/TrabajoFinal/php-kube
o ➔ minikube service php-apache-service
|-----|-----|-----|
|   |-----|-----|-----|
|   | NAMESPACE | NAME | TARGET PORT | URL
|   |-----|-----|-----|
|   |-----|-----|-----|
|   | default | php-apache-service | 80 | http://192.168.49.2:30080
|   |
|   |-----|-----|-----|
|   |-----|-----|-----|
|   Starting tunnel for service php-apache-service.
|-----|-----|-----|-----|
|   | NAMESPACE | NAME | TARGET PORT | URL
|   |-----|-----|-----|
|   | default | php-apache-service | http://127.0.0.1:52495 |
|   |
|   |-----|-----|-----|
|   |-----|-----|-----|
|   🚀 Opening service default/php-apache-service in default browser...
|   ! Because you are using a Docker driver on darwin, the terminal needs to
|   be open to run it.

```

Iteraciones: 1 - Resultado: 0

y tras recargar la página para simular peticiones obtenemos:

Iteraciones: 1024 - Resultado: 21829.126749192

```

    ⏪ ⏩ ⏴ ⏵ ① 127.0.0.1:52495
    Iteraciones: 65536 - Resultado: 11184682.458943
    ⏪ ⏩ ⏴ ⏵ ① 127.0.0.1:52495
    Iteraciones: 8388608 - Resultado: 16197334551.51

```

IMPORTANTE: Para automatizar todo este proceso he creado un script en el que se despliega todo el proceso anterior, se puede encontrar en `/php-kube-macOS/deploy-php-minikube.sh`

Ahora pasamos a verificar que el horizontal Pod autoscaler está funcionando correctamente y para ello debemos asegurarnos que **el estado de metrics-server en el namespace kube-system está activo** con `minikube addons enable metrics-server`, si con eso no es suficiente se debe añadir en el deployment un flags para que ignoren TLS: `kubectl -n kube-system edit deployment metrics-server` o reiniciar minikube de la siguiente manera `minikube start --addons=metrics-server --extra-config=kubelet.authentication-token-webhook=true` para ignorar TLS (lo mismo que ocurría en el caso de Linux).

En mi caso, tras `minikube addons enable metrics-server` si que estaba todo correctamente:

```

2025-06-06 11:46:44 🕒 Nurias-MacBook-Pro in ~
[○ → kubectl get pods -n kube-system
NAME                      READY   STATUS    RESTARTS   AGE
coredns-674b8bbfcf-8hm5r   1/1     Running   1 (62s ago)  9m2s
etcd-minikube              1/1     Running   1 (60s ago)  9m7s
kube-apiserver-minikube   1/1     Running   1 (57s ago)  9m7s
kube-controller-manager-minikube  1/1     Running   1 (67s ago)  9m7s
kube-proxy-86mzk            1/1     Running   1 (67s ago)  9m2s
kube-scheduler-minikube   1/1     Running   1 (67s ago)  9m7s
metrics-server-7fbb699795-k6m2k  1/1     Running   0          37s
storage-provisioner         1/1     Running   3 (50s ago)  9m4s

2025-06-06 11:47:02 🕒 Nurias-MacBook-Pro in ~
[○ → kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
php-apache-hpa Deployment/php-apache  cpu: 5%/50%  3          5          3          8m56s

```

Por lo tanto, podemos **dmostrar el escalado automático del proyecto creado**. Para ello voy a acceder a un pod para generar carga dentro de él `kubectl exec -it <nombre-del-pod> -- /bin/sh` ahora, dentro del pod, añadiré una línea de comandos haciendo peticiones al servicio para subir el uso de CPU `while true; do curl -s http://php-apache-service; done`

```

2025-06-07 21:16:53 🕒 Nurias-MacBook-Pro in ~
[○ → kubectl exec -it php-apache-75cb8bb769-b6rl7 -- /bin/sh

```

Finalmente, desde otra terminal (al mismo tiempo) voy monitorizando el HPA para poder observar cómo se comporta Kubernetes y cómo va aumentando el número de réplicas (pods) a medida que aumenta el consumo de CPU:

