Cheat sheet de Docker

Para escribir un Dockerfile con la documentación abierta y diagnosticar una falla en minutos. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y, cuando se puede, un laboratorio opcional para tu Mac. Después de los errores está tu laboratorio Iris completo, archivo por archivo, con lo que ya probaste y lo que falta. En la versión web, al final hay un diagnosticador de síntomas y un revisor de Dockerfile.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

0 Números y códigos de bolsillo

¿Qué números conviene saberse de memoria para leer un fallo en segundos?

SINTAXIS CON VARIABLES

`docker run --rm` *imagen* *comando* **`; echo "exit=$?"`**

El exit code del contenedor llega a tu shell: `125`, `126` y `127` los pone Docker, y `255`, runc; cualquier otro es de tu proceso. Si el daemon no corre, el CLI sale con 1 (antes de la 29.1.3, `docker run` daba 125 o 127).

`docker ps -a --filter name=` *contenedor*

STATUS dice `Exited (137) 5 seconds ago`: el código queda ahí aunque el contenedor ya no corra.

`docker stop -t` *segundos* *contenedor*

SIGTERM, espera *segundos* (10 por defecto en Linux) y después SIGKILL.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
imagen	imagen a correr	`classifier-gateway:3f9c2ab (en el lab, busybox:1.38)`
comando	comando que reemplaza al CMD de la imagen	`python -c "import app.main"`
contenedor	nombre o ID del contenedor	`gw`
segundos	plazo entre SIGTERM y SIGKILL	`30 para el gateway: drena hasta 20 s y en k8s tiene 30 s tras el SIGTERM`
$n_{\text{señal}}$	número de la señal que mató al proceso (kill -l las lista)	`9 = SIGKILL, 15 = SIGTERM`
t_{gracia}	plazo de SIGTERM a SIGKILL	`10 s en Docker y en compose; en k8s/gateway.yaml, 45 − 15 de preStop = 30 s`

FÓRMULAS

$$\text{exit code} = 128 + n_{\text{señal}} \quad \text{(cuando una señal mató al proceso)}$$

$$130 = 128 + 2\ \text{(INT)} \qquad 137 = 128 + 9\ \text{(KILL)} \qquad 139 = 128 + 11\ \text{(SEGV)} \qquad 143 = 128 + 15\ \text{(TERM)}$$

$$t_{\text{SIGKILL}} = t_{\text{SIGTERM}} + t_{\text{gracia}} \qquad t_{\text{gracia}} = 10\ \text{s por defecto}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`0`	Terminó bien.	`un job que procesó todo y salió`
`1 u otro`	Lo decidió tu proceso: excepción no capturada, sys.exit(1) o su propio código. La causa está en docker logs.	`ImportError al arrancar main.py`
`125`	Falló Docker y el contenedor nunca corrió: flag inválido, nombre en uso, imagen inexistente o pull denegado, error del daemon.	`docker run --foo busybox:1.38`

PIEZA	QUÉ HACE	EJEMPLO
126	El comando existe pero no se puede ejecutar: sin permiso de ejecución, o es un directorio.	`docker run busybox:1.38 /etc`
127	El comando no existe en la imagen o no está en el PATH.	`exec: "uvicorn": executable file not found in $PATH`
130	128 + 2 (SIGINT): lo cortaste con Ctrl+C.	`docker run -it … y Ctrl+C`
137	128 + 9 (SIGKILL): OOM killer, docker kill o fin del plazo de docker stop.	`OOMKilled true en docker inspect = memoria`
139	128 + 11 (SIGSEGV): segfault en código nativo, como una extensión C/C++ o una librería incompatible.	`un .so compilado contra otra versión`
143	128 + 15 (SIGTERM): el proceso murió por el SIGTERM sin atraparlo.	`docker stop de un sleep corrido con --init`
255	Lo devuelve runc (no está en la tabla de Docker): el contenedor ya arrancó, pero el kernel no pudo ejecutar el entrypoint.	`docker logs: exec /app/run.sh: exec format error (otra arquitectura), o no such file or directory (shebang con CRLF)`
`Docker.raw`	Disco de la VM Linux de Docker Desktop con imágenes, contenedores, volúmenes y caché; docker system df desglosa qué lo ocupa.	`Settings → Resources → Advanced → Disk image location (por defecto ~/Library/Containers/com.docker.docker/Data/vms/0/data)`
`/var/lib/docker/volumes`	La ruta que muestra docker volume inspect existe dentro de la VM, no en tu Mac.	`Mountpoint: /var/lib/docker/volumes/datos/_data`
BuildKit	Builder por defecto de Docker Desktop y de Docker Engine (desde 23.0): cache mounts, secretos, etapas en paralelo.	`docker build ya usa BuildKit`
failed to connect to the docker API	No existe el socket del daemon: Docker Desktop no está corriendo. El CLI sale con 1 (docker run también, desde la 29.1.3).	`at unix:///Users/…/.docker/run/docker.sock; check if the path is correct and if the daemon is running`

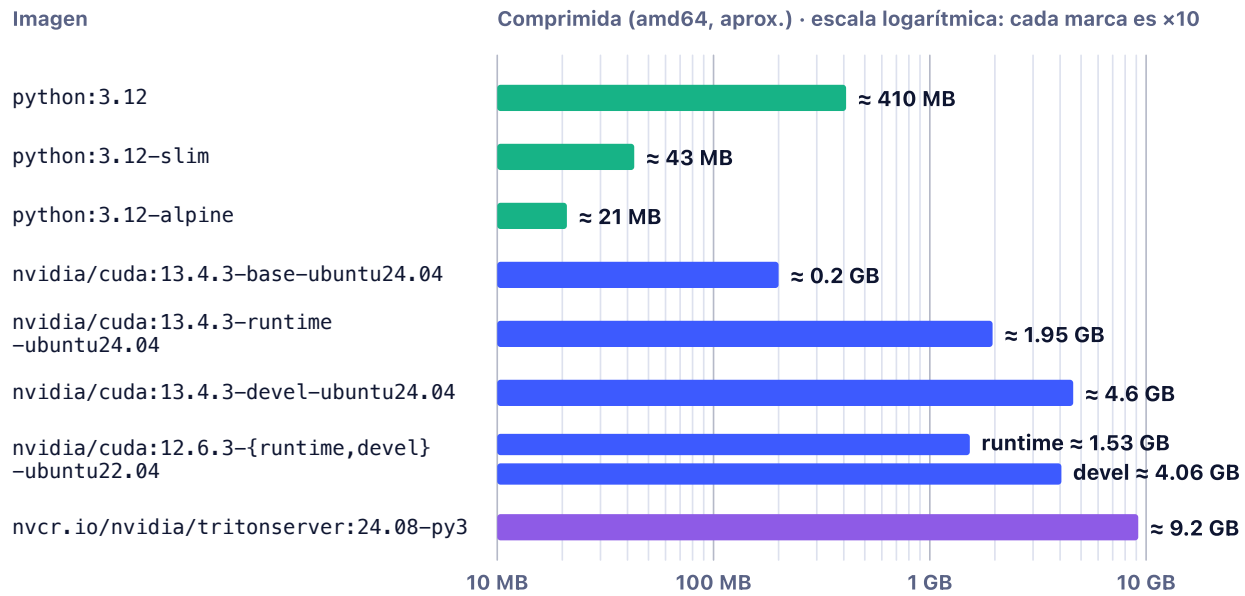


IMAGEN	COMPRIMIDA (AMD64, APROX.)	QUÉ TRAE	ÚSALA PARA
`python:3.12`	≈ 410 MB	Debian + toolchain (buildpack-deps): gcc, headers	La etapa builder si algo se compila
`python:3.12-slim`	≈ 43 MB	Debian mínimo + Python	La final de casi cualquier servicio Python

IMAGEN	COMPRIMIDA (AMD64, APROX.)	QUÉ TRAE	ÚSALA PARA
python:3.12-alpine	≈ 21 MB	Alpine (musl) + Python	Herramientas chicas; no para ML (sección 3)
nvidia/cuda:13.4.3-base-ubuntu24.04	≈ 0.2 GB	Lo mínimo de CUDA (cudart)	Traes el resto tú
nvidia/cuda:13.4.3-runtime-ubuntu24.04	≈ 1.95 GB	Librerías de CUDA de runtime	La final de una imagen GPU que espera CUDA instalado
nvidia/cuda:13.4.3-devel-ubuntu24.04	≈ 4.6 GB	+ nvcc, headers y librerías estáticas	Solo la etapa builder
nvidia/cuda:12.6.3-{runtime,devel}-ubuntu22.04	≈ 1.53 / 4.06 GB	Lo mismo con CUDA 12.6	Nodos con driver anterior a la rama 580 (CUDA 13 la exige)
nvcr.io/nvidia/tritonserver:24.08-py3	≈ 9.2 GB	Triton con sus backends	El serving del C01 (cifra del módulo 5 del curso)

En disco ocupan más: descomprimidas, ≈ 2–3× (aprox.). Con el image store de containerd (el default de Docker Desktop), docker images muestra DISK USAGE (en disco: capas descomprimidas y también las comprimidas) y CONTENT SIZE (las capas comprimidas). En arm64, las de Python pesan parecido; las de CUDA, más.

Tamaños comprimidos para linux/amd64, leídos en Docker Hub en octubre de 2026 (aprox.): es lo que viaja por la red en cada pull. Los tags se republican con parches, así que cambian un poco.

EJEMPLO REAL · PARAR EL GATEWAY DEL C01 SIN CORTAR REQUESTS

BASH · ZSH **reference/c01/gateway/Dockerfile (CMD) y reference/c01/k8s/gateway.yaml**

```
# El CMD del gateway drena hasta 20 s tras el SIGTERM (--timeout-graceful-shutdown 20);
# docker stop solo espera 10 s por defecto: dale los mismos 30 s que tiene en k8s
docker stop -t 30 gw

# ¿Cómo terminó? STATUS dice "Exited (código) …"; OOMKilled=true confirma memoria
docker ps -a --filter name=gw
docker inspect -f '{{.State.ExitCode}} {{.State.OOMKilled}}' gw

# En Kubernetes: 45 s de plazo total, y el preStop (sleep 15) gasta 15 antes del SIGTERM
grep -n -e terminationGracePeriodSeconds -e sleep k8s/gateway.yaml
```

- `--timeout-graceful-shutdown 20` está en el CMD del gateway: es el tiempo que uvicorn se da para terminar los requests en curso.
- `docker stop` espera 10 s y manda SIGKILL: con requests en vuelo, al gateway dale `-t 30`. En compose el equivalente es `stop_grace_period` (también 10 s por defecto); el `docker-compose.yml` del C01 no lo fija, así que `stop_grace_period: 30s` sería un agregado tuyo.
- `137` con `OOMKilled true` confirma memoria. Con `false` suele ser el plazo vencido o un `docker kill`, pero si nadie hizo stop ni kill no descartes la memoria: containerd puede perder el evento de OOM.
- Si lo arrancaste con `--rm`, después del stop ya no existe: para ver su exit code, córrelo sin `--rm`.
- Por qué 30: en el clúster manda `terminationGracePeriodSeconds: 45`, que empieza a contar con el preStop (`sleep 15`); tras el SIGTERM, a uvicorn le quedan 30 s.

CÓMO RECORDARLA

Por encima de 128 (salvo 255), resta 128 y busca la señal: **137 = 9 = KILL** (memoria, o se acabó el plazo) y **143 = 15 = TERM**. `125`, `126` y `127` dicen que Docker no pudo arrancar tu comando: falla de Docker, no es ejecutable, no existe.

