

# PRÀCTICA 2:

Entrenament d'un Model i creació d'un Servidor  
amb Docker i Kubernetes

Nom 1: David Morillo Massagué

NIU 1: 1666540

Nom 2: Adrià Muro Gómez

NIU 2: 1665191

Usuari utilitzat a la pràctica: gixpd-ged-22

## Taula de continguts

<b>Introducció.....</b>	<b>3</b>
<b>Objectius.....</b>	<b>3</b>
<b>Desenvolupament.....</b>	<b>4</b>
Pas 1:.....	4
Docker.....	4
Minikube.....	5
Kubernetes.....	5
Funcionament dels tres serveis.....	6
Pas 2:.....	7
Creació dels Dockerfiles.....	7
Construcció de les Imatges Docker.....	8
Execució de les Aplicacions amb Docker.....	9
Verificació del Funcionament del Servidor.....	9
Pas 3:.....	11
ConfigMap (Mapa de configuració).....	11
Job (Feina).....	12
Deployment (Desplegament).....	12
Service (Servei).....	13
Aplicació dels canvis.....	14
Posada en marxa.....	15
Comprovació del servei.....	16
<b>Conclusions.....</b>	<b>16</b>

# Introducció

En aquesta pràctica, hem creat un servei per exposar un model entrenat en ciència de dades utilitzant Docker i Kubernetes. Aquest procés inclou la configuració i instal·lació de dependències, la creació d'imatges Docker per a entrenar i servir el model, i el desplegament final en un entorn Kubernetes.

## Objectius

L'objectiu principal d'aquesta pràctica és construir un servei escalable i eficient per exposar un model de ciència de dades, utilitzant contenidors Docker i Kubernetes per a la seva gestió i desplegament. Els objectius específics són:

- Instal·lar i configurar entorns de contenidorització: Configurar Docker, Minikube i Kubectl per crear una infraestructura de desenvolupament que permeti gestionar contenidors i orquestrar serveis de manera efectiva.
- Desenvolupar imatges Docker per a aplicacions de modelatge i serveis: Crear dues imatges Docker: una per entrenar un model i una altra per desplegar un servei Flask que permeti interactuar amb el model mitjançant API.
- Implementar el servei en un entorn de producció amb Kubernetes: Desplegar els contenidors utilitzant Kubernetes per aprofitar les funcionalitats d'escalabilitat i resiliència, incloent-hi la configuració de recursos de Kubernetes com ConfigMaps, Jobs, Deployments i Services.
- Aplicar bones pràctiques en contenidorització i desplegament: Utilitzar les millors pràctiques de Docker i Kubernetes per optimitzar el rendiment, la seguretat i la mantenibilitat del servei.

# Desenvolupament

## Pas 1:

Per començar, vam instal·lar i ens vam informar de les dependències necessàries:

## Docker

Docker és una eina per empaquetar aplicacions en "contenidors" amb tot el que necessiten per funcionar. Així, s'assegura que l'aplicació funcioni igual en qualsevol lloc.

Seguint la documentació oficial

<https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository>, vam instal·lar Docker. Això ho vam fer creant un script de shell per a facilitar i automatitzar la feina:

```

1 #!/bin/bash
2
3 # Add Docker's official GPG key:
4 sudo apt-get update
5 sudo apt-get install ca-certificates curl
6 sudo install -m 0755 -d /etc/apt/keyrings
7 sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
8 sudo chmod a+r /etc/apt/keyrings/docker.asc
9
10 # Add the repository to Apt sources:
11 echo \
12 "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
13 $(. /etc/os-release && echo "${VERSION_CODENAME}") stable" | \
14 sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
15 sudo apt-get update
16
17 sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
18
19 sudo docker run hello-world
  
```

A més, vam afegir l'usuari al grup sudo per permetre l'execució de comandes sense permisos root, seguint la documentació

<https://docs.docker.com/engine/install/linux-postinstall/>. Confirmem que tenim la instal·lació de Docker llesta amb aquesta comanda, sense permisos sudo:

```

adminp@adminp:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
  
```

## Minikube

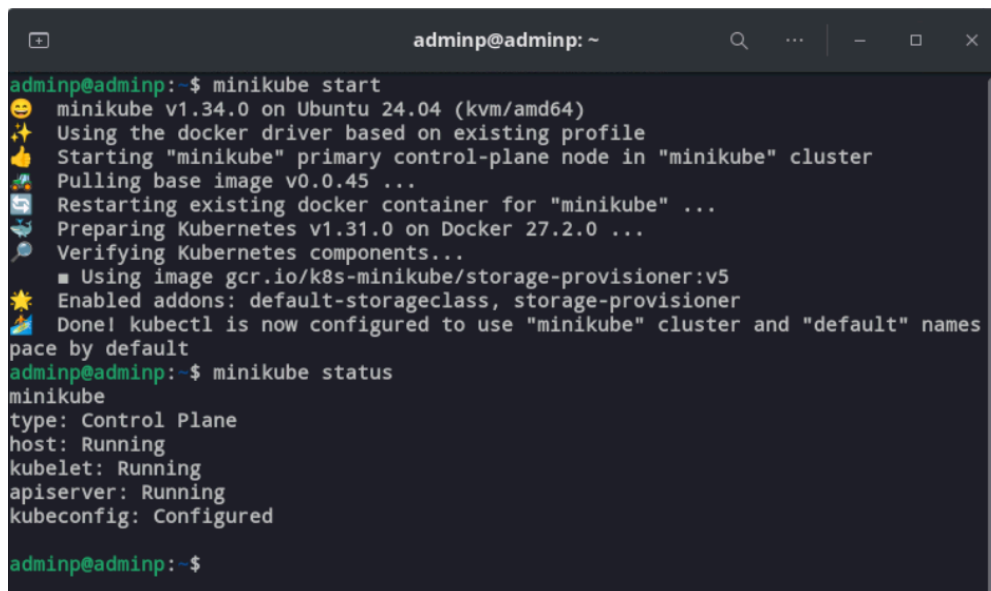
Permet crear un clúster de Kubernetes a l'ordinador per provar coses sense necessitat d'infraestructura real. És una manera senzilla de simular un entorn de Kubernetes localment.

Vam seguir els passos d'instal·lació

<https://minikube.sigs.k8s.io/docs/start/?arch=%2Flinux%2Fx86-64%2Fstable%2Fbinary+download>. Després d'executar següents comandes d'instal·lació:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

Vam executar *minikube start*, per a iniciar el servei, i *minikube status* per a verificar la seva correcta instal·lació:



```
adminp@adminp: ~
adminp@adminp:~$ minikube start
🐳 minikube v1.34.0 on Ubuntu 24.04 (kvm/amd64)
🌟 Using the docker driver based on existing profile
👍 Starting "minikube" primary control-plane node in "minikube" cluster
📡 Pulling base image v0.0.45 ...
🔄 Restarting existing docker container for "minikube" ...
🔧 Preparing Kubernetes v1.31.0 on Docker 27.2.0 ...
🔍 Verifying Kubernetes components...
   ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Enabled addons: default-storageclass, storage-provisioner
🏠 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
adminp@adminp:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
adminp@adminp:~$
```

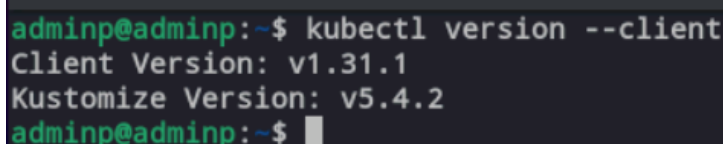
## Kubernetes

És una plataforma per gestionar contenidors a gran escala. S'encarrega de distribuir, escalar i mantenir els contenidors en funcionament.

Es va seguir també la documentació oficial

<https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/> per a la seva instal·lació.

Comprovem que s'ha instal·lat correctament al sistema i està llest per a funcionar amb minikube:



```
adminp@adminp:~$ kubectl version --client
Client Version: v1.31.1
Kustomize Version: v5.4.2
adminp@adminp:~$
```

## Funcionament dels tres serveis

De forma resumida, Docker crea els contenidors amb l'aplicació, Minikube crea un entorn local per Kubernetes, i Kubernetes gestiona aquests contenidors perquè l'aplicació funcioni de manera òptima, com en un entorn real.

## Pas 2:

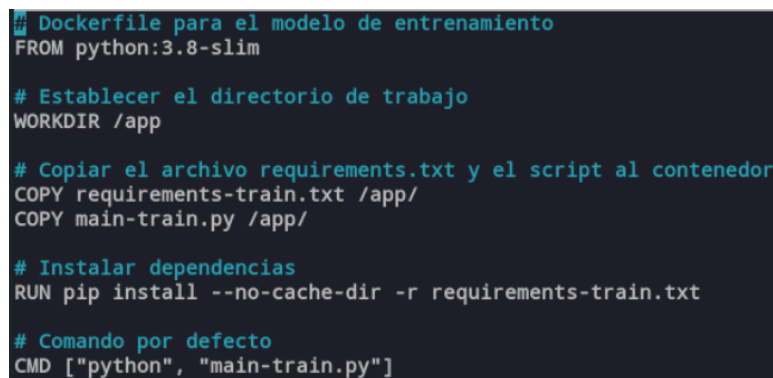
Per a exposar el nostre model entrenat en un entorn controlat, vam desenvolupar dues aplicacions separades encapsulades en imatges Docker. Aquest procés inclou la creació de Dockerfiles específics per a cada aplicació, la construcció de les imatges, i l'execució de les aplicacions en contenidors. Les aplicacions creades són:

1. **model-train**: Aquesta imatge conté l'aplicació encarregada d'entrenar el model i desar-lo.
2. **model-server**: Aquesta imatge inclou un servidor Flask que permet accedir al model a través d'una API amb dues rutes: una pàgina d'informació i una ruta per obtenir el resultat del model.

## Creació dels Dockerfiles

Per a cada aplicació es va crear un Dockerfile:

- Dockerfile per a model-train: Aquest Dockerfile inclou el codi necessari per entrenar el model. La configuració especifica el conjunt d'instruccions per muntar el script que entrena el model i el desa en disc per ser utilitzat posteriorment pel servidor.



```
Dockerfile para el modelo de entrenamiento
FROM python:3.8-slim

# Establecer el directorio de trabajo
WORKDIR /app

# Copiar el archivo requirements.txt y el script al contenedor
COPY requirements-train.txt /app/
COPY main-train.py /app/

# Instalar dependencias
RUN pip install --no-cache-dir -r requirements-train.txt

# Comando por defecto
CMD ["python", "main-train.py"]
```

- Dockerfile per a model-server: Aquest Dockerfile prepara un entorn amb Flask per proporcionar el servei d'API. La ruta base / mostra una pàgina HTML simple que explica el funcionament del servei, mentre que la ruta /model carrega el model i retorna el resultat segons els paràmetres d'entrada rebuts a través de l'URL. (Aquí pots utilitzar la captura del Dockerfile per a model-server).

```

# Dockerfile para el servidor de modelo
FROM python:3.8-slim

# Establecer el directorio de trabajo
WORKDIR /app

# Copiar el archivo requirements.txt y los archivos de la aplicación
COPY main-server.py /app/
COPY requirements-server.txt /app/

# Instalar dependencias
RUN pip install --no-cache-dir -r requirements-server.txt

# Comando por defecto
CMD ["flask", "--app", "main-server.py", "run", "--host=0.0.0.0"]

```

## Construcció de les Imatges Docker

Després de definir els Dockerfiles, vam crear les imatges Docker corresponents amb les comandes següents:

**Construcció de la imatge model-train:** La imatge es va construir amb la comanda *docker build -f Dockerfile -t model-train:default*.

```

adminp@adminip: ~/PROJECTE/docker-prac2/train_model$ docker build -f Dockerfile -t model-train:default .
[+] Building 23.1s (10/10) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 418B
=> [internal] load metadata for docker.io/library/python:3.8-slim
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.8-slim@sha256:1d52838af602b4b5a831beb13a0e4d073280665ea7be7f69ce2382f29c5a613f
=> => resolve docker.io/library/python:3.8-slim@sha256:1d52838af602b4b5a831beb13a0e4d073280665ea7be7f69ce2382f29c5a613f
=> => sha256:b5f62925bd0f63f48cc8acd5e87d0c3a07e2f229cd2fb0a9586e68ed17f45ee3 5.25kB / 5.25kB
=> => sha256:302e3ee498053a7b5332ac79e8efebec16e900289fc1ecd1c754ce8fa047fcab 29.13MB / 29.13MB
=> => sha256:030d7bdc20a63e3d22192b292d006a69fa333949f536d62865d1bd0506685cc 3.51MB / 3.51MB
=> => sha256:a3f1dfe736c5f959143f23d75ab522a60be2da902efac236f4fb2a153cc14a5d 14.53MB / 14.53MB
=> => sha256:1d52838af602b4b5a831beb13a0e4d073280665ea7be7f69ce2382f29c5a613f 10.41kB / 10.41kB
=> => sha256:314bc2fb0714b7807b75699c98f0c73817e579799f2d91567ab7e9510f5601a5 1.75kB / 1.75kB
=> => sha256:3971691a363796c39467aae4cdce6ef773273fe6bfc67154d01eb589befb912 248B / 248B
=> => extracting sha256:302e3ee498053a7b5332ac79e8efebec16e900289fc1ecd1c754ce8fa047fcab 2.2s
=> => extracting sha256:030d7bdc20a63e3d22192b292d006a69fa333949f536d62865d1bd0506685cc 0.2s
=> => extracting sha256:a3f1dfe736c5f959143f23d75ab522a60be2da902efac236f4fb2a153cc14a5d 1.1s
=> => extracting sha256:3971691a363796c39467aae4cdce6ef773273fe6bfc67154d01eb589befb912 0.0s
=> [internal] load build context
=> => transferring context: 76B
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements-train.txt /app/
=> [4/5] COPY main-train.py /app/
=> [5/5] RUN pip install --no-cache-dir -r requirements-train.txt
=> => exporting to image
=> => exporting layers
=> => writing image sha256:c9bfb5cb08ff8662faf47a34912a4064fd3c085b07fd3e12dbda017548ee1b23
=> => naming to docker.io/library/model-train:default

```

Aquesta comanda crea la imatge a partir del Dockerfile de model-train i la etiqueta com model-train:default.

**Construcció de la imatge model-server:** Per al servidor Flask, vam utilitzar la comanda *docker build -f Dockerfile -t model-server:default*.



```
adminp@adminp:~/PROJECTE/docker-prac2/server$ docker build -f Dockerfile -t model-server:default .
[+] Building 8.8s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 453B
=> [internal] load metadata for docker.io/library/python:3.8-slim
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.8-slim@sha256:1d52838af602b4b5a831beb13a0e4d073280665ea7be7f69ce2382f29c5a613f
=> [internal] load build context
=> => transferring context: 78B
=> CACHED [2/5] WORKDIR /app
=> [3/5] COPY main-server.py /app/
=> [4/5] COPY requirements-server.txt /app/
=> [5/5] RUN pip install --no-cache-dir -r requirements-server.txt
=> exporting to image
=> => exporting layers
=> => writing image sha256:99ed82da4cc88383248cad01187cde06923bda85c4723e3fbc418f266f4a57c3
=> => naming to docker.io/library/model-server:default
```

Aquesta instrucció genera la imatge `model-server:default` basada en el Dockerfile del servidor.

## Execució de les Aplicacions amb Docker

Amb les imatges creades, vam procedir a executar cada aplicació en contenidors separats:

### Execució del model d'entrenament:

La comanda `docker run -e MODEL_PATH=/model/model.npy -v /tmp:/model model-train:default`, executa el contenidor que entrena i desa el model. Quan finalitza, el model queda emmagatzemat per ser utilitzat pel servidor.

```
adminp@adminp:~/PROJECTE/docker-prac2$ docker run -e MODEL_PATH=/model/model.npy -v /tmp:/model model-train:default
Model trained successfully
Model Score: 0.8086921460343672
```

### Execució del servidor Flask:

Pel servidor utilitzem `docker run -e MODEL_PATH=/model/model.npy -v /tmp:/model model-server:default`. El servidor s'executa en el port 5000 (per defecte).

```
adminp@adminp:~/PROJECTE/docker-prac2$ docker run -e MODEL_PATH=/model/model.npy -v /tmp:/model model-server:default
* Serving Flask app 'main-server.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

## Verificació del Funcionament del Servidor

Un cop el servidor Flask s'ha iniciat, vam verificar-ne el funcionament comprovant que l'API respon correctament. En el buscador vam accedir a `https://172.17.0.2:5000` confirmant que el servidor estava habilitat.



Amb la següent comanda, vam verificar la sortida del model mitjançant una consulta a l'API:

```
curl 172.17.0.2:5000/model?minutes=5
```

Aquesta comanda envia una petició a la ruta /model amb el paràmetre minutes=5, i el servidor retorna correctament el resultat en format JSON. En el nostre cas ens va retornar el valor 22.8796259181197.

```
adminp@adminp:~$ curl 172.17.0.2:5000/model?minutes=5
{"spent":22.8796259181197}
```

Aquest procés garanteix que el model es pot entrenar i que el servidor Flask serveix correctament el model entrenat, la qual cosa ens prepara per al desplegament en Kubernetes.

### Pas 3:

Per desplegar el servei, el primer que vam fer després de comprovar el correcte funcionament dels les imatges Docker, va ser preparar-les per a poder-les desplegar amb Kubernetes. Per a això es van seguir els següents passos:

Guardar les imatges: Amb la comanda següent, vam guardar en format .tar cadascuna de les imatges de Docker que havien sigut prèviament creades:

```
adminp@adminp:~$ docker save -o model-train.tar model-train:default
adminp@adminp:~$ docker save -o model-server.tar model-server:default
```

Un cop amb les imatges guardades al propi directori, amb l'eina d'imatge load de Minikube vam recuperar aquestes imatges per a carregar-les a aquest servei amb les següents comandes:

```
adminp@adminp:~$ minikube image load model-train.tar
adminp@adminp:~$ minikube image load model-server.tar
```

Per a verificar que s'havien carregat correctament executem:

```
adminp@adminp:~$ minikube image ls
registry.k8s.io/pause:3.10
registry.k8s.io/kube-scheduler:v1.31.0
registry.k8s.io/kube-proxy:v1.31.0
registry.k8s.io/kube-controller-manager:v1.31.0
registry.k8s.io/kube-apiserver:v1.31.0
registry.k8s.io/etcd:3.5.15-0
registry.k8s.io/coredns/coredns:v1.11.1
gcr.io/k8s-minikube/storage-provisioner:v1.11.1
docker.io/library/model-train:default
docker.io/library/model-server:default
```

Els següents passos són la creació d'arxius de configuració per al desplegament del servei de Kubernetes:

### ConfigMap (Mapa de configuració)

Documentació utilitzada: <https://kubernetes.io/docs/concepts/configuration/configmap/>

Vam crear un l'arxiu configmap.yaml per a emmagatzemar els valors necessaris per identificar la ruta del model, tant per la feina com per al desplegament. Aquest pas és necessari i realment important en aplicacions en el món real, ja que permet que qualsevol entorn virtual de Kubernetes, pugui accedir a la variable de l'entorn **MODEL\_PATH**, i el valor d'aquest li retorni **/model/model.npy**. Això resulta realment útil si decidim canviar de model o el seu directori, i no haver de configurar cada servei que l'utilitza individualment, sinó que només hem d'indicar el nou path del model en el ConfigMap. L'arxiu resultant seria aquest:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: model-config
data:
  MODEL_PATH: "/model/model.npy"
```

## Job (Feina)

Documentació utilitzada: <https://kubernetes.io/docs/concepts/workloads/controllers/job/>

Aquest Job entrena el model, fent servir la imatge *model-train:default*, i el desa en un volum compartit. S'obté d'un ConfigMap anomenat *model-config* i proporciona la ruta del model al contenidor (pas anterior). Finalment, l'argument *OnFailure* reintenta el Job si falla:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: model-train-job
spec:
  template:
    spec:
      containers:
        - name: model-train
          image: model-train:default
          env:
            - name: MODEL_PATH
              valueFrom:
                configMapKeyRef:
                  name: model-config
                  key: MODEL_PATH
          restartPolicy: OnFailure
```

## Deployment (Desplegament)

Documentació utilitzada:

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

Per servir el model entrenat, vam crear un Deployment amb 3 rèpliques, executant la imatge *model-server:default*, que serveix el model. Com en el Job, vam afegir la variable *MODEL\_PATH* i definim recursos per a cada rèplica. L'argument *Liveness Probe* comprova que el servidor està viu fent una petició HTTP a la ruta */* al port 5000. Si falla, Kubernetes reinicia el pod. L'arxiu *deployment.yaml* quedaria d'aquesta manera:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-server-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: model-server
  template:
    metadata:
      labels:
        app: model-server
    spec:
      containers:
        - name: model-server
          image: model-server:default
          env:
            - name: MODEL_PATH
              valueFrom:
                configMapKeyRef:
                  name: model-config
                  key: MODEL_PATH
          livenessProbe:
            httpGet:
              path: /
              port: 5000
          readinessProbe:
            httpGet:
              path: /
              port: 5000
          restartPolicy: Always
```

## Service (Servei)

Documentació utilitzada: <https://kubernetes.io/docs/concepts/services-networking/service/>  
Finalment, vam configurar un Servei que exposa les 3 rèpliques del model. Per accedir-hi, es pot utilitzar `kubectl port-forward service/modelserver 5000:5000`, facilitant així l'accés al servei dins de l'entorn Kubernetes. Amb "port: 5000", exposa el servei al port 5000 dins del clúster i redirigeix el tràfic al mateix port (targetPort: 5000) dels pods:

```
apiVersion: v1
kind: Service
metadata:
  name: model-server-service
spec:
  selector:
    app: model-server
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
```

## Aplicació dels canvis

En aquestes altures de la configuració, en nostre directori té la següent estructura:

```
adminp@adminp:~/docker-prac2$ tree
.
├── configmap.yaml
├── deployment.yaml
├── job.yaml
├── model
│   ├── Dockerfile
│   ├── main-train.py
│   └── requirements-train.txt
├── model-server.tar
├── model-train.tar
├── server
│   ├── Dockerfile
│   ├── main-server.py
│   └── requirements-server.txt
└── service.yaml

3 directories, 12 files
```

Un cop realitzat els canvis necessaris per al desplegament del servei, en la mateixa carpeta en la que tenim tots els arxius .yaml, executem la comanda següent:

```
adminp@adminp:~/docker-prac2$ kubectl apply -f .
configmap/model-config unchanged
deployment.apps/model-server-deployment unchanged
job.batch/model-train-job unchanged
service/model-server-service unchanged
```

Això actualitza les configuracions del servei que s'està executant, aplicant els canvis que hem realitzat en aquest pas de l'informe.

## Posada en marxa

Per a posar en marxa el servei creat, primer caldrà assegurar-nos que el servei de Kubernetes està en marxa, per a això executem les següents comandes:

```
adminp@adminp:~/docker-prac2$ minikube status

minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

```
adminp@adminp:~/docker-prac2$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
model-server-deployment-5f44c47958-72bg6   1/1     Running   9 (102m ago)   5h
model-server-deployment-5f44c47958-7mzb9   1/1     Running   9 (102m ago)   5h
model-server-deployment-5f44c47958-88v7g   1/1     Running   9 (102m ago)   5h
```

En el cas de no estar ja iniciat, iniciarem Minikube amb *minikube start*, i si hi hagués cap problema en l'estatus dels pods de Kubernetes, una solució que podria funcionar, seria reiniciar Minikube amb *minikube stop*, i *minikube start* altre cop. Aquesta última comanda ens va ajudar a solventar alguns problemes en l'etapa final de desplegament.

Per a la redirecció de ports, per a accedir des de localhost:5000, necessitem executar la comanda següent:

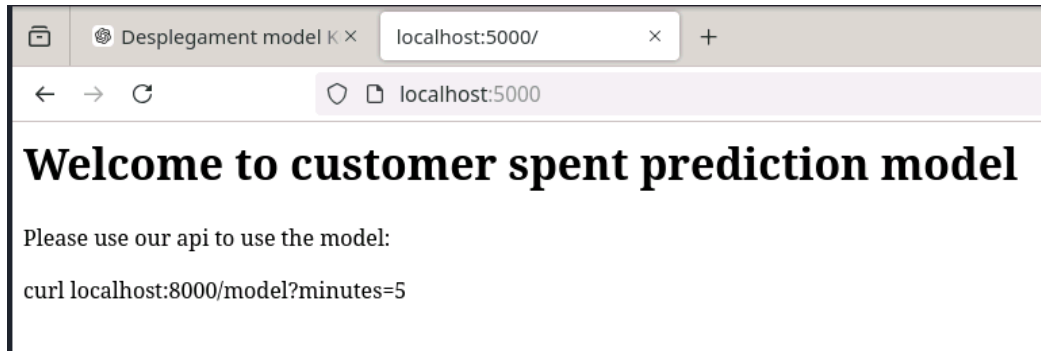
```
adminp@adminp:~$ kubectl port-forward service/model-server-service 5000:5000
Forwarding from 127.0.0.1:5000 -> 5000
Forwarding from [::1]:5000 -> 5000
```

Aquesta comanda redirigeix el port local 5000 al port 5000 del servei de Kubernetes anomenat model-server-service. Això et permet accedir al servei model-server-service des del teu ordinador, com si estigués funcionant localment al port 5000.

## Comprovació del servei

Per a verificar el correcte funcionament del servei desplegat, només cal accedir al port 5000 en un navegador qualsevol, el qual ens mostra que el servidor de Kubernetes que hem desplegat.

Accedim a l'API a través de <http://localhost:5000/>:



## Conclusions

En aquesta pràctica, hem aconseguit crear un servei per entrenar i oferir el resultat d'un model mitjançant Docker i Kubernetes. Hem configurat un entorn de contenidorització eficient, construint dues imatges Docker: una per entrenar el model i una altra per desplegar un servidor Flask que proporciona accés a l'API del model. Hem utilitzat la variable d'entorn MODEL\_PATH per facilitar la gestió de les rutes dels models.

El desplegament en Kubernetes ha estat un èxit, utilitzant ConfigMaps, Jobs, Deployments i Services per assegurar una gestió òptima dels contenidors. Hem validat el funcionament del servei, verificant que l'API respon correctament a les peticions.

El procés ha demostrat la importància de seguir bones pràctiques en la contenidorització i el desplegament per garantir la seguretat, escalabilitat i mantenibilitat del servei en un entorn de producció.