```
mmnuria — kubectl get pods -w — 80x24
php-apache-75cb8bb769-b6rl7 1/1 Running 1 (9m8s ago) 13m
php-apache-75cb8bb769-f52gd 1/1 Running 1 (9m8s ago) 13m
php-apache-75cb8bb769-rg4vq 1/1 Running 1 (9m8s ago) 13m
kube-apiserver-minikube 0/1 RunContainerError 6 (14s ago) 6m6s
kube-apiserver-minikube 0/1 CrashLoopBackOff 6 (24s ago) 6m16s
php-apache-75cb8bb769-2jjx7 0/1 Pending 0 0s
php-apache-75cb8bb769-2jjx7 0/1 Pending 0 0s
php-apache-75cb8bb769-sc9qq 0/1 Pending 0 0s
php-apache-75cb8bb769-sc9qq 0/1 Pending 0 0s
php-apache-75cb8bb769-2jjx7 0/1 ContainerCreating 0 0s
php-apache-75cb8bb769-sc9qq 0/1 ContainerCreating 0 0s
php-apache-75cb8bb769-2jjx7 1/1 Running 0 3s
php-apache-75cb8bb769-sc9qq 1/1 Running 0 3s
kube-apiserver-minikube 0/1 RunContainerError 7 (13s ago) 11m
kube-apiserver-minikube 0/1 CrashLoopBackOff 7 (16s ago) 11m
php-apache-75cb8bb769-sc9qq 1/1 Terminating 0 5m46s
php-apache-75cb8bb769-2jjx7 1/1 Terminating 0 5m46s
php-apache-75cb8bb769-2jjx7 0/1 Completed 0 5m47s
php-apache-75cb8bb769-sc9qq 0/1 Completed 0 5m47s
php-apache-75cb8bb769-2jjx7 0/1 Completed 0 5m47s
php-apache-75cb8bb769-2jjx7 0/1 Completed 0 5m47s
php-apache-75cb8bb769-sc9qq 0/1 Completed 0 5m47s
php-apache-75cb8bb769-sc9qq 0/1 Completed 0 5m47s

mmnuria — kubectl get hpa -w — 80x24
2025-06-07 21:17:20 ✘ Nurias-MacBook-Pro in ~
[ o ➔ kubectl get hpa -w
NAME REFERENCE TARGETS MINPODS MAXPODS REPLI
CAS AGE
php-apache-hpa Deployment/php-apache cpu: 1%/50% 3 5 3
15m
php-apache-hpa Deployment/php-apache cpu: 18%/50% 3 5 3
15m
php-apache-hpa Deployment/php-apache cpu: 86%/50% 3 5 3
16m
php-apache-hpa Deployment/php-apache cpu: 86%/50% 3 5 5
17m
php-apache-hpa Deployment/php-apache cpu: 32%/50% 3 5 5
17m
php-apache-hpa Deployment/php-apache cpu: 1%/50% 3 5 5
18m
php-apache-hpa Deployment/php-apache cpu: 1%/50% 3 5 5
21m
php-apache-hpa Deployment/php-apache cpu: 1%/50% 3 5 5
22m
php-apache-hpa Deployment/php-apache cpu: 1%/50% 3 5 3
```

Aquí se puede ver, cómo a medida que va aumentando el uso de CPU se crean pods nuevos para soportar la demanda creciente y el efecto contrario cuando he detenido las peticiones, como al mismo tiempo que se reduce el uso de CPU se van eliminando dichos pods que ahora son innecesarios.

Lo siguiente a probar es la **simulación del fallo de un pod**, de forma que podemos **probar la resiliencia y comportamiento de Kubernetes ante errores**. Para ello vamos a eliminar un pod manualmente y observar qué hace automáticamente Kubernetes. Para poder realizarlo, necesitamos saber los nombres exactos de los pods:

```

2025-06-06 11:50:36 ✉ Nurias-MacBook-Pro in ~
[○ → kubectl get pods -w
NAME READY STATUS RESTARTS AGE
load-generator 1/1 Running 0 2m43s
php-apache-75cb8bb769-8x2jz 1/1 Running 1 (4m44s ago) 10m
php-apache-75cb8bb769-qlm4x 1/1 Running 0 6s
php-apache-75cb8bb769-rkch5 1/1 Running 1 (4m44s ago) 10m
]

```

Sabiendo los nombres de los pods, eliminamos uno manualmente:

```

2025-06-06 11:51:03 ✉ Nurias-MacBook-Pro in ~
[○ → kubectl delete pod php-apache-75cb8bb769-8x2jz
pod "php-apache-75cb8bb769-8x2jz" deleted

```

Y a la vez, en otra terminal vamos a ir viendo como Kubernetes automáticamente crea un nuevo pod para mantener el estado requerido de nuestro servicio.