TRAMPA

Leer `137` como "falta de memoria" sin mirar `OOMKilled`: también da 137 un `docker stop` cuyo proceso ignoró el SIGTERM durante los 10 s. Y en Mac, buscar `/var/lib/docker` en Finder: esa ruta vive dentro de la VM.

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Docker: construir, correr, depurar · M5 · Reducir tamaño (y por qué importa para el cold start)

La cadena

¿Qué comandos llevan tu Dockerfile hasta un contenedor corriendo en otra máquina?

SINTAXIS CON VARIABLES

`docker build --platform` *plataforma* **`-t`** *repo*`:`*tag* *contexto*

Dockerfile + contexto → imagen. *plataforma* es la del nodo, no la de tu Mac.

`docker tag` *repo*`:`*tag* *registry*`/`*repo*`:`*tag*

Un segundo nombre para la misma imagen (mismo ID): el registry va dentro del nombre.

`docker push` *registry*`/`*repo*`:`*tag*

Sube las capas que faltan y termina con *tag*`: digest: sha256:… size: …`.

`docker run -d --name` *nombre* **`-p`** *puerto_host*`:`*puerto_contenedor* *registry*`/`*repo*`@sha256:`*digest*

Si la imagen no está local, `run` hace el pull; por digest, de exactamente ese contenido.

VARIABLES

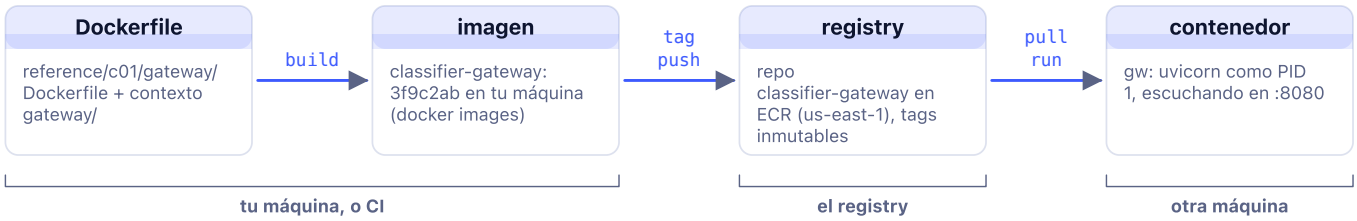
VARIABLE	QUÉ ES	EJEMPLO
plataforma	sistema y arquitectura de la máquina donde correrá	`linux/amd64 (nodos g4dn, c6i)`
repo	nombre del repositorio, en minúsculas	`classifier-gateway`
tag	versión legible; en el C01, el commit	`3f9c2ab (git rev-parse --short=7 HEAD)`
contexto	directorio con el Dockerfile y lo que este copia	`gateway/ (desde reference/c01)`
registry	host del registry	`123456789012.dkr.ecr.us-east-1.amazonaws.com`
digest	los 64 caracteres hex del digest que devolvió el push	`9b1c… (recortado)`
nombre	nombre del contenedor	`gw`
puerto_host	puerto en tu máquina	`8080`
puerto_contenedor	puerto donde escucha el proceso (el del EXPOSE)	`8080`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`Dockerfile`	La receta: instrucciones que el builder ejecuta en orden.	`reference/c01/gateway/Dockerfile`
`contexto`	Archivos que el builder puede leer: el último argumento de docker build, menos .dockerignore.	`gateway/`
`docker build -t`	Construye la imagen y le pone nombre:tag.	`docker build -t classifier-gateway:3f9c2ab gateway/`
`--platform`	Arquitectura de destino; en Apple Silicon, sin este flag sale arm64.	`--platform linux/amd64`
`imagen`	Capas de solo lectura + config (CMD, ENV, USER); se identifica por su digest.	`classifier-gateway:3f9c2ab`
`docker tag`	Agrega un nombre a una imagen que ya existe; no copia nada.	`docker tag classifier-gateway:3f9c2ab $REG/classifier-gateway:3f9c2ab`
`docker login`	Autentica contra el registry y guarda la credencial (Docker Desktop usa el llavero de macOS).	`aws ecr get-login-password … | docker login …`
`docker push`	Sube las capas que el registry no tiene y publica el tag.	`Layer already exists = esa capa ya estaba`
`registry`	Almacén de imágenes: repositorios, tags y manifests por digest.	`ECR en el C01; registry:3 en el lab`
`docker pull`	Baja las capas que faltan, por tag o por digest.	`docker pull $REG/classifier-gateway@sha256:9b1c…`
`docker run`	Crea y arranca un contenedor; si falta la imagen, la baja primero.	`Unable to find image '…' locally`
`contenedor`	Proceso + capa escribible sobre la imagen; lo que escribe se pierde al borrarlo.	`gw`

PIEZA	QUÉ HACE	EJEMPLO
`docker buildx build --push`	Build y push en un solo paso: lo que hace CI.	`docker buildx build --platform linux/amd64 -t $REG/classifier-gateway:$TAG --push gateway/`
`docker ps / docker logs`	¿Corre, con qué salud, y qué imprime?	`docker logs -f gw`

4 cosas unidas por 5 comandos



BUILD `docker build --platform linux/amd64 -t classifier-gateway:3f9c2ab gateway/`

TAG

`docker tag classifier-gateway:3f9c2ab 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway:3f9c2ab`

PUSH `docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway:3f9c2ab`

imprime `3f9c2ab: digest: sha256:… size: …`

PULL `docker pull 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway@sha256:9b1c…`

RUN

`docker run -d --name gw -p 8080:8080 123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-gateway@sha256:9b1c…`

- Antes del push va docker login contra ese registry (sección 11).
- En CI, build y push son un solo paso: docker buildx build --platform linux/amd64 --push.

Cuatro cosas (Dockerfile, imagen, registry, contenedor) unidas por cinco comandos. El nombre de la imagen decide a qué registry va el push.

EJEMPLO REAL · LA CADENA COMPLETA DEL GATEWAY DEL C01

BASH · ZSH **reference/c01 (gateway/ y k8s/gateway.yaml) y el ciclo diario de la hoja de comandos**

```
# Desde reference/c01. El tag es el commit corto: sabes exactamente qué código corre
TAG=$(git rev-parse --short=7 HEAD)
REG=123456789012.dkr.ecr.us-east-1.amazonaws.com

# 1) Dockerfile + contexto → imagen, para los nodos x86 (no para tu Mac)
docker build --platform linux/amd64 -t classifier-gateway:$TAG gateway/

# 2) Segundo nombre con el registry delante: mismo IMAGE ID, nada se copia
docker tag classifier-gateway:$TAG $REG/classifier-gateway:$TAG

# 3) Login y push: la última línea trae el digest (3f9c2ab: digest: sha256:… size: …)
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin $REG
docker push $REG/classifier-gateway:$TAG

# 4) En otra máquina: pull por digest y run (run también hace el pull si falta)
docker pull $REG/classifier-gateway@sha256:9b1c…
docker run -d --name gw -p 8080:8080 -e MODEL_NAME=classifier $REG/classifier-gateway@sha256:9b1c…

# 5) El contenedor: ¿corre y está sano?
docker ps --filter name=gw
curl -i localhost:8080/healthz
```

- El tag es el commit (`git rev-parse --short=7 HEAD`): de un contenedor llegas al código exacto. Es el mismo `classifier-gateway:3f9c2ab` de `k8s/gateway.yaml` .
- `--platform linux/amd64` : sin él, tu Mac construye arm64 y el nodo x86 no la puede ejecutar.
- `docker tag` no duplica nada: `docker images` muestra el mismo ID con dos nombres.
- El push imprime el digest; desde ahí, pull y run van por `@sha256:` (sección 11). `9b1c…` está recortado: el real tiene 64 caracteres.

- En tu Mac, correr la imagen amd64 imprime `WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8)` : corre emulada; sirve para probar, no para medir latencia.

CÓMO RECORDARLA

Cuatro cosas, cinco verbos: Dockerfile → `build` → imagen → `tag` y `push` → registry → `pull` y `run` → contenedor.

TRAMPA

Hacer `docker push classifier-gateway:3f9c2ab` sin el host delante: sin registry en el nombre, Docker apunta a Docker Hub (`docker.io/library/classifier-gateway`), no a tu ECR.

En el curso: M5 · Registries y tags inmutables · Docker y kubectl · Docker: construir, correr, depurar · C01 · Archivos de referencia

1 Anatomía de un Dockerfile

¿Qué escribes, en qué orden y por qué?

SINTAXIS CON VARIABLES

FROM *imagen* **:** *tag* **AS** *etapa*

Cada `FROM` abre una etapa limpia; `AS` le da nombre para `COPY --from`.

COPY --from= *etapa* *origen* *destino*

Sin `--from` copia del contexto, y los archivos quedan de root; con `--from`, BuildKit conserva el dueño que tenían en esa etapa. `--chown` fija otro.

ENV *VAR* **=** *valor*

Persiste en la imagen y en cada contenedor. `ARG`, en cambio, vive solo durante el build.

CMD [" *ejecutable* **", "** *arg* **", "…"]**

Forma exec: lista JSON con comillas dobles, y tu proceso es el PID 1. ENTRYPOINT y CMD en detalle: sección 5.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
imagen	imagen base	`python`
tag	versión de la base	`3.13-slim`
etapa	nombre de la etapa	`builder, runtime`
origen	ruta en la otra etapa (o en el contexto)	`/opt/venv`
destino	ruta en la imagen que estás armando	`/opt/venv`
VAR	variable de entorno	`PYTHONUNBUFFERED`
valor	su valor	`1`
ejecutable	programa que será el PID 1	`uvicorn`
arg	argumentos, uno por elemento de la lista	`app.main:app, --port, 8080`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`# syntax=docker/dockerfile:1`	Primera línea: usa la versión estable más reciente del frontend de Dockerfile 1.x.	`línea 1 del gateway`
FROM	Imagen base y nombre de etapa; cada FROM empieza una etapa nueva y limpia.	`FROM python:3.13-slim AS runtime`
WORKDIR	Directorio de trabajo para lo que sigue (RUN, COPY, CMD); lo crea si no existe.	`WORKDIR /srv`
COPY	Copia desde el contexto (quedan de root) o desde otra etapa (conservan su dueño); --chown fija otro.	`COPY --from=builder /opt/venv /opt/venv`
ADD	Como COPY, pero además baja URLs, clona repos git y descomprime tar locales. Para archivos locales, COPY.	`ADD --checksum=sha256:… https://…/x.tar.gz /opt/`
RUN	Ejecuta un comando durante el build y guarda lo que cambió como una capa.	`RUN python -m venv /opt/venv`
ENV	Variable de entorno para el build y para cada contenedor; queda en la imagen.	`ENV PYTHONUNBUFFERED=1`
ARG	Variable solo de build (--build-arg); su valor queda en docker history (fila del ARG y RUN siguientes).	`ARG GIT_SHA`
EXPOSE	Documenta el puerto donde escucha el proceso; no publica nada (eso es -p en docker run).	`EXPOSE 8080`
USER	Usuario de los RUN siguientes y del proceso; con UID numérico, Kubernetes puede verificar runAsNonRoot.	`USER 10001`
HEALTHCHECK	Comando que Docker corre cada --interval; exit 0 = healthy, 1 = unhealthy. Kubernetes lo ignora.	`HEALTHCHECK --interval=10s --timeout=2s … CMD […]`
CMD	Comando por defecto del contenedor; en forma exec, tu proceso es el PID 1.	`CMD ["uvicorn", "app.main:app", …]`
ENTRYPOINT	Ejecutable fijo; CMD pasa a ser sus argumentos (detalle en la sección 5).	`las imágenes de NGC: /opt/nvidia/nvidia_entrypoint.sh`
LABEL	Metadatos clave=valor (repo, commit); no agrega contenido a la imagen.	`LABEL org.opencontainers.image.revision="3f9c2ab"`



INSTRUCCIÓN	¿TOCA EL FILESYSTEM?	¿CUÁNDO ACTÚA?	¿QUÉ QUEDA EN LA IMAGEN?
FROM	Pone las capas de la base	Build	Las capas de la base
RUN	Sí: una capa con lo que cambió	Build	El resultado; el comando, en docker history
COPY / ADD	Sí: una capa con los archivos	Build	Los archivos copiados
WORKDIR	Solo crea el directorio si falta	Build y run	El directorio de trabajo
ENV	No	Build y run	La variable (la ves con docker inspect)
ARG	No	Solo build	Nada en la config; sí en docker history
USER	No	Build (los RUN siguientes) y run	El usuario del proceso
EXPOSE	No	Ninguno: documenta	El puerto, como metadato
HEALTHCHECK	No	Run (docker y compose; Kubernetes lo ignora)	El comando de salud
CMD / ENTRYPOINT	No	Run	El comando de arranque
LABEL	No	Ninguno	Metadatos clave=valor

docker image inspect muestra la config (ENV, USER, ExposedPorts, Cmd, Healthcheck); docker history, una fila por instrucción, también las de la base.

Qué hace cada instrucción y cuándo actúa. El contenido de la imagen sale de FROM, RUN, COPY y ADD; el resto es configuración.

EJEMPLO REAL · EL DOCKERFILE DEL GATEWAY DE C01

DOCKERFILE **reference/c01/gateway/Dockerfile (todas las instrucciones; sin los comentarios)**

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim AS builder
ENV PIP_NO_CACHE_DIR=1 PIP_DISABLE_PIP_VERSION_CHECK=1
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
COPY requirements.txt .
RUN pip install --only-binary=:all: -r requirements.txt

FROM python:3.13-slim AS runtime
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PATH="/opt/venv/bin:$PATH"
RUN groupadd --system --gid 10001 app && \
    useradd --system --uid 10001 --gid app --no-create-home --shell /usr/sbin/nologin app
WORKDIR /srv
COPY --from=builder /opt/venv /opt/venv
COPY app/ ./app/
USER 10001
EXPOSE 8080
HEALTHCHECK --interval=10s --timeout=2s --start-period=10s --retries=3 \
    CMD ["python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8080/healthz', timeout=1)"]
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080", "--workers", "1", \
     "--timeout-keep-alive", "75", "--no-access-log", "--timeout-graceful-shutdown", "20"]
```

- Dos `FROM`, dos etapas: `builder` arma el venv; `runtime` arranca de una slim limpia y solo recibe `/opt/venv` por `COPY --from=builder`.
- `ENV PATH="/opt/venv/bin:$PATH"` en las dos etapas: así se encuentra el `uvicorn` del venv; sin eso, el contenedor sale con 127.
- `COPY requirements.txt .` y `pip install` van antes de `COPY app/`: editar el código no reinstala las 37 dependencias (sección 2).
- `groupadd` y `useradd` crean el UID 10001 y `USER 10001` lo activa; `COPY app/` sin `--chown` deja el código de root: el proceso lo lee pero no lo modifica.
- `EXPOSE 8080` solo documenta. El `HEALTHCHECK` usa Python porque la slim no trae curl, y Kubernetes lo ignora (allá mandan las probes).
- `CMD` en forma exec con `--workers 1`: uvicorn es el PID 1 y recibe el SIGTERM (sección 5).

CÓMO RECORDARLA

Léelo en dos tiempos: lo que corre en **build** (FROM, RUN, COPY, ARG) arma el filesystem; lo que vale en **run** (ENV, USER, WORKDIR, HEALTHCHECK, CMD) es la configuración con la que arranca el contenedor.

TRAMPA

Creer que `EXPOSE 8080` abre el puerto: sin `-p 8080:8080` nada llega desde tu Mac. Y comentar al final de una instrucción (`FROM python:3.13-slim # base`): en un Dockerfile, `#` solo es comentario al inicio de la línea.

En el curso: M5 · Un Dockerfile de ML, decisión por decisión · C01 · Archivos de referencia

Docker y kubectl · Docker: construir, correr, depurar

2 Capas, caché y .dockerignore

¿Por qué cambiar una línea a veces reconstruye todo, y cómo lo evitas?

SINTAXIS CON VARIABLES

`COPY` *archivo_deps* **`.`**

Primero solo el archivo de dependencias: esta capa cambia cuando cambian las dependencias, no cuando cambia el código.

`RUN --mount=type=cache,target=` *dir_cache* **`pip install -r`** *archivo_deps*

Cache mount: si la capa se rehace, pip reusa lo que ya bajó. No lo combines con `PIP_NO_CACHE_DIR=1`, que apaga ese cache.

`COPY` *código* *destino*

El código al final: cambia en cada commit y solo invalida lo que viene después.

`docker build --pull --no-cache -t` *imagen*`:`*tag* *contexto*

Build desde cero: `--pull` trae la base más reciente y `--no-cache` rehace cada paso.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
archivo_deps	lockfile o lista de dependencias	`requirements.txt`
dir_cache	directorio de cache del gestor de paquetes	`/root/.cache/pip`
código	lo que cambia en cada commit	`app/`
destino	dónde queda en la imagen	`./app/ (dentro de WORKDIR /srv)`
imagen	nombre de la imagen	`classifier-gateway`
tag	versión	`3f9c2ab`
contexto	directorio que recibe el builder	`gateway/`
cambió_i	la instrucción i cambió su texto, sus archivos (COPY/ADD) o un ARG que usa	`editar main.py cambia COPY app/`
rebuild_i	la capa i se vuelve a construir	`todo lo que está debajo de un COPY . . invalidado`
k	primera instrucción que cambió	`COPY app/ (si solo tocaste código)`
$s_i,\ t_i$	tamaño y tiempo de build de la capa i	`app/ ≈ 48 KB; venv ≈ 44 MB comprimido`
B_{deploy}	bytes que baja un nodo que ya tenía la versión anterior	`≈ 48 KB con el orden del C01`
T_{build}	tiempo del build	`≈ 2–5 s si solo cambia app/ (estimación del módulo 5 del curso)`

FÓRMULAS

$$\text{rebuild}_i = \text{cambió}_i \vee \text{rebuild}_{i-1} \qquad \text{rebuild}_0 = \text{cambió la base}$$

$$k = \min\{i : \text{cambió}_i\} \Rightarrow \text{se rehacen } k, k{+}1, \ \ldots\ , n$$

$$B_{\text{deploy}} \approx \sum_{i \geq k} s_i \qquad T_{\text{build}} \approx \sum_{i \geq k} t_i$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
capa	Diff de archivos que deja un RUN, COPY o ADD; se comparte entre imágenes por digest.	`venv del gateway: ≈ 44 MB comprimido`
CACHED	BuildKit reutilizó el paso: misma instrucción, mismos archivos, mismas capas anteriores.	`#8 CACHED con --progress=plain`
RUN (caché)	Compara el texto del comando, no lo que descarga: un RUN igual no se vuelve a ejecutar solo.	`RUN apt-get update se queda viejo`
COPY / ADD (caché)	Comparan un checksum de los archivos copiados; el mtime no cuenta.	`editar main.py invalida COPY app/`

PIEZA	QUÉ HACE	EJEMPLO
ARG (caché)	Un valor nuevo invalida desde su primer uso, y los RUN posteriores lo reciben como variable.	`ARG GIT_SHA arriba = reinstalar en cada commit`
contexto	Los archivos que el builder puede leer: el último argumento de docker build.	`gateway/ en docker build … gateway/`
.dockerignore	Patrones fuera del contexto, relativos a su raíz; ** para cualquier profundidad, ! para excepciones.	`**/__pycache__/, .env, **/*.plan`
Dockerfile.dockerignore	Ignore propio de un Dockerfile, a su lado; tiene prioridad sobre el de la raíz.	`api/Dockerfile.dockerignore para docker build -f api/Dockerfile .`
RUN --mount=type=cache	Directorio que persiste entre builds para el cache del gestor de paquetes; no entra en la imagen.	`target=/root/.cache/pip`
--no-cache	Rehace todos los pasos, pero sobre la base que ya tienes.	`docker build --no-cache …`
--no-cache-filter	Ignora la caché solo en las etapas que nombras.	`--no-cache-filter builder`
--pull	Vuelve a resolver y bajar las imágenes base aunque ya las tengas.	`trae el último parche de python:3.13-slim`
docker builder prune	Borra la caché de build colgante (dangling); con -a, toda la que no se usa. Libera disco en la VM.	`cuando docker system df muestra GB de caché`

i	Instrucción (en el orden del Dockerfile)	Tamaño comprimido (a escala)	En el build
0	`FROM python:3.13-slim`	43 MB	CACHED
1	`ENV PIP_NO_CACHE_DIR=1 PYTHONUNBUFFERED=1`	0 MB	CACHED
2	`WORKDIR /srv`	0 MB	CACHED
3	`COPY . .`	0.05 MB	cambió
4	`RUN pip install --only-binary=:all: -r requirements.txt`	44 MB	se rehace
5	`USER 10001`	0 MB	se rehace
6	`CMD ["uvicorn", "app.main:app", …]`	0 MB	se rehace

Desde k = 3: se rehacen 4 de 7 pasos y vuelven a viajar $\Sigma\, s_i$ ($i \geq 3$) ≈ 44 MB en cada deploy.

El gateway con COPY . . arriba (antipatrón). Editas app/main.py: COPY . . cambia su checksum y se rehace todo lo de abajo, incluidos ≈ 44 MB comprimidos de dependencias que no cambiaron. Con el orden del C01 (requirements.txt antes que app/) solo se rehace la capa del código, ≈ 48 KB. Tamaños comprimidos aproximados, del módulo 5 del curso.

EJEMPLO REAL · EL .DOCKERIGNORE DEL GATEWAY DEL C01

TEXTO reference/c01/gateway/.dockerignore

```
# Lo que NO entra al contexto de build del gateway: builds más rápidos y sin filtrar basura a la imagen
# Tests y su config: no corren en producción (se corren en CI antes del build)
tests/
pytest.ini
requirements-dev.txt
# Artefactos locales de Python en cualquier carpeta: se regeneran, y un venv de macOS no sirve en Linux
**/__pycache__/
**/*.pyc
**/.pytest_cache/
.venv/
venv/
# Control de versiones y editores
.git/
.gitignore
.vscode/
.idea/
# Secretos locales: jamás dentro de una imagen (se inyectan como env/Secret en runtime)
.env
*.pem
# Imágenes de prueba y modelos en cualquier carpeta (**/): el gateway no los necesita (el modelo vive en Triton)
```

```
**/*.jpg
**/*.jpeg
**/*.png
**/*.onnx
**/*.plan
```

- Los patrones son relativos a la raíz del contexto (`gateway/`) y `**/` los aplica a cualquier profundidad. A diferencia de `.gitignore`, un `*.pyc` sin `**/` solo cubriría la raíz.
- `**/__pycache__/` y `**/*.pyc` : correr pytest en tu Mac crea `app/__pycache__` ; sin estas líneas, `COPY app/` los copiaría (basura en la imagen y otro checksum para esa capa).
- `tests/` , `pytest.ini` y `requirements-dev.txt` : los tests corren en CI antes del build; la imagen no los necesita.
- `.env` y `*.pem` : secretos locales fuera del contexto; con un `COPY . .` terminarían en una capa.
- `**/*.onnx` , `**/*.plan` e imágenes: el modelo vive en Triton, y un checkpoint en el contexto alarga los builds (lo ves en `transferring context`).

CÓMO RECORDARLA

Lo que cambia poco, arriba; lo que cambia en cada commit, abajo. La caché se corta en la primera instrucción que cambió y desde ahí se rehace todo.

TRAMPA

`ARG GIT_SHA` (o `COPY . .`) al principio del Dockerfile: el valor cambia en cada commit y, como los `RUN` que siguen lo reciben como variable de entorno, pueden perder la caché y reinstalar todo. Declara ese ARG al final, junto al LABEL que lo usa. Y `--no-cache` no trae una base nueva: eso es `--pull` .

En el curso: M5 · Imagen, contenedor y capas · M5 · Un Dockerfile de ML, decisión por decisión · Docker y kubectl · Docker: construir, correr, depurar

3 Imágenes chicas y seguras

¿Cómo dejas en la imagen final solo lo que el proceso necesita para correr, y nada que no deba estar?

SINTAXIS CON VARIABLES

`FROM` *imagen* `:` *tag* `@sha256:` *digest* `AS builder`

Base fijada por contenido; el tag queda para humanos y para Renovate o Dependabot. Fija el digest del índice multi-plataforma, no el de amd64.

`COPY --from=builder` *artefacto* *artefacto*

Lo único que cruza de etapa: el venv, un binario, unas wheels.

`RUN apt-get update && apt-get install -y --no-install-recommends` *paquetes* `&& rm -rf /var/lib/apt/lists/*`

Todo en un solo RUN: lo que borras en otro RUN sigue en la capa anterior.

`RUN --mount=type=secret,id=` *id* `,env=` *VAR* *comando*

El secreto existe solo durante ese RUN; se pasa con `docker build --secret id=` *id* `,env=` *VAR* `.`

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
imagen	imagen base	`python`
tag	versión legible	`3.13-slim`
digest	hash del índice multi-plataforma de esa base	`lo imprime docker buildx imagetools inspect python:3.13-slim`
artefacto	lo que la etapa final necesita para correr	`/opt/venv`
paquetes	paquetes del sistema que el proceso usa en runtime	`ffmpeg (forwarder del C02)`
id	nombre del secreto en el build	`hf_token`
VAR	variable de entorno que lo expone en ese RUN	`HF_TOKEN`
comando	lo que necesita el secreto	`python download_weights.py`
S_{final}, $S_{\text{base run}}$	tamaño de la imagen final y de su base	`≈ 43 MB de python:3.13-slim + el venv`
s_i, s_k	tamaño de cada capa	`el venv del gateway, ≈ 44 MB comprimido`

FÓRMULAS

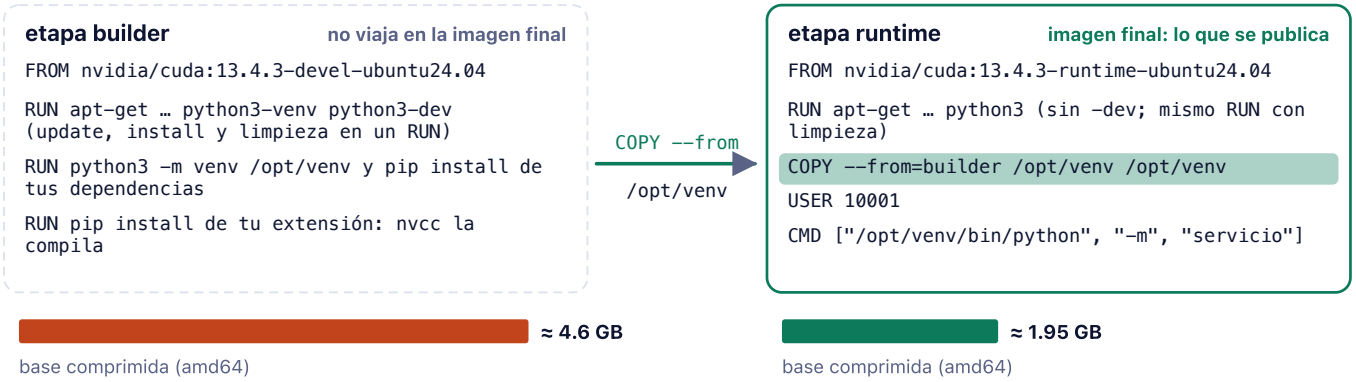
$$S_{\text{final}} = S_{\text{base run}} + \sum_{k \,\in\, \text{etapa final}} s_k \qquad (S_{\text{builder}} \text{ no aparece})$$

$$S = \sum_i s_i, \quad s_i \geq 0 \Rightarrow \text{borrar en una capa posterior no resta}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`multi-stage`	Varias FROM; la final copia solo artefactos con COPY --from y lo demás se descarta.	`builder → runtime en el gateway del C01`
`--target`	Construye solo hasta una etapa: para depurar o correr tests.	`docker build --target builder -t gw:builder gateway/`
`python:3.12 (full)`	Debian + toolchain (buildpack-deps): gcc y headers; ≈ 410 MB comprimida.	`etapa builder si algo se compila`
`python:3.12-slim`	Debian mínimo + Python; ≈ 43 MB comprimida.	`la final del gateway (3.13-slim)`
`python:3.12-alpine`	musl en vez de glibc: pip no puede usar las wheels manylinux.	`torch 2.14.0 no tiene wheel musllinux`
`-runtime vs -devel`	devel suma nvcc, headers y librerías estáticas: ≈ 2.5–2.7 GB más, comprimida.	`nvidia/cuda:13.4.3-devel… solo en builder`
`--no-cache-dir / PIP_NO_CACHE_DIR=1`	pip no guarda en la capa una copia de lo que bajó.	`ENV PIP_NO_CACHE_DIR=1 (gateway)`
`--only-binary=:all:`	Solo wheels: si falta una, el build falla en vez de compilar sin avisar.	`pip install --only-binary=:all: -r requirements.txt`
`--no-install-recommends`	apt instala solo las dependencias obligatorias, no las recomendadas.	`apt-get install -y --no-install-recommends ffmpeg`

PIEZA	QUÉ HACE	EJEMPLO
`rm -rf /var/lib/apt/lists/*`	Borra los índices de apt; solo resta si va en el mismo RUN que el install.	última línea del RUN de apt del C02
`USER (UID numérico)`	Proceso sin root; con un número, Kubernetes puede verificar runAsNonRoot.	`USER 10001`
`@sha256: en FROM`	Fija la base por contenido; el tag se republica con cada parche.	`python:3.13-slim@sha256:…`
`RUN --mount=type=secret`	Monta el secreto en /run/secrets/ID (o como env) solo durante ese RUN; no queda en capas ni historial.	`docker build --secret id=hf_token,env=HF_TOKEN .`



Cruza a la etapa final: /opt/venv (dependencias y la extensión ya compiladas)

El caso donde multi-stage paga de verdad: CUDA. Las bases 13.4.3 pesan ≈ 4.6 GB (devel) y ≈ 1.95 GB (runtime), comprimidas para amd64 en Docker Hub; la final no paga nvcc ni headers en cada pull. Patrón ilustrativo, no construido aquí; es el mismo builder → runtime con COPY --from del gateway del C01.

EJEMPLO REAL · EL FORWARDER DEL C02: MULTI-STAGE, APT LIMPIO Y UID FIJO

DOCKERFILE — reference/c02/edge/forwarder/Dockerfile (todas las instrucciones; sin los comentarios)

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim AS builder
ENV PIP_NO_CACHE_DIR=1 PIP_DISABLE_PIP_VERSION_CHECK=1
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
COPY requirements.txt .
RUN pip install --only-binary=:all: -r requirements.txt

FROM python:3.13-slim AS runtime
RUN apt-get update && \
    apt-get install -y --no-install-recommends ffmpeg && \
    rm -rf /var/lib/apt/lists/*
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PATH="/opt/venv/bin:$PATH" \
    DATA_DIR=/var/lib/forwarder
RUN groupadd --system --gid 10001 app && \
    useradd --system --uid 10001 --gid app --no-create-home --shell /usr/sbin/nologin app && \
    mkdir -p /var/lib/forwarder && chown app:app /var/lib/forwarder
WORKDIR /srv
COPY --from=builder /opt/venv /opt/venv
COPY app.py rules.py ./
USER 10001
CMD ["python", "app.py"]
```

- Mismo esqueleto que el gateway del C01: `builder` arma `/opt/venv` con `--only-binary=:all:` y `PIP_NO_CACHE_DIR=1`; la final copia solo el venv y el código.
- `apt-get update`, `install --no-install-recommends ffmpeg` y `rm -rf /var/lib/apt/lists/*` en un solo `RUN`: si la limpieza fuera en otro RUN, las listas seguirían en la capa del install.
- ffmpeg va en la etapa final porque el proceso lo ejecuta en runtime; lo que solo sirve para construir se queda en `builder`.
- El UID 10001 es dueño solo de `/var/lib/forwarder` (la cola SQLite): el código sigue siendo de root y de solo lectura.
- Sin `HEALTHCHECK` a propósito: no hay puerto que consultar (el revisor lo marca como info, y está bien).

- Multi-stage no es obligatorio: el C03 usa una sola etapa porque no hay nada que compilar y lo pesado ya viene en la base.

CÓMO RECORDARLA

Construye gordo, corre flaco: compiladores, headers y caches se quedan en el builder; a la imagen final solo llega lo que copias con `COPY --from`.

TRAMPA

Limpiar en otro `RUN` (`RUN rm -rf /var/lib/apt/lists/*` después del install) no resta un byte: el archivo sigue en la capa de abajo. Y Alpine para ML: con musl, pip no puede usar las wheels manylinux, y torch no publica wheels musllinux (en Alpine, `pip install torch` termina en `No matching distribution found for torch`).

En el curso: M5 · Reducir tamaño (y por qué importa para el cold start) · M5 · Imágenes CUDA: runtime vs devel, drivers y compatibilidad · M5 · Un Dockerfile de ML, decisión por decisión · M15 · Secretos y credenciales

4 docker run y el ciclo de vida

¿Cómo arrancas un contenedor, lo miras por dentro y entiendes por qué se detuvo?

SINTAXIS CON VARIABLES

`docker run -d --name` *nombre* **`-p`** *puerto_host* **`:`** *puerto_cont* **`-e`** *VAR* **`=`** *valor* *imagen* **`:`** *tag*

Las flags van **antes** de la imagen; lo que escribas después de la imagen reemplaza al `CMD`.

`docker run --rm -it --entrypoint` *programa* *imagen* **`:`** *tag* *args*

`--entrypoint` cambia el programa y descarta el `CMD` de la imagen; lo que va después de la imagen son sus argumentos.

`docker run -d --name` *nombre* **`--restart`** *política* **`--memory`** *mem* **`--cpus`** *cpus* *imagen* **`:`** *tag*

Qué hace Docker cuando el proceso termina y cuánto puede usar: lo que en Kubernetes hacen el kubelet y `resources.limits`.

`docker ps -a --filter name=`*nombre*
`docker logs -f` *nombre*
`docker exec -it` *nombre* *programa*
`docker stop` *nombre* **`&& docker rm`** *nombre*

Después todo va por nombre: ver el estado (incluidos los que terminaron), seguir los logs, entrar, parar y borrar.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre del contenedor; único en el host, aunque esté parado	`gw`
puerto_host	puerto de tu Mac	`8080`
puerto_cont	puerto donde escucha el proceso dentro del contenedor	`8080`
VAR	nombre de una variable de entorno	`MODEL_NAME`
valor	su valor	`classifier`
imagen	repositorio de la imagen	`classifier-gateway`
tag	versión de la imagen	`dev (en CI, el commit: 3f9c2ab)`
programa	ejecutable que corre en lugar del ENTRYPOINT, o el que lanzas con exec	`sh (para entrar a una imagen de NGC sin su script de entrada)`
args	argumentos del programa	`-c 'ls /models'`
política	no (default), on-failure[:N], always o unless-stopped	`unless-stopped (como el gateway en Compose)`
mem	tope de RAM, con sufijo b, k, m o g; mínimo 6m	`512m`
cpus	tope de CPU en cores; acepta fracciones	`1.5`
d_k	espera antes del reinicio número k; vuelve a 0.1 s si el contenedor alcanzó a correr 10 s	`0.1 s, 0.2 s, 0.4 s… hasta 60 s`
k	número de reinicio seguido	`1, 2, 3`
m, m_{total}	valor de --memory y lo que el contenedor puede usar entre RAM y swap si no fijas --memory-swap (la VM de Docker Desktop trae 1 GB de swap)	`32m → 64m en total`

FÓRMULAS

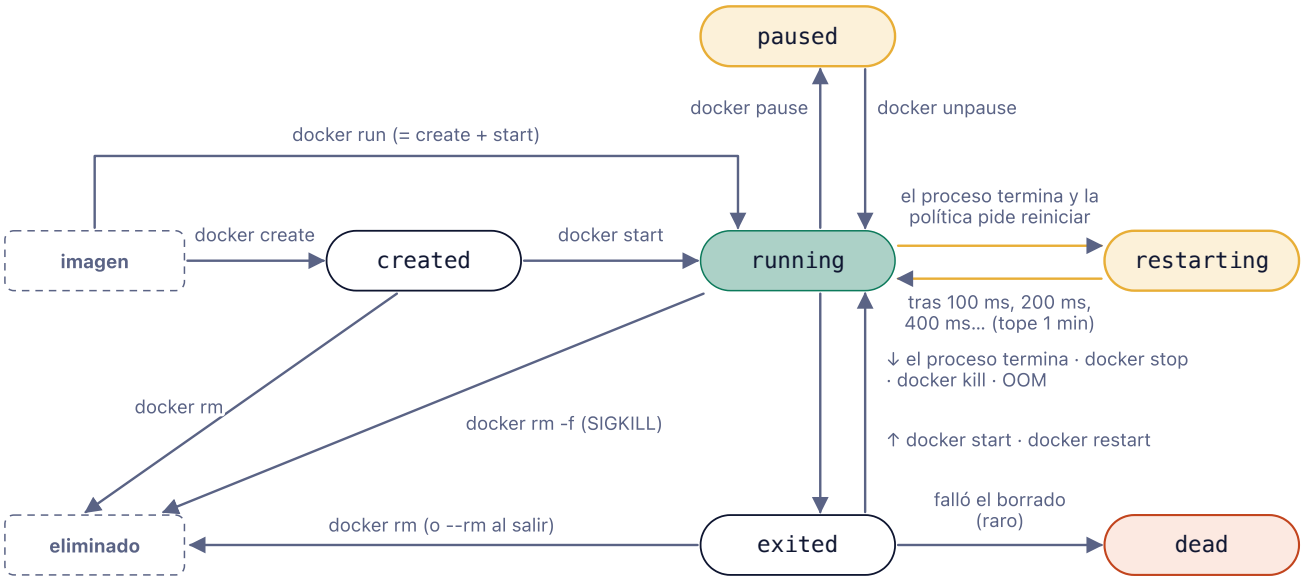
$$d_k = \min\left(0.1\text{s} \cdot 2^{k-1}, 60\text{s}\right)$$

$$m_{\text{total}} = 2m$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
-d · -it	-d: segundo plano e imprime el ID. -it: stdin abierto y una terminal, para shells y REPLs.	`docker run -it --rm python:3.12-slim`
--rm · --name	--rm borra el contenedor y sus volúmenes anónimos al salir; --name le da un nombre para los demás comandos.	`--rm --name gw`
-p *puerto_host* **:** *puerto_cont*	Publica un puerto: el host escucha y reenvía al contenedor. Primero va el host (sección 6).	`-p 8080:8080`
-e · --env-file	Variables de entorno: una por flag, o un archivo con una línea VAR=valor por variable.	`-e TRITON_URL=triton:8000`
-v · --mount	Monta una carpeta del host o un volumen; --mount también un tmpfs (sección 7).	`-v ./config:/etc/app:ro`
--restart	Qué hace Docker cuando el proceso termina: no, on-failure[:N], always o unless-stopped. Incompatible con --rm.	`--restart unless-stopped`
--memory · --cpus	Topes de RAM (pasar RAM + swap: OOM kill, exit 137) y de CPU (pasarlo lo frena, no lo mata).	`--memory 512m --cpus 1`
--entrypoint · -w · --user	Cambian el programa, el directorio de trabajo (lo crea si falta) y el UID:GID del proceso.	`--entrypoint sh -w /srv --user 10001:10001`
docker ps -a	Lista también los contenedores que terminaron; STATUS trae el exit code.	`Exited (137) 5 seconds ago`
docker logs -f	stdout y stderr del contenedor, aunque ya haya terminado; -f sigue, -n da las últimas líneas.	`docker logs -f -n 50 gw`
docker exec -it	Proceso nuevo dentro de un contenedor que corre; no es el PID 1 ni reinicia nada.	`docker exec -it gw sh`
docker inspect -f	JSON completo del contenedor; -f extrae campos con una plantilla Go.	`docker inspect -f '{{.State.ExitCode}} {{.State.OOMKilled}}' gw`
docker stats	CPU, memoria contra su tope, red y disco en vivo; --no-stream da una sola lectura.	`docker stats --no-stream gw`
docker stop · kill · rm -f	stop: SIGTERM, espera 10 s y SIGKILL. kill: SIGKILL inmediato. rm -f: borra aunque corra, con SIGKILL.	`docker stop -t 30 gw`

Recuadros punteados: no son estados del contenedor (la imagen de partida y el contenedor ya borrado).



Estados de un contenedor y los comandos que lo mueven. restarting solo existe con una política de reinicio; docker ps sin -a esconde los created, exited y dead.

POLÍTICAS DE REINICIO (--RESTART EN DOCKER RUN, RESTART: EN COMPOSE)

POLÍTICA	REINICIA CUANDO…	DESPUÉS DE DOCKER STOP	AL REINICIAR DOCKER (O DOCKER DESKTOP)
no (default)	nunca	queda parado	no arranca
on-failure[:N]	el exit code no es 0, hasta N veces	queda parado	no arranca
always	termina por cualquier motivo	queda parado	arranca otra vez, aunque lo hayas parado tú
unless-stopped	termina por cualquier motivo	queda parado	arranca solo si estaba corriendo

Las esperas entre reinicios crecen (100 ms, 200 ms, 400 ms… hasta 1 min). Ninguna política mira el HEALTHCHECK: solo el exit code.

EJEMPLO REAL · EL GATEWAY DE C01 CON DOCKER RUN, SIN TRITON

BASH · ZSH — reference/c01/gateway/Dockerfile y el servicio gateway de reference/c01/docker-compose.yml, escritos como docker run

```
# Desde reference/c01. El gateway solo no necesita GPU (en tu Mac corre emulado como amd64)
docker build --platform linux/amd64 -t classifier-gateway:dev gateway/

# Los valores del servicio gateway de docker-compose.yml, como flags
docker run -d --name gw -p 8080:8080 \
  -e MODEL_NAME=classifier -e INFER_TIMEOUT_S=0.15 \
  --restart unless-stopped --memory 512m --cpus 1 \
  classifier-gateway:dev

docker ps --filter name=gw      # STATUS: Up … (health: starting) y, a los segundos, (healthy)
curl -i localhost:8080/healthz   # 200: el proceso vive
curl -i localhost:8080/readyz    # 503: sin Triton no está listo
docker inspect -f '{{.State.Health.Status}} user={{.Config.User}} cwd={{.Config.WorkingDir}}' gw
docker logs -f -n 20 gw          # Ctrl+C deja de seguir; el contenedor sigue
docker stop -t 30 gw && docker inspect -f 'exit={{.State.ExitCode}}' gw
docker rm gw
```

- Cada `-e` sale del bloque `environment:` del servicio `gateway`, y `--restart unless-stopped` de su línea `restart:`. `TRITON_URL` queda fuera a propósito: sin él, el gateway apunta a `localhost:8000` (sección 6).
- `(healthy)` en `docker ps` sale del `HEALTHCHECK` del Dockerfile, que le pega a `/healthz` con Python porque la imagen slim no trae curl.
- `/healthz` en 200 y `/readyz` en 503 a la vez es la degradación diseñada de C01: el proceso vive, pero no puede inferir.
- `inspect` confirma lo que fijó el Dockerfile: `healthy user=10001 cwd=/srv` (`USER 10001`, `WORKDIR /srv`).
- `docker stop -t 30` da `exit=0` porque el `CMD` está en forma exec: uvicorn es el PID 1, recibe SIGTERM y drena. Los 30 s son los que el gateway tiene en k8s después del SIGTERM (sección 5).

CÓMO RECORDARLA

run = create + start, con las flags antes de la imagen. Ante un contenedor caído, siempre `ps -a` → `logs` → `inspect` (estado, exit code, `OOMKilled`, `RestartCount`).

TRAMPA

`docker ps` sin `-a` esconde los contenedores que terminaron; el que "desapareció" sigue ahí con su exit code y sus logs. Y una política de reinicio no arregla un bug: con `--restart always`, un error al arrancar se vuelve un bucle de reinicios cada vez más espaciados (hasta 1 min) que lo disimula. Mira `RestartCount` y los logs antes de subir límites.

En el curso: Docker y kubectl · Docker: construir, correr, depurar | Docker y kubectl · Depurar pods: síntoma → comandos → causa típica | M5 · docker compose para desarrollo local | C01 · Archivos de referencia

5 ENTRYPOINT vs CMD, PID 1 y señales

¿Quién es el PID 1 de tu contenedor y qué hace cuando llega SIGTERM?

SINTAXIS CON VARIABLES

`CMD ["` *ejecutable* **`", "`** *arg* **`"]`**

Forma exec: lista JSON con comillas dobles. Docker ejecuta el programa sin shell y ese proceso es el PID 1.

`ENTRYPOINT ["` *ejecutable* **`"]`**
`CMD ["` *arg* **`"]`**

`ENTRYPOINT` fija el programa y `CMD` trae los argumentos por defecto: lo que pongas después de la imagen en `docker run` reemplaza solo al `CMD`.

```
#!/bin/sh
set -e
preparación
exec "$@"
```

Script de entrada (`ENTRYPOINT ["/entrypoint.sh"]`): prepara y, con `exec`, le cede el PID 1 al `CMD`.

`docker run --init --stop-timeout` *segundos* *imagen*

`--init` pone tini como PID 1 para reenviar señales; `--stop-timeout` cambia los 10 s antes del SIGKILL. En Compose: `init: true` y `stop_grace_period`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ejecutable	programa que queda como PID 1	`uvicorn`
arg	un argumento por elemento de la lista	`"app.main:app", "--host", "0.0.0.0"`
preparación	lo que el script hace antes de ceder el control	`generar un archivo de config desde variables de entorno`
segundos	plazo entre SIGTERM y SIGKILL	`30`
imagen	imagen con su tag	`lab-pid1:exec`
$n_{\text{señal}}$	número de la señal que terminó al proceso	`9 (SIGKILL) → 137; 15 (SIGTERM) → 143`
t_{gracia}	plazo de SIGTERM a SIGKILL (docker stop -t, stop_grace_period)	`10 s por defecto en docker stop y Compose; 30 s en el k8s del gateway (45 − 15 de preStop)`
T_{k8s}	terminationGracePeriodSeconds: plazo total en Kubernetes, preStop incluido	`45 s en k8s/gateway.yaml`
t_{preStop}	espera antes del SIGTERM (solo en Kubernetes); corre dentro del plazo total de Kubernetes	`15 s en el gateway de C01`
t_{drenado}	lo que el servidor tarda en terminar lo que tiene en curso	`hasta 20 s (--timeout-graceful-shutdown 20)`
margen	colchón para el resto del apagado	`10 s`

FÓRMULAS

$$\text{código de salida} = 128 + n_{\text{señal}}$$

$$t_{\text{drenado}} + \text{margen} \le t_{\text{gracia}} = T_{\text{k8s}} - t_{\text{preStop}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Forma exec	Lista JSON: Docker ejecuta el programa directo, sin shell; tampoco expande $VARIABLES.	`CMD ["uvicorn", "app.main:app"]`
Forma shell	Docker la envuelve en /bin/sh -c; en imágenes Debian, sh queda como PID 1.	`CMD uvicorn app.main:app`
ENTRYPOINT	El programa fijo de la imagen; los argumentos de docker run se le agregan al final.	`ENTRYPOINT ["/opt/nvidia/nvidia_entrypoint.sh"] (bases de NGC)`
CMD	Comando o argumentos por defecto; se reemplaza con lo que va después de la imagen.	`docker run classifier-triton:… tritonserver --help`

PIEZA	QUÉ HACE	EJEMPLO
PID 1	Primer proceso del contenedor: el kernel le descarta las señales sin handler (salvo SIGKILL) y, si muere, muere todo.	`docker top gw`
SIGTERM (15)	El aviso de docker stop y del kubelet: deja de aceptar trabajo, termina lo que tienes y sal.	`docker stop -t 30 gw`
SIGKILL (9)	Muerte inmediata; no se puede atrapar. Llega cuando vence el plazo.	`docker kill gw`
Plazo (grace period)	docker stop -t, --stop-timeout o stop_grace_period: 10 s por defecto; en Kubernetes, 30 s.	`terminationGracePeriodSeconds: 45 (gateway de C01)`
STOPSIGNAL · --stop-signal	Cambian la primera señal de docker stop (SIGTERM por defecto).	`STOPSIGNAL SIGINT`
--init · tini	PID 1 mínimo que reenvía señales y recoge zombis; devuelve el código de salida del hijo.	`docker run --init … · init: true`
exec "$@"	Última línea de un script de entrada: el shell se reemplaza por el CMD, que pasa a ser PID 1.	`el entrypoint de NGC hace exec del CMD`
JSONArgsRecommended	Aviso de BuildKit al construir un CMD o ENTRYPOINT en forma shell.	`...to prevent unintended behavior related to OS signals`
uvicorn --timeout-graceful-shutdown	Con SIGTERM deja de aceptar, espera hasta N s las requests en curso y corre el shutdown del lifespan.	`--timeout-graceful-shutdown 20`
gunicorn --graceful-timeout	TERM es apagado ordenado: espera a los workers hasta N s (30 por defecto); INT o QUIT, rápido.	`gunicorn app.main:app -k uvicorn_worker.UvicornWorker -b 0.0.0.0:8080 --graceful-timeout 20`



docker stop manda SIGTERM al PID 1, espera el plazo (10 s por defecto) y manda SIGKILL. Cada carril es un PID 1 distinto y su código de salida.

QUÉ EJECUTA DOCKER SEGÚN COMBINES ENTRYPOINT Y CMD (LA TABLA DE LA REFERENCIA DEL DOCKERFILE)

	SIN ENTRYPOINT	ENTRYPOINT EXEC_ENTRY P1_ENTRY (SHELL)	ENTRYPOINT ["EXEC_ENTRY", "P1_ENTRY"] (EXEC)
Sin CMD	error: no se permite	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"] (exec)	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd

	SIN ENTRYPOINT	ENTRYPOINT EXEC_ENTRY P1_ENTRY (SHELL)	ENTRYPOINT ["EXEC_ENTRY", "P1_ENTRY"] (EXEC)
CMD exec_cmd p1_cmd (shell)	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Las cuatro combinaciones con ambas instrucciones: solo exec + exec da lo que quieres. ENTRYPOINT en forma shell ignora el CMD y, en bases Debian, deja a sh como PID 1; ENTRYPOINT exec con CMD shell le pasa "/bin/sh -c ..." como argumentos. Si la imagen base define CMD y tú agregas ENTRYPOINT, ese CMD se borra: vuelve a declararlo.

CÓDIGOS DE SALIDA QUE VAS A VER

CÓDIGO	QUÉ PASÓ	CÓMO LO CONFIRMAS
0	Terminó bien, también si atrapó SIGTERM y salió limpio	el log muestra el apagado
1 u otro código	Lo eligió tu proceso (1: una excepción sin atrapar en Python)	docker logs: el traceback
125	Falló Docker mismo (flag inválida, imagen que no existe o pull denegado, error del daemon): el contenedor nunca corrió	el mensaje del CLI, p. ej. The container name "/gw" is already in use
126	El comando existe pero no se puede ejecutar (permisos, es un directorio)	permission denied: un script sin chmod +x
127	Comando no encontrado	exec: "uvicorn": executable file not found in $PATH
130 = 128 + 2	SIGINT: Ctrl+C	lo apretaste en un docker run -it
137 = 128 + 9	SIGKILL: el OOM killer o el fin del plazo (docker stop, docker kill)	inspect: OOMKilled true es memoria; false es plazo vencido o kill, salvo que nadie lo haya parado
143 = 128 + 15	Murió por SIGTERM: no lo manejó (lo típico con --init y un proceso sin handler) o lo re-lanzó al terminar de drenar (uvicorn, si no es PID 1)	inspect: ExitCode 143
255	Lo devuelve runc (no está en la tabla de Docker): el contenedor ya arrancó, pero el kernel no pudo ejecutar el entrypoint	docker logs: exec /ruta: exec format error (otra arquitectura) o no such file or directory (shebang con CRLF o intérprete ausente, como bash en Alpine)

125: falló Docker y el contenedor nunca corrió. 126 y 127: docker run cuando no pudo arrancar tu comando (el contenedor queda en Created), o un shell adentro con el mismo significado. 255: runc, con el contenedor ya arrancado. El resto es el exit code de tu proceso.

EJEMPLO REAL · CÓMO SE APAGA EL GATEWAY DE C01

DOCKERFILE **reference/c01/gateway/Dockerfile (últimas líneas)**

```
#   --timeout-graceful-shutdown 20  en SIGTERM termina los requests en curso (< terminationGracePeriodSeconds)
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080", "--workers", "1", \
     "--timeout-keep-alive", "75", "--no-access-log", "--timeout-graceful-shutdown", "20"]
```

- Lista JSON con comillas dobles: forma exec. uvicorn es el PID 1 y le llega directo el SIGTERM de `docker stop` o del kubelet.
- `--timeout-graceful-shutdown 20`: con SIGTERM, uvicorn deja de aceptar conexiones, espera hasta 20 s las requests en curso y corre el shutdown del lifespan. Sin el flag espera sin límite.
- En Kubernetes, `k8s/gateway.yaml` fija `terminationGracePeriodSeconds: 45` con un preStop de 15 s, y el plazo corre desde el preStop: después del SIGTERM al gateway le quedan 30 s (20 de drenado + margen). Por eso en Docker se usan 30 s: `docker stop -t 30` o `stop_grace_period: 30s`. Triton: preStop de 10 s + `--exit-timeout-secs=30` + margen = 60.
- El `docker-compose.yml` no fija `stop_grace_period`: Compose usa 10 s, menos que los 20 s de drenado. En desarrollo casi no importa; para ensayar el apagado de producción, agrégale tú `stop_grace_period: 30s` al gateway (es una sugerencia: no está en el archivo).
- El worker del caso C04 (`reference/c04/worker/Dockerfile`) parte de una base de NGC cuyo `ENTRYPOINT` imprime el banner y hace exec del `CMD`: el patrón `exec "$@"`, que deja a python como el proceso que recibe SIGTERM.

CÓMO RECORDARLA

el PID 1 solo recibe las señales que sabe manejar. Forma exec + un servidor que maneje SIGTERM (uvicorn, gunicorn, Triton); si no controlas el proceso, `--init`; en un script de entrada, `exec "$@"` al final. **137** = te mataron, **143** = murió por SIGTERM, **0** = salió solo.

TRAMPA

la forma exec no basta si tu proceso no maneja SIGTERM. Python no instala un handler (solo convierte SIGINT en `KeyboardInterrupt`), así que un script propio como PID 1 ignora `docker stop` y muere a los 10 s con 137. Y la forma shell no falla igual en todas las bases: en Debian y Ubuntu, `/bin/sh` (dash) se queda como PID 1; en Alpine (busybox) y donde `/bin/sh` es bash, `sh -c` suele hacer exec del último comando y tu proceso queda como PID 1 por casualidad: basta un pipe (`|`) para que deje de serlo.

En el curso: M5 · Un Dockerfile de ML, decisión por decisión · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · C01 · Archivos de referencia · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica

6 Redes y puertos

¿Quién puede llegar a tu contenedor, por qué puerto y con qué nombre?

SINTAXIS CON VARIABLES

`docker run -d -p` *ip_host* **`:`** *puerto_host* **`:`** *puerto_cont* *imagen*

Si omites la IP (`-p 8080:8080`), Docker publica en todas las interfaces (`0.0.0.0` y `[::]`); con `127.0.0.1`, solo tu máquina.

`docker network create` *red*

`docker run -d --name` *nombre* **`--network`** *red* *imagen*

`# otro contenedor de la red lo encuentra como` **`http://`** *nombre* **`:`** *puerto_cont*

En una red definida por ti, Docker resuelve los nombres de contenedor. Entre contenedores se usa el puerto **del contenedor**, sin `-p`.

`uvicorn` *app* **`--host 0.0.0.0 --port`** *puerto_cont*

Dentro del contenedor, escucha en todas las interfaces: con el default de uvicorn (`127.0.0.1`) solo aceptaría conexiones del propio contenedor.

`http://host.docker.internal:` *puerto_host*

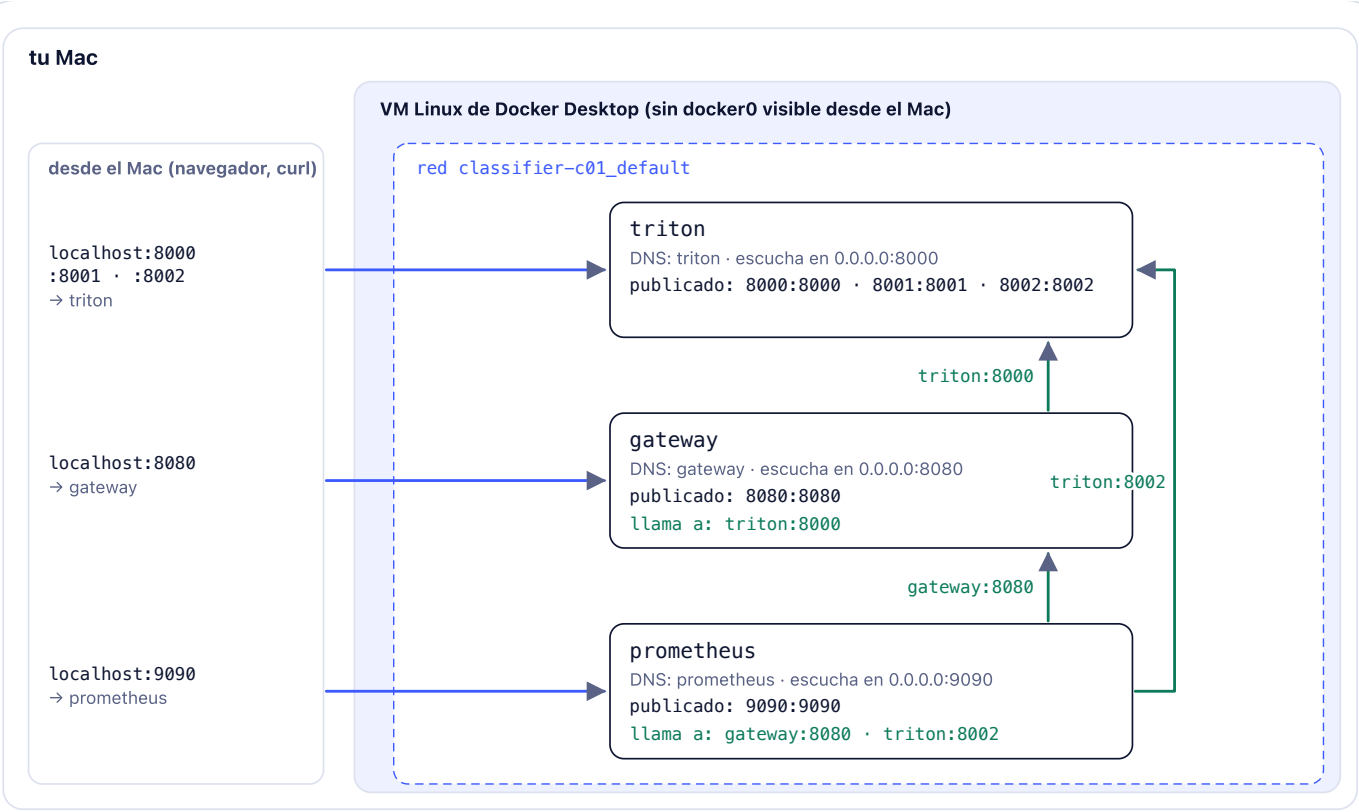
Desde un contenedor hacia un servicio de tu Mac (Docker Desktop). En un Linux sin Desktop: `--add-host host.docker.internal=host-gateway`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ip_host	interfaz del host donde se publica; si la omites, todas	`127.0.0.1`
puerto_host	puerto del host	`8080`
puerto_cont	puerto donde escucha el proceso adentro	`8080 (gateway), 8000 (Triton)`
imagen	imagen con su tag	`classifier-gateway:dev`
red	red definida por ti	`lab-net; en Compose, classifier-c01_default`
nombre	nombre del contenedor o del servicio: es su nombre DNS en la red	`triton`
app	módulo y objeto ASGI	`app.main:app`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`-p` *puerto_host* **`:`** *puerto_cont*	El host escucha en el primero y reenvía al segundo; primero va el host.	`-p 8080:80 (host 8080 → contenedor 80)`
`-p 127.0.0.1:` *puerto_host* **`:`** *puerto_cont*	Publica solo en el loopback del host: nadie más de tu red lo ve.	`-p 127.0.0.1:9090:9090`
`-P`	Publica todos los puertos de EXPOSE en puertos altos al azar del host.	`docker run -P … y luego docker port`
`EXPOSE`	Documenta el puerto para quien corre la imagen (y lo usa -P); no publica nada.	`EXPOSE 8080 (gateway de C01)`
`docker port`	Muestra qué puerto del host apunta a cuál del contenedor.	`8000/tcp -> 127.0.0.1:8001`
`bridge por defecto`	La red de docker run sin --network: los contenedores se ven solo por IP, sin DNS por nombre.	`docker network ls → bridge`
`Red definida por ti`	docker network create: DNS por nombre de contenedor y aislamiento del resto.	`docker network create lab-net`
`Red de Compose`	Compose crea la red del proyecto y cada servicio se resuelve por su nombre.	`classifier-c01_default · triton:8000`
`docker network connect`	Conecta a otra red un contenedor que ya corre.	`docker network connect lab-net lab-web`
`localhost adentro`	Es el propio contenedor: ni tu Mac ni el contenedor de al lado.	`default localhost:8000 de TRITON_URL en app/main.py`
`0.0.0.0 vs 127.0.0.1 (bind)`	Adentro, 0.0.0.0 acepta lo que llega por la red de Docker; 127.0.0.1, solo lo del propio contenedor.	`--host 0.0.0.0 en el CMD del gateway`
`gunicorn -b`	Su default es 127.0.0.1:8000 (0.0.0.0:$PORT si existe PORT): en un contenedor, pásale -b 0.0.0.0.	`-b 0.0.0.0:8080`
`host.docker.internal`	Nombre que Docker Desktop resuelve a la IP interna de tu Mac.	`curl http://host.docker.internal:8000`



- Desde el Mac: localhost:8080, localhost:9090 (puertos publicados por Docker Desktop).
- Entre contenedores: servicio:puerto del contenedor, sin -p.
- localhost dentro del gateway es el gateway.
- Del contenedor a tu Mac: host.docker.internal.
- Adentro, cada servidor escucha en 0.0.0.0.

La red de Compose de C01 vista desde tu Mac: entras por los puertos publicados; adentro, cada servicio es un nombre DNS y se usa el puerto del contenedor.

EJEMPLO REAL · PUERTOS Y NOMBRES EN EL COMPOSE DE C01

YAML `reference/c01/docker-compose.yml (extracto)`

```yaml
name: classifier-c01  # prefijo de contenedores y red: no choca con otros proyectos compose

services:
  triton:
    # (…)
    ports:
      - "8000:8000"  # HTTP/REST KServe v2 (el gateway) y /v2/health/*
      - "8001:8001"  # gRPC (perf_analyzer, clientes que lo prefieran)
      - "8002:8002"  # métricas Prometheus nv_*
    # (…)

  gateway:
    # (…)
    ports:
      - "8080:8080"  # la API pública: POST /v1/classify
    environment:  # misma configuración que en k8s, solo cambian los valores
      TRITON_URL: "triton:8000"  # nombre del servicio en la red de compose, sin http://
      # (…)
```

- `name: classifier-c01` le pone nombre al proyecto: Compose crea la red `classifier-c01_default` y los contenedores `classifier-c01-triton-1`, `classifier-c01-gateway-1`.
- `TRITON_URL: "triton:8000"`: nombre del servicio + puerto del contenedor. Si faltara, `app/main.py` usa su default `localhost:8000`, que dentro del gateway es el gateway mismo, y `/readyz` queda en 503.
- Los `ports:` son para entrar desde tu Mac (curl, perf_analyzer, la UI de Prometheus). Entre contenedores no hacen falta: Prometheus scrapea `gateway:8080` y `triton:8002` por la red.
- `EXPOSE 8080` en el Dockerfile del gateway solo documenta; lo que abre el puerto es `"8080:8080"`. Y su `CMD` lleva `--host 0.0.0.0` para que ese puerto llegue al proceso.

- Los puertos van entre comillas porque el spec de Compose recomienda escribirlos siempre como string: YAML puede leer `xx:yy` como un número en base 60.

CÓMO RECORDARLA

-p es host:contenedor y solo sirve para entrar desde afuera; entre contenedores usas **servicio:puerto del contenedor** en una red con nombre (Compose ya crea una). Adentro, `localhost` es el propio contenedor y el servidor escucha en `0.0.0.0`.

TRAMPA

`-p 8080:8080` sin IP publica en todas las interfaces. Docker Desktop deja ese puerto accesible desde tu red local, y en un servidor Linux la referencia de Compose advierte que así se saltan las reglas del firewall del host. Publica lo interno (Prometheus, una base, un notebook) en `127.0.0.1`. Y desde tu Mac no llegas a la IP de un contenedor, porque el bridge vive dentro de la VM: usa el puerto publicado.

En el curso: M5 · docker compose para desarrollo local · Docker y kubectl · docker compose · Docker y kubectl · Docker: construir, correr, depurar · C01 · Archivos de referencia

7 Volúmenes y permisos

¿Dónde quedan tus datos cuando el contenedor muere, y quién puede escribirlos?

SINTAXIS CON VARIABLES

`-v` *ruta_host* `:` *ruta_cont* `:ro`

`--mount type=bind,src=` *ruta_host* **`,dst=`** *ruta_cont* **`,readonly`**

Bind mount: un archivo o carpeta de tu máquina adentro del contenedor; `:ro` para que el proceso solo pueda leer.

`-v` *volumen* `:` *ruta_cont*

`--mount type=volume,src=` *volumen* **`,dst=`** *ruta_cont*

Volumen nombrado: lo maneja Docker, se crea si no existe y sobrevive a `docker rm`.

`--tmpfs` *ruta_cont* **`:size=`** *tamaño*

`--mount type=tmpfs,dst=` *ruta_cont* **`,tmpfs-size=`** *tamaño*

En memoria: rápido, cuenta para `--memory` y se borra al parar el contenedor.

`RUN mkdir -p` *ruta_cont* **`&& chown`** *uid* `:` *gid* *ruta_cont*

`USER` *uid* `:` *gid*

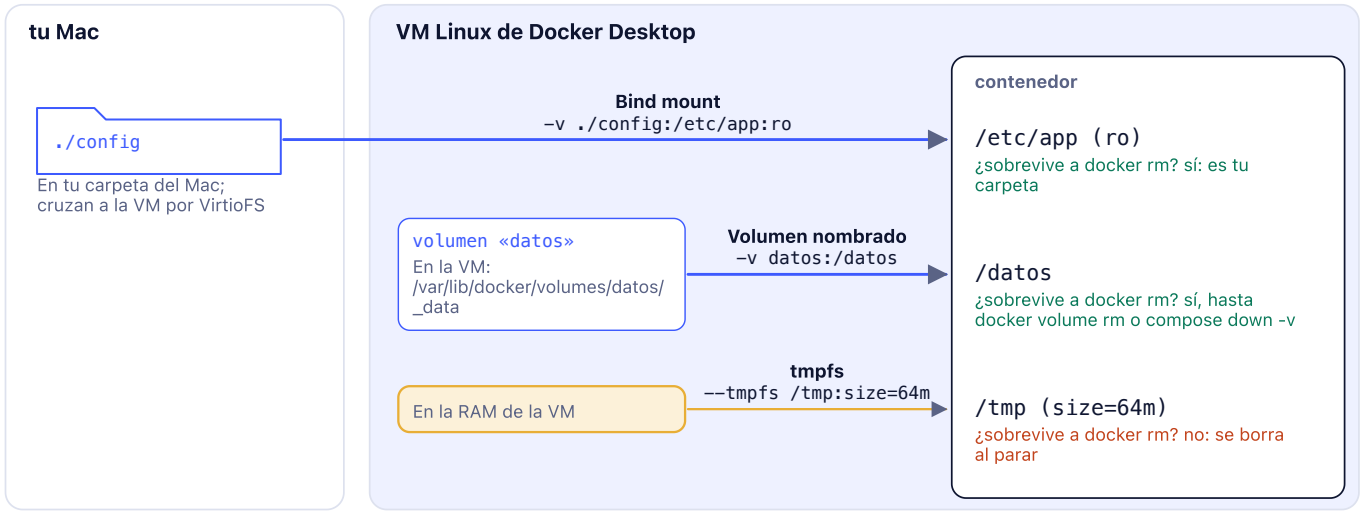
La carpeta de datos ya existe en la imagen con el dueño del `USER`: un volumen nuevo montado ahí hereda ese dueño.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ruta_host	archivo o carpeta de tu máquina; en Compose, relativa con ./	`./monitoring/prometheus.yml`
ruta_cont	ruta absoluta dentro del contenedor	`/etc/prometheus/prometheus.yml`
volumen	nombre del volumen	`prom-data`
tamaño	tope del tmpfs; sin tope, hasta el 50 % de la RAM del host	`64m`
uid	UID numérico del USER	`10001 (gateway de C01); 65534 = nobody (Prometheus)`
gid	GID numérico	`10001`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Bind mount	Archivo o carpeta del host dentro del contenedor; tapa lo que la imagen tenía en esa ruta.	`./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro`
Volumen nombrado	Almacenamiento de Docker; si está vacío al crear el contenedor, copia lo que la imagen tenía ahí, con dueño.	`-v prom-data:/prometheus`
Volumen anónimo	Como el nombrado, con nombre al azar; lo crean VOLUME o -v /ruta, y --rm lo borra.	`VOLUME ["/prometheus"] en la imagen de Prometheus`
tmpfs	Sistema de archivos en RAM: cuenta para --memory y desaparece al parar.	`--tmpfs /tmp:size=64m`
:ro · readonly	Monta en solo lectura; escribir da Read-only file system.	`la config de Prometheus en C01`
-v vs --mount	-v crea la carpeta del host si falta; --mount falla con bind source path does not exist.	`--mount type=bind,src=./config,dst=/config`
--read-only	La capa del contenedor queda de solo lectura; solo se escribe en los montajes (volúmenes, bind mounts sin :ro, tmpfs).	`--read-only --tmpfs /tmp (como readOnlyRootFilesystem en k8s)`
--user *uid* `:` *gid*	UID y GID del proceso. Los permisos se comparan por número, no por nombre.	`--user "$(id -u):$(id -g)"`
chown en el build	Deja la carpeta de datos con el dueño del USER; los volúmenes nuevos lo heredan.	`RUN chown -R nobody:nobody /etc/prometheus /prometheus`
Capa escribible	Lo que el proceso escribe fuera de volúmenes; se pierde con docker rm.	`un log escrito en /srv sin volumen`
docker volume ls · inspect · rm · prune	Lista, muestra dónde vive (Mountpoint), borra uno; prune borra los anónimos sin uso (-a, también nombrados).	`docker volume inspect -f '{{.Mountpoint}}' prom-data`
VirtioFS (Docker Desktop)	Cómo pasan tus carpetas del Mac a la VM: el default; la alternativa es gRPC FUSE.	`se ve en Settings > General (no hace falta tocarlo)`



	BIND MOUNT	VOLUMEN NOMBRADO	TMPFS
Con -v	-v ./config:/etc/app:ro	-v datos:/datos	--tmpfs /tmp:size=64m
Con --mount	type=bind,src=./config,dst=/etc/app,readonly	type=volume,src=datos,dst=/datos	type=tmpfs,dst=/tmp,tmpfs-size=64m
Dónde viven los datos	En tu carpeta del Mac; cruzan a la VM por VirtioFS	En la VM: /var/lib/docker/volumes/datos/_data	En la RAM de la VM
Si la ruta del host no existe	-v la crea como carpeta; --mount falla	el volumen se crea	no aplica
Lo que la imagen tenía en esa ruta	queda tapado	si el volumen está vacío, se copia con dueño y permisos	queda tapado
Dueño de la raíz	el del host; en Linux se compara su UID con el del proceso	root, o el dueño que esa carpeta tenía en la imagen	root, modo 1777
¿Sobrevive a docker rm?	sí: es tu carpeta	sí, hasta docker volume rm o compose down -v	no: se borra al parar
Velocidad en tu Mac	la más lenta con muchos archivos chicos	la del disco de la VM	la de la memoria
Úsalo para	código y config en desarrollo	bases, caches, modelos descargados	temporales que no deben tocar disco

- Todo lo de la VM (imágenes, capas, volúmenes) vive en el disco virtual de Docker Desktop: ~/Library/Containers/com.docker.docker/Data/vms/0/data/Docker.raw.
- Docker Desktop comparte por defecto /Users, /Volumes, /private, /tmp y /var/folders.

Los tres montajes: de dónde salen los datos, quién es el dueño y qué sobrevive. En Docker Desktop, todo lo que no es tu carpeta vive dentro de la VM.

EJEMPLO REAL · LOS MONTAJES DE C01, Y EL QUE LE FALTA

YAML `reference/c01/docker-compose.yml (servicio prometheus, sin su depends_on)`

```yaml
  prometheus:
    image: prom/prometheus:v3.13.3  # tag fijo (nunca latest): 3.13 es la LTS vigente (soporte hasta 2027-07-31); sube de parche dentro de 3.13.x
    ports:
      - "9090:9090"  # UI: http://localhost:9090/alerts muestra las 6 alertas y su estado
    volumes:  # montados en la ruta que usa el CMD por defecto de la imagen: no hace falta `command`
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - ./monitoring/alerts.yml:/etc/prometheus/alerts.yml:ro
```

- Bind mounts de archivos sueltos con `:ro`: la config entra desde el repo y Prometheus solo la lee. Cambiarla no exige reconstruir nada: basta con que Prometheus la vuelva a leer (reiniciar el contenedor alcanza).
- Las rutas con `./` son relativas a la carpeta del compose: `docker compose config` las muestra absolutas, con `type: bind` y `read_only: true`.

- Lo que no se monta: la imagen de Prometheus declara `VOLUME ["/prometheus"]` para su base de series. Sin un volumen nombrado, Docker crea uno anónimo por contenedor: `docker compose down -v` lo borra y un `down` + `up` arranca con otro vacío.
- Para conservarla: `- prom-data:/prometheus` en el servicio y `prom-data:` bajo un `volumes:` de primer nivel. Funciona sin chown porque la imagen ya trae `/prometheus` con dueño `nobody` y un volumen nuevo copia ese dueño. Un bind mount de una carpeta tuya, en Linux, falla con `open /prometheus/queries.active: permission denied`.
- El gateway muestra el otro lado: el código entra con dueño root y el proceso corre como `USER 10001`, así que no puede modificarse. En k8s el único lugar escribible es un `emptyDir` en `/tmp` (`k8s/gateway.yaml`); en Docker, el equivalente es `--read-only --tmpfs /tmp`.

CÓMO RECORDARLA

bind = tu carpeta, volumen = carpeta de Docker, tmpfs = RAM. Los permisos se comparan por número: crea la carpeta de datos en el Dockerfile con el dueño del `USER` y un volumen nuevo hereda ese dueño. Nunca `chmod 777` : arregla el dueño.

TRAMPA

tu Mac no es buen juez de permisos. En Docker Desktop, un bind mount pasa por el sistema de archivos compartido de la VM y los dueños no se comportan como en Linux: lo que funciona en tu laptop puede fallar en el servidor Linux, donde el UID 10001 del contenedor no coincide con el dueño de la carpeta del host. Valida con volúmenes nombrados (viven en la VM, con permisos de Linux de verdad) o en CI sobre Linux. Y en Mac, los bind mounts con miles de archivos chicos (un venv, un dataset de imágenes) son lentos: ponlos en un volumen.

En el curso: M5 · docker compose para desarrollo local · M5 · Un Dockerfile de ML, decisión por decisión · M5 · Imagen, contenedor y capas · C01 · Archivos de referencia

8 Docker Compose

¿Cómo levantas varios servicios en orden, con su configuración, y los apagas sin perder datos?

SINTAXIS CON VARIABLES

```
services:
  servicio:
    build: contexto
    ports:
      - "puerto_host:puerto_cont"
    depends_on:
      dependencia:
        condition: service_healthy
```

Un bloque por servicio; su nombre es también su nombre DNS en la red del proyecto. `build` construye desde una carpeta; `image` baja una imagen hecha.

```
  dependencia:
    image: imagen
    healthcheck:
      test: ["CMD", "curl", "-fsS", "url_salud"]
      interval: intervalo
      retries: reintentos
      start_period: arranque
```

El healthcheck corre **dentro** del contenedor: el binario tiene que existir en la imagen (Triton trae curl; una slim de Python, no).

```
docker compose up --build -d
docker compose logs -f servicio
docker compose exec servicio comando
docker compose down
```

Construir y levantar en segundo plano, seguir un servicio, entrar a uno que corre y bajar todo. `down -v` borra además los volúmenes.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
servicio	nombre del servicio (y su nombre DNS)	`gateway`
contexto	carpeta con el Dockerfile, relativa al compose	`./gateway`
puerto_host	puerto de tu Mac	`8080`
puerto_cont	puerto del contenedor	`8080`
dependencia	servicio que tiene que estar sano antes	`triton`
imagen	imagen con tag fijo	`prom/prometheus:v3.13.3`
url_salud	endpoint que responde 2xx solo si el servicio puede atender	`http://localhost:8000/v2/health/ready`
intervalo	interval: cada cuánto corre el chequeo (default 30s)	`5s`
reintentos	retries: fallos seguidos para declarar unhealthy (default 3)	`3`
arranque	start_period: tiempo de arranque en que los fallos no cuentan (default 0s)	`60s`
comando	lo que corres dentro del servicio	`sh, o curl -fsS localhost:8000/v2/health/ready`
$t_{\text{unhealthy}}$	cuándo se declara unhealthy un servicio que nunca queda listo (con start_period, el primer fallo que cuenta cae al terminar el start_period)	`C01: 60 + 2 × 5 ≈ 70 s`

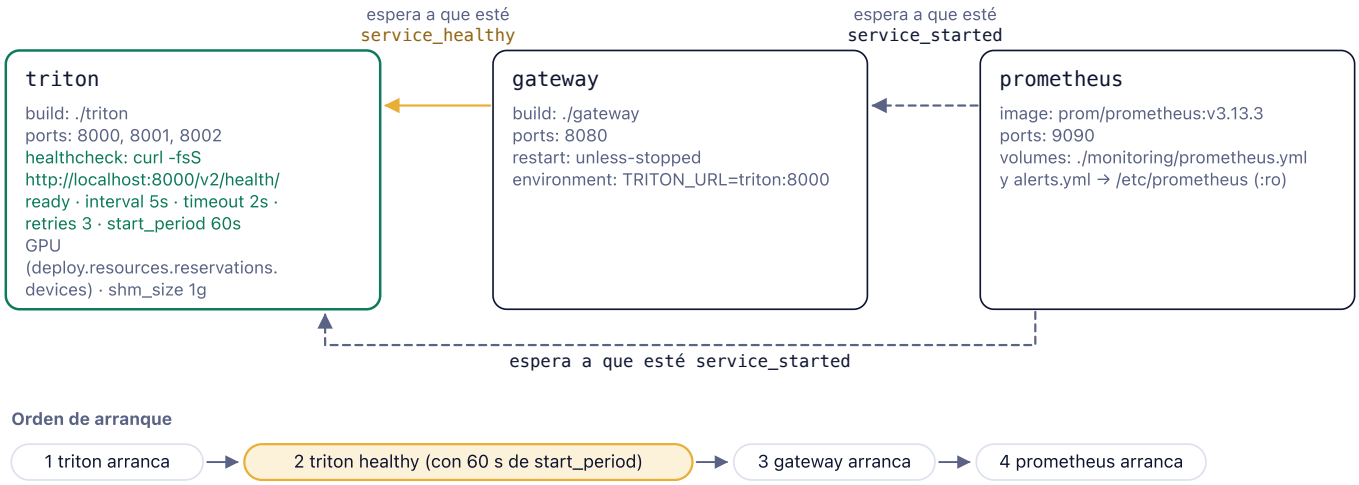
FÓRMULAS

$$t_{\text{unhealthy}} \approx \text{arranque} + (\text{reintentos} