Clase 198 — Docker para empaquetar modelos

Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Docker + Nilsson, Docker Deep Dive.

Duración estimada: 75 min.

Clase 198 — Docker para empaquetar modelos

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Docker + Nilsson, Docker Deep Dive. Duración estimada: 75 min.*

Objetivo

Empaquetar un modelo entrenado + su runtime (Python, deps, código) en una imagen Docker reproducible, optimizada (multi-stage build, capas cacheadas, imagen <500 MB), y segura (non-root user, no secrets baked in). El resultado se corre idéntico en tu laptop, en CI, y en producción.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un Dockerfile correcto para un servicio ML: base slim, multi-stage, layer caching, non-root.
- Diferenciar COPY de ADD, RUN de CMD de ENTRYPOINT, y cuándo usar cada uno.
- Reducir tamaño de imagen (de 2 GB a <500 MB) con python:3.12-slim, .dockerignore, y multi-stage builds.
- Versionar imágenes con tags semánticos (mymodel:1.2.3) y digests (@sha256:...) — y por qué :latest es trampa en producción.
- Diagnosticar image not building, image too big, slow rebuild con docker history, dive, docker scout.

Temas

#	Tema	Por qué importa
1	Imagen, capa, container, registry	Vocabulario base.
2	Dockerfile: FROM, COPY, RUN, CMD, ENTR	Las instrucciones que importan.
3	Layer caching: orden de instrucciones	Diferencia entre build de 2 min vs 30 s.
4	Multi-stage build	Separar build-deps de runtime-deps.
5	Imágenes base: python:slim vs distroless v	Trade-off tamaño / compat / debug.
6	Security: non-root user, secrets via env/m	No bakear API keys en la imagen.

Definiciones y características

- Imagen: filesystem inmutable + metadata (entrypoint, env, ports). Compuesta por capas read-only apiladas (cada RUN/COPY es una capa).
- Container: instancia ejecutándose de una imagen + capa writable encima. Efímero por default — al borrarlo, se pierde lo escrito (salvo volúmenes).
- Layer caching: si la instrucción N de tu Dockerfile no cambió ni cambiaron las anteriores, Docker reusa la capa cacheada. Regla de oro: lo que cambia poco va primero, lo que cambia mucho va al final.
- Multi-stage build: usar varias secciones FROM ... AS stage y copiar artefactos de una a otra con COPY --from=stage. Permite usar imagen grande para compilar y imagen pequeña para correr.
- .dockerignore: como .gitignore pero para docker build. Crítico — sin él, mandás todo el repo (incluido .git/, __pycache__/, datasets) como build context.

- Tag semántico (myapp:1.2.3): fija la versión exacta. Digest (myapp@sha256:abc...): fija el contenido exacto (inmutable). Producción usa digest. :latest no tiene garantía — puede cambiar entre docker pull y docker run.
- Distroless (gcr.io/distroless/python3): imagen sin shell, sin package manager, casi sin nada. Reduce attack surface; complica debug (no podés exec un shell).

Dataset / recursos

- Modelo: RandomForestClassifier entrenado en Iris, serializado con joblib.
- API: FastAPI sirviendo POST /predict.
- Herramientas: docker>=24, opcional dive, docker scout.

Ejercicios

1. Dockerfile básico: empaquetá un script predict.py que carga model.pkl y predice una fila random. FROM python:3.12-slim, instalá deps de requirements.txt, COPY . /app, CMD ["python", "predict.py"]. Build con docker build -t miml:v1 . y corré.
2. Layer caching: cambiá una línea en predict.py sin tocar requirements.txt. Rebuildá. Confirmá que la capa de pip install se reusa (mensaje "CACHED").
3. Multi-stage: separá en dos stages: builder (FROM python:3.12 AS builder, instalá deps con compiladores) y runtime (FROM python:3.12-slim, copiá solo site-packages del builder). Compará tamaño con docker images.
4. Non-root: agregá RUN useradd -m app && USER app antes del CMD. Verificá con docker run --rm miml:v3 whoami.
5. Tags y digest: hacé docker push miml:v1 a Docker Hub. Obtené el digest con docker inspect miml:v1 --format '{{index .RepoDigests 0}}'. Discutí por qué deploys de producción referencian el digest.

Homework verificable

Repo con:

1. Dockerfile multi-stage que produce imagen <500 MB para un modelo sklearn + FastAPI.
2. .dockerignore que excluye .git, __pycache__, *.ipynb, data/raw, mlruns/.
3. docker-compose.yml que levanta el servicio en :8000 con healthcheck.
4. README del repo con comandos build, run, push, y la URL/digest de la imagen pushed.
5. Output de dive miml:latest (o docker history) mostrando que ninguna capa única pesa más de 200 MB.

Criterio de aceptación: docker run --rm -p 8000:8000 miml:v1 levanta y responde a curl localhost:8000/predict -X POST -d '{"features":[5.1,3.5,1.4,0.2]}' -H 'content-type: application/json'. El contenedor corre como app, no como root (docker exec ... whoami devuelve app).

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Imagen pesa 2.5 GB con un modelo de 50 MB	Estás usando python:3.12 (no slim) y/o tra
Cada build rebuildea todo aunque no cambié	Tenés COPY . /app antes de RUN pip install
docker build manda 3 GB de context	Falta .dockerignore. Fix: crearlo con al m

Error ModuleNotFoundError en el container	El pip install corrió en otra versión de P
Container corre como root	No agregaste USER al Dockerfile. Fix: RUN
:latest apunta a algo distinto que ayer	:latest es mutable. Otro docker push myimg
COPY no encuentra model.pkl	El path es relativo al build context (el .

Preguntas frecuentes

¿slim, alpine, o distroless?

slim (Debian-based, sin doc/locales): default razonable, ~120 MB, glibc, compatible con wheels precompilados. alpine (musl): 50 MB pero usa musl, no glibc — wheels precompilados de muchas libs Python no funcionan. distroless: la imagen runtime más chica y segura; complica debugging. Para ML: slim salvo razón fuerte.

¿Docker o Podman?

Podman es API-compatible con Docker (alias podman ↔ docker), rootless por default, sin daemon. Para individuales: indistinto. Para producción Kubernetes: el OCI runtime que use el cluster (containerd, CRI-O). Las imágenes son OCI, no "Docker images" — funcionan en cualquier OCI runtime.

¿Cómo paso credenciales al container?

(1) Variables de entorno (docker run -e API_KEY=...) — visible en docker inspect. (2) Secrets mount (--secret, Docker Swarm/Kubernetes Secret) — preferido. (3) IAM role (en EC2/EKS) — el container hereda credenciales sin que las escribas. Nunca: ENV API_KEY=xxx en el Dockerfile, queda en una capa para siempre.

¿Tamaño del modelo dentro o fuera de la imagen?

Modelos pequeños (<200 MB): adentro, simple. Modelos grandes o que cambian seguido: afuera, en S3/MLflow, bajados en entrypoint. Trade-off: adentro = inmutable + lento de pull; afuera = imagen liviana pero acoplamiento con storage.

¿CMD vs ENTRYPOINT?

ENTRYPOINT es el "binario" fijo del container; CMD son sus "args default". Combinados: ENTRYPOINT ["python"] + CMD ["app.py"] → docker run img otro.py ejecuta python otro.py. Para imagen de modelo: ambos juntos, o solo CMD ["python", "app.py"] si querés que sea fácilmente overrideable.

¿Por qué docker scout o trivy?

Escanean la imagen contra CVE conocidas. Crítico antes de subir a producción — una python:3.12-slim de hace 6 meses tiene 200 CVE conocidas; rebuildear con la base actual elimina la mayoría. CI debería tener un step trivy image myimg:tag --severity HIGH,CRITICAL --exit-code 1.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11 — Model Deployment and Prediction Service.
- Docker official docs — Best practices for writing Dockerfiles.
- Dive — TUI para inspeccionar capas y eficiencia.
- Docker Scout / Trivy — vulnerability scanning.

- distroless — imágenes mínimas de Google.

Siguiente clase

Clase 199 — APIs con FastAPI sirviendo modelos

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import os, shutil, tempfile
from pathlib import Path
WORK = Path(tempfile.gettempdir()) / 'docker_demo'
if WORK.exists(): shutil.rmtree(WORK)
WORK.mkdir(); os.chdir(WORK)

# 1. Entrenar un modelo y guardarlo
import joblib
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
X, y = load_iris(return_X_y=True)
m = RandomForestClassifier(n_estimators=50, random_state=42).fit(X, y)
joblib.dump(m, 'model.pkl')
print('model.pkl:', Path('model.pkl').stat().st_size, 'bytes')
```

Archivos complementarios

- notebook.ipynb