```

2025-06-06 11:49:31 ✉ Nurias-MacBook-Pro in ~
[○ → kubectl get pods -w
NAME READY STATUS RESTARTS AGE
load-generator 1/1 Running 0 3m
php-apache-75cb8bb769-8x2jz 1/1 Running 1 (5m1s ago) 10m
php-apache-75cb8bb769-qlm4x 1/1 Running 0 23s
php-apache-75cb8bb769-rkch5 1/1 Running 1 (5m1s ago) 10m
php-apache-75cb8bb769-8x2jz 1/1 Terminating 1 (5m18s ago) 11m
php-apache-75cb8bb769-md4b8 0/1 Pending 0 0s
php-apache-75cb8bb769-md4b8 0/1 Pending 0 0s
php-apache-75cb8bb769-md4b8 0/1 ContainerCreating 0 0s
php-apache-75cb8bb769-8x2jz 0/1 Completed 1 (5m19s ago) 11m
php-apache-75cb8bb769-md4b8 1/1 Running 0 1s
php-apache-75cb8bb769-8x2jz 0/1 Completed 1 11m
php-apache-75cb8bb769-8x2jz 0/1 Completed 1 11m
php-apache-75cb8bb769-8x2jz 0/1 Completed 1 11m

```

```

2025-06-06 12:34:08 ✉ Nurias-MacBook-Pro in ~
[○ → kubectl get pods
NAME READY STATUS RESTARTS AGE
load-generator 1/1 Running 0 46m
php-apache-75cb8bb769-md4b8 1/1 Running 0 42m
php-apache-75cb8bb769-qlm4x 1/1 Running 0 43m
php-apache-75cb8bb769-rkch5 1/1 Running 1 (48m ago) 54m

```

Finalmente, hemos conseguido desplegar con éxito un servicio en Kubernetes con Minikube de forma que garantice escalabilidad y resiliencia, tanto en un sistema operativo como Linux y macOS.

5. Desarrollo sin Kubernetes

El objetivo de este apartado es desplegar el mismo servicio, pero sin utilizar Kubernetes, con el fin de poder realizar una **comparativa práctica y detallada entre los resultados obtenidos con y sin su uso**.

Los archivos de configuración correspondientes a este despliegue se encuentran en la carpeta /php-sinKube. En este caso, el entorno se ha montado utilizando Docker y Nginx, intentando simular un escenario lo más parecido posible al de Kubernetes.

Tanto index.php como el Dockerfile son idénticos a los utilizados en /php-kube-macOS, por lo que no los volveré a mostrar aquí.

Por otro lado, el archivo docker-compose.yml sí ha sido creado específicamente para este apartado. Es un fichero simple donde se definen tres contenedores PHP (php1, php2 y php3), todos basados en la misma imagen, y un balanceador de carga Nginx (nginx-balancer) que se conecta a ellos y expone el servicio por el puerto 8080, tal como se ha trabajado en las prácticas de la asignatura.

```
2025-06-06 12:57:48 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal/php-sinKube
o ➔ cat docker-compose.yml

services:
  php1:
    build: .
    container_name: php1

  php2:
    build: .
    container_name: php2

  php3:
    build: .
    container_name: php3

  nginx:
    image: nginx:alpine
    container_name: nginx-balancer
    ports:
      - "8080:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - php1
      - php2
      - php3
```

El servicio se desplegamos con docker compose up -d

```
[+] Running 5/5p3
✓ Network php-sinkube_default      Star          Created0.1s later php1
✓ Container php2                   Star          Started0.6s later nginx-balanc
er
✓ Container php3                  Started0.5s
✓ Container php1                  Started0.5s
✓ Container nginx-balancer        Started0.7s
```

<input type="checkbox"/>	<input checked="" type="checkbox"/>	●	php-sinkube			0.02%	6 minutes ago	<input type="checkbox"/>	⋮	<input type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>	●	php3	3b8925fab261	php-sinkube-php3	0.01%	6 minutes ago	<input type="checkbox"/>	⋮	<input type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>	●	php2	391e85f4e030	php-sinkube-php2	0.01%	6 minutes ago	<input type="checkbox"/>	⋮	<input type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>	●	php1	99b6869d11cb	php-sinkube-php1	0%	6 minutes ago	<input type="checkbox"/>	⋮	<input type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>	●	nginx-balancer	454c08741e3c	nginx:alpine	8080:80	8080:80	<input type="checkbox"/>	⋮	<input type="checkbox"/>

Y al visitar el puerto 8080 de nuestra máquina obtenemos la respuesta del balanceador de carga, redirigiendo entre los servidores

Iteraciones: 1 - Resultado: 0

Iteraciones: 8 - Resultado: 13.47757340129

Iteraciones: 32 - Resultado: 117.65060957852

IMPORTANTE: Al igual que en el apartado anterior, se ha incorporado un script para automatizar directamente la creación y despliegue de este apartado. Se encuentra en el mismo directorio con el nombre `deploy-php-sinminikube.sh`

Para la parte de la **simulación del escalado**, en este caso, no quedaría de otra que realizarlo añadiendo manualmente una nueva réplica (instancia en el `docker-compose.yml`), podría ser algo así:

```
php4:
  build: .
  container_name: php4
```

y agregándole al balanceador de carga:

```
server php4:80;
```

Ahora por último, tendríamos que volver a reiniciar el contenedor: `docker compose up --build --force-recreate`

En cuanto a la **resiliencia a fallos**, si eliminamos uno de los contenedores, veremos que el balanceador de carga redirige automáticamente las peticiones a las réplicas que siguen activas. Esto permite “simular” cierta resiliencia, ya que el servicio continúa funcionando sin necesidad de crear nuevas instancias.

Sin embargo, si todas las réplicas fallan, debido a cómo funciona Docker, el servicio dejará de estar disponible. Y es precisamente ahí donde se evidencia una de las grandes ventajas que ofrece Kubernetes, ya que gestiona de forma automática la recuperación de contenedores caídos. Esta comparación y la anterior la desarrollaremos en detalle en el siguiente apartado.

6. Resultados obtenidos y análisis

De cara a realizar un análisis correcto y detallado de los resultados obtenidos en los diferentes despliegues, es importante partir de algunas consideraciones teóricas importantes. Por ello, antes de entrar en la parte práctica, cabe destacar que, a nivel teórico, sabemos que:

1. Los recursos usados por ambos servicios deben ser:

Aspecto	Con Kubernetes (Minikube)	Sin Kubernetes (Docker standalone / servidor)
CPU y memoria	Más alto, porque Minikube ejecuta un cluster con varios componentes (kubelet, kube-proxy, container runtime, etc.) que consumen recursos adicionales. Además el overhead del cluster.	Menor uso, solo corre el contenedor o la app directamente, sin capa extra.
Almacenamiento	Requiere configuración de volúmenes persistentes para datos duraderos, mayor complejidad.	Puede montar volúmenes directamente sin capas extra, más simple.
Red	Redes virtualizadas (CNI), que introducen algo de latencia y consumo de recursos.	Red simple, menos capas de abstracción.
Escalabilidad	Facilita el escalado automático (réplicas, autoscaling), con gestión eficiente de recursos.	Escalado manual, con más esfuerzo y menos automatización.

2. La facilidad de uso de ambos servicios deben ser:

Aspecto	Con Kubernetes	Sin Kubernetes
Configuración inicial	Requiere curva de aprendizaje, instalar y configurar minikube, kubectl, YAMLs, etc.	Más sencillo, solo Docker, docker-compose o servidor web.
Gestión y despliegue	Declarativo con YAML, permite control de versiones y actualizaciones fáciles.	Puede ser más manual (ejecutar comandos, scripts).
Desarrollo local	Minikube simula el entorno de producción Kubernetes, lo que ayuda a detectar problemas antes.	Entorno más simple pero diferente al de producción.

Monitoreo y logs	Kubernetes ofrece herramientas avanzadas para monitoreo, logs centralizados.	Logs y monitoreo más limitados, suelen requerir herramientas externas.
-------------------------	--	--

3. Los tiempos de respuesta de ambos servicios deben ser:

Aspecto	Con Kubernetes	Sin Kubernetes
Inicio de aplicación	Más lento, arranque de pods y servicios puede tardar varios segundos.	Más rápido, levantar contenedores o servidor es inmediato.
Tiempo de respuesta	Ligeramente más alto debido a la capa extra de red y orquestación.	Más bajo, acceso directo al contenedor o servicio.
Escalado y recuperación	Rápido, Kubernetes reinicia pods automáticamente y escala bajo demanda.	Manual y más lento, depende del administrador o scripts.

Por lo tanto, a modo resumen, podemos destacar que a nivel teórico:

Ventajas Kubernetes	Ventajas Sin Kubernetes
- Orquestación automatizada y escalabilidad	- Simplicidad y rapidez para despliegues pequeños
- Alta disponibilidad y auto-recuperación	- Menor consumo de recursos
- Facilita gestión de configuraciones y secretos	- Menos complejidad y curva de aprendizaje
- Entorno consistente entre desarrollo y producción	- Ideal para proyectos pequeños o pruebas rápidas

Una vez entendido a nivel teórico cómo deberían funcionar los distintos servicios, pasamos a obtener resultados prácticos reales a partir de los despliegues realizados anteriormente. Esto nos permitirá **comprobar y justificar que lo expuesto en el apartado teórico se cumple en la práctica**.

Para medir el comportamiento de las herramientas en el entorno Kubernetes con Minikube, una opción útil es utilizar el Dashboard que Kubernetes proporciona a través de una interfaz web. Esta herramienta permite visualizar en tiempo real métricas, recursos activos, uso de pods, servicios, etc. Se puede acceder fácilmente ejecutando el siguiente comando `minikube dashboard`

Estado de Carga de trabajo

Despliegues	Running: 1	Pods	Running: 4	Replica Sets	Running: 1
-------------	------------	------	------------	--------------	------------

Despliegues

Nombre	Imágenes	Etiquetas	Pods	Fecha de creación
php-apache	equipo13-apache-image:p6	-	3 / 3	an hour ago

Despliegues

Nombre	Imágenes	Etiquetas	Pods	Fecha de creación
php-apache	equipo13-apache-image:p6	-	3 / 3	an hour ago

Pods

Nombre	Imágenes	Etiquetas	Nodo	Estado	Reinicios	Utilización de CPU (núcleos)	Utilización de memoria (octetos)	Fecha de creación
php-apache-75cb8bb769-md4b8	equipo13-apache-image:p6	app: php-apache pod-template-hash: 75cb8bb769	minikube	Running	0	1,00m	11.77Mi	an hour ago
php-apache-75cb8bb769-qlm4x	equipo13-apache-image:p6	app: php-apache pod-template-hash: 75cb8bb769	minikube	Running	0	1,00m	12.24Mi	an hour ago
load-generator	busybox	run: load-generator	minikube	Running	0	0,00m	368.00Ki	an hour ago
php-apache-75cb8bb769-rkh5	equipo13-apache-image:p6	app: php-apache pod-template-hash: 75cb8bb769	minikube	Running	1	1,00m	41.49Mi	an hour ago

Replica Sets

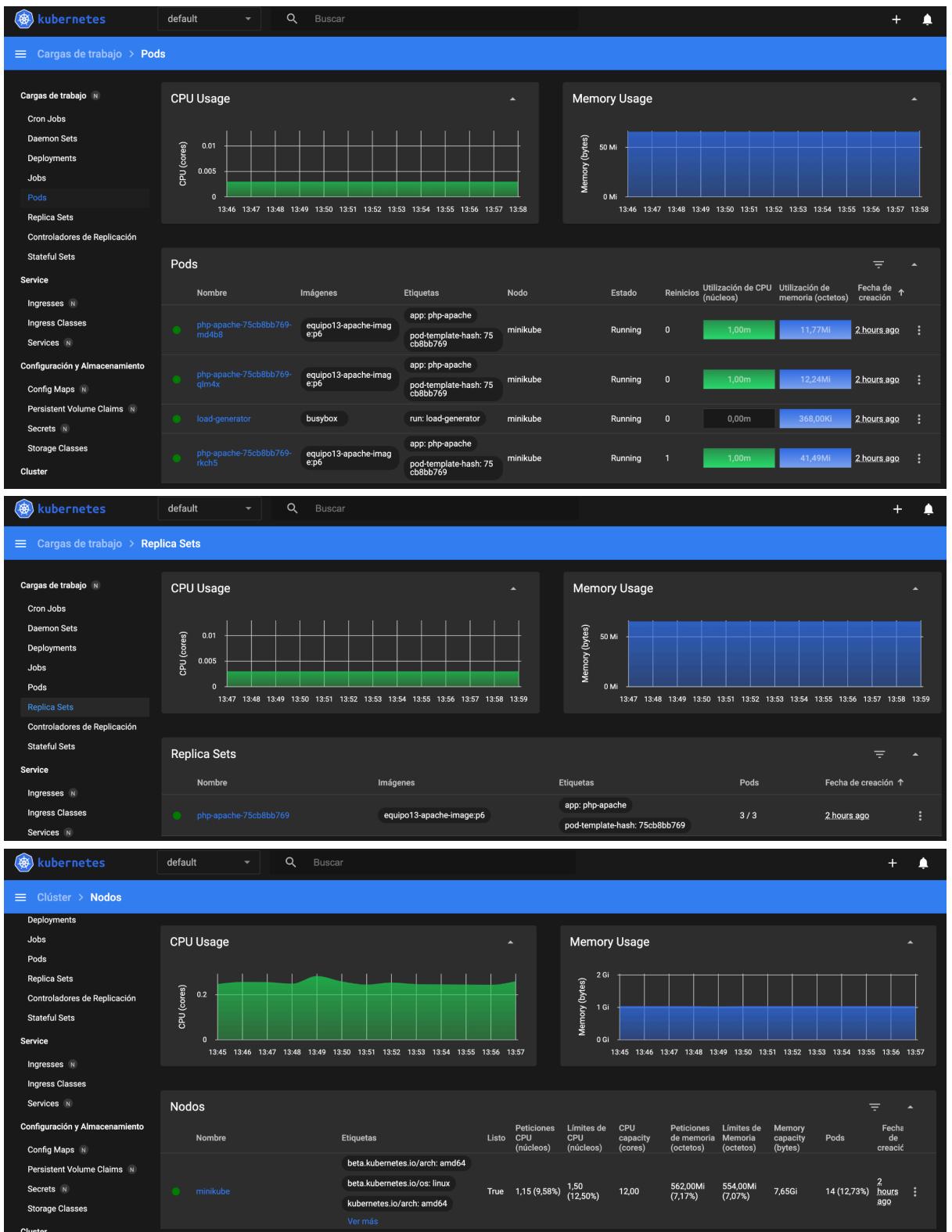
Nombre	Imágenes	Etiquetas	Pods	Fecha de creación
php-apache-75cb8bb769	equipo13-apache-image:p6	app: php-apache pod-template-hash: 75cb8bb769	3 / 3	an hour ago

CPU Usage

Memory Usage

Despliegues

Nombre	Imágenes	Etiquetas	Pods	Fecha de creación
php-apache	equipo13-apache-image:p6	-	3 / 3	2 hours ago



Sin embargo, para realizar un análisis aún más completo, incluyendo también las métricas del despliegue del contenedor usando Docker sin Kubernetes, hemos desarrollado un script `/test_rendimiento.sh`. Este script automatiza la recopilación de pruebas de rendimiento y guarda los resultados en diferentes archivos, lo que permite comparar con mayor precisión las diferencias entre ambos tipos de despliegue.

```

2025-06-06 14:21:25 Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal
o ➔ cat test-rendimiento.sh
#!/bin/bash

mkdir -p resultados

timestamp=$(date +"%Y%m%d_%H%M%S")
[
echo "Iniciando pruebas de rendimiento - $timestamp" | tee resultados/pruebas_$timestamp.log

# --- Kubernetes ---

# URL fija para túnel minikube
K8S_URL="http://127.0.0.1:52495"

# Comprobar si el puerto 52495 está escuchando para detectar si el túnel está activo
if ! nc -z 127.0.0.1 52495; then
    echo "ERROR: No se detecta el túnel minikube en 127.0.0.1:52495." | tee -a resultados/pruebas_$timestamp.log
    echo "Ejecuta 'minikube service php-apache-service --url' en otra terminal antes de correr este script." | tee -a resultados/pruebas_$timestamp.log
    exit 1
fi

# --- Docker ---

DOCKER_URL="http://localhost:8080"
DOCKER_CONTAINER="nginx-balancer" # Contenedor para stats Docker

# Parámetros ApacheBench
REQ_NUM=100
CONCURRENCY=10

# --- FUNCIONES ---

function curl_test() {
    local url=$1
    local name=$2
    echo "Test curl simple - $name - URL: $url" | tee -a resultados/pruebas_$timestamp.log
    for i in {1..3}; do
        echo "Curl intento $i:" | tee -a resultados/pruebas_$timestamp.log
        { time curl -s $url > /dev/null; } 2>&1 | tee -a resultados/pruebas_$timestamp.log
    done
    echo "" | tee -a resultados/pruebas_$timestamp.log
}

function ab_test() {
    local url=$1
    local name=$2
    local req_num=$3
    local concurrency=$4
    echo "Test carga ApacheBench - $name - URL: $url - Req: $req_num, Conc: $concurrency" | tee -a resultados/pruebas_$timestamp.log
    if ab -n $req_num -c $concurrency $url/ > resultados/ab_${name}_${timestamp}.log 2>&1; then
        echo "Resultado guardado en resultados/ab_${name}_${timestamp}.log" | tee -a resultados/pruebas_$timestamp.log
    else
        echo "ERROR: ApacheBench falló para $name. Revisa resultados/ab_${name}_${timestamp}.log" | tee -a resultados/pruebas_$timestamp.log
    fi
    echo "" | tee -a resultados/pruebas_$timestamp.log
    sleep 5
}

```

```

function k8s_stats() {
    echo "Estadísticas Kubernetes pods y nodo:" | tee -a resultados/pruebas_$timestamp.log
    kubectl top pods >> resultados/k8s_top_pods_$timestamp.log 2>&1
    [ kubectl top nodes >> resultados/k8s_top_nodes_$timestamp.log 2>&1
    echo "Guardado en resultados/k8s_top_pods_$timestamp.log y resultados/k8s_top_nodes_$timestamp.log" | tee -a resultados/pruebas_$timestamp.log
    echo "" | tee -a resultados/pruebas_$timestamp.log
}

function docker_stats_snapshot() {
    echo "Estadísticas Docker snapshot para contenedor $DOCKER_CONTAINER:" | tee -a resultados/pruebas_$timestamp.log
    docker stats --no-stream --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}" $DOCKER_CONTAINER > resultados/docker_stats_$DOCKER_CONTAINER_${timestamp}.log
    echo "Guardado en resultados/docker_stats_$DOCKER_CONTAINER_${timestamp}.log" | tee -a resultados/pruebas_$timestamp.log
    echo "" | tee -a resultados/pruebas_$timestamp.log
}

# --- EJECUCIÓN ---

# Validar Kubernetes
if ! kubectl get pods > /dev/null 2>&1; then
    echo "ERROR: Kubernetes no disponible. Omitiendo pruebas Kubernetes." | tee -a resultados/pruebas_$timestamp.log
else
    echo "== Pruebas para Kubernetes ==" | tee -a resultados/pruebas_$timestamp.log
    curl_test $K8S_URL "kubernetes"
    ab_test $K8S_URL "kubernetes" $REQ_NUM $CONCURRENCY
    k8s_stats
fi

# Validar Docker
if ! docker ps | grep -q $DOCKER_CONTAINER; then
    echo "ERROR: Contenedor Docker $DOCKER_CONTAINER no está corriendo. Omitiendo pruebas Docker." | tee -a resultados/pruebas_$timestamp.log
else
    echo "== Pruebas para Docker tradicional ==" | tee -a resultados/pruebas_$timestamp.log
    curl_test $DOCKER_URL "docker"
    ab_test $DOCKER_URL "docker" $REQ_NUM $CONCURRENCY
    docker_stats_snapshot
fi

echo "Pruebas completadas - resultados en carpeta 'resultados/'"

```

Los resultados obtenidos se han guardado en la carpeta `/resultados`. A continuación, procederé a analizarlos, aunque si se prefiere se pueden consultar directamente, en dicho directorio.

IMPORTANTE: Si se desea ejecutar el script automatizado para el test de rendimiento, es necesario ajustar una línea específica (mostrada en la siguiente captura de pantalla). Esto se debe a que, en macOS, el driver de Docker no expone directamente la IP del clúster, por lo que la URL de localhost solo funciona correctamente desde el propio Mac. Por lo tanto, se debe modificar dicha línea utilizando la IP correcta del túnel o red local que se haya abierto en tu máquina, para que las pruebas se realicen correctamente contra el contenedor desplegado.

```

2025-06-06 14:21:25 ✘ Nurias-MacBook-Pro in ~/Desktop/UNI/QUINTO/SEGUNDO CUATRI/SWAP/TrabajoFinal
o ➔ cat test-rendimiento.sh
#!/bin/bash

mkdir -p resultados

timestamp=$(date +"%Y%m%d_%H%M%S")

echo "Iniciando pruebas de rendimiento - $timestamp" | tee resultados/pruebas_$timestamp.log

# --- Kubernetes ---

# URL fija para túnel minikube
K8S_URL="http://127.0.0.1:52495"

# Comprobar si el puerto 52495 está escuchando para detectar si el túnel está activo
if ! nc -z 127.0.0.1 52495; then
    echo "ERROR: No se detecta el túnel minikube en 127.0.0.1:52495." | tee -a resultados/pruebas_$timestamp.log
    echo "Ejecuta 'minikube service php-apache-service --url' en otra terminal antes de correr este script."
    | tee -a resultados/pruebas_$timestamp.log
    exit 1
fi

```

A modo resumen, los resultados que hemos obtenido tras la ejecución del script han sido:

Métrica	Docker tradicional (nginx)	Kubernetes (Apache)
---------	----------------------------	---------------------

Requests per second (RPS)	900.24	125.01
Time taken for 100 requests	0.111 s	0.800 s
Mean time per request (ms)	11.1 ms	80 ms
Failed requests	0	0
CPU usage contenedor	0.00% (nginx-balancer)	1-2m cores (php-apache pods)
Memoria usada	2.3 MiB (nginx-balancer)	13-41 MiB (php-apache pods)
Uso CPU nodo	2%	2%
Uso memoria nodo	13%	13%

Lo que nos lleva al siguiente análisis final:

- Rendimiento (Throughput y Latencia):** Docker tradicional tiene un rendimiento mucho más alto (900 RPS frente a 125 RPS), además la latencia media es mucho menor en Docker tradicional (~11 ms vs ~80 ms). Esto es **consistente con lo mencionado anteriormente, de que los contenedores sin orquestación (menos capas y menos abstracción) tienden a responder más rápido para cargas pequeñas y simples, con menos sobrecarga.**
- Consumo de recursos:** El contenedor **nginx-balancer en Docker tradicional usa muy poca CPU y memoria. En cambio, en Kubernetes, los pods php-apache usan más memoria (13-41 MiB) y algo más de CPU**, probablemente porque Kubernetes introduce overhead de gestión, y tienes varios pods en paralelo. La memoria y CPU del nodo son similares, pero el entorno Kubernetes tiene más consumo total por la gestión del cluster.
- Escalabilidad y Resiliencia:** Aunque no se muestra directamente en estos resultados, **Kubernetes tiene ventajas en orquestación, auto-recuperación y escalabilidad** que no son evidentes en una sola prueba rápida. El tiempo por petición mayor en Kubernetes puede ser tolerable o mejorar con escalado horizontal (más pods), mientras que Docker tradicional es más rígido para escalabilidad teniendo que realizarse manualmente como se explicó en el anterior apartado.
- Simplicidad y uso:** **La prueba muestra que para cargas pequeñas y sencillas, Docker tradicional es más rápido y consume menos recursos**, apoyando la ventaja de “simplicidad y rapidez para despliegues pequeños”. Kubernetes, aunque más complejo y con overhead, facilita el manejo de configuraciones, despliegues y mantenimiento a largo plazo.

7. Conclusiones

El trabajo ha consistido en el análisis, despliegue y comparación práctica de dos tecnologías fundamentales para la gestión y ejecución de aplicaciones en contenedores: Docker tradicional y Kubernetes. Se han evaluado aspectos clave como el rendimiento, consumo de recursos, complejidad de gestión y adecuación a distintos escenarios de uso, mediante pruebas de carga y monitorización de recursos.

Los objetivos planteados, enfocados en entender las ventajas y limitaciones de cada enfoque, así como en comprobar su comportamiento real bajo diferentes condiciones, se han cumplido satisfactoriamente. La experimentación ha evidenciado que Docker tradicional es más eficiente para entornos de desarrollo, pruebas rápidas y cargas ligeras, ofreciendo simplicidad y menor consumo de recursos. En contraste, Kubernetes destaca en entornos de producción que demandan alta disponibilidad, escalabilidad automática y gestión avanzada, aunque con un coste en mayor latencia y consumo de recursos.

Sin embargo, el trabajo también ha mostrado que Kubernetes añade complejidad y overhead que no siempre se justifica, especialmente en proyectos pequeños o de corto plazo. Esta conclusión crítica reafirma la importancia de seleccionar la tecnología adecuada según el contexto y necesidades específicas, evitando sobre complicar soluciones cuando no es necesario.

Para futuras mejoras, sería recomendable profundizar en la optimización de configuraciones en Kubernetes para minimizar su impacto en rendimiento y consumo, así como explorar herramientas complementarias que faciliten su gestión. Además, una mejora clave a implementar sería la persistencia de sesiones, lo que permitiría mantener el estado de usuario a través de los distintos pods o contenedores, mejorando la experiencia en aplicaciones con estados dinámicos o interacciones continuas. Asimismo, ampliar el análisis a escenarios de cargas variables y más complejas permitiría obtener conclusiones más robustas y aplicables a casos reales más amplios.

En definitiva, este trabajo aporta una visión práctica y fundamentada que puede servir como guía para la toma de decisiones en la elección entre Docker sin Kubernetes y con Kubernetes, destacando sus fortalezas, limitaciones y casos de uso más adecuados.

8. Bibliografía

- Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), 2. <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>
- The Kubernetes Authors. (2025). *Kubernetes Documentation*. Kubernetes.io. <https://kubernetes.io/docs/home/>
- The Docker Authors. (2025). *Docker Documentation*. Docker.com. <https://docs.docker.com/>
- Apache Software Foundation. (2025). *Apache HTTP Server Version 2.4 Documentation*. <https://httpd.apache.org/docs/2.4/>
- The Apache Software Foundation. (2025). ApacheBench (ab) User Guide. <https://httpd.apache.org/docs/2.4/programs/ab.html>
- Minikube Contributors. (2025). *Minikube Documentation*. Kubernetes.io. <https://minikube.sigs.k8s.io/docs/>
- Red Hat. (2025). ¿Qué es Kubernetes?. <https://www.redhat.com/es/topics/containers/what-is-kubernetes>
- Red Hat. (2025). ¿Qué es Docker?. <https://www.redhat.com/es/topics/containers/what-is-docker>
- OpenWebinars. (2025). Kubernetes: Fundamentos y conceptos clave. <https://openwebinars.net/blog/kubernetes-fundamentos-y-conceptos-clave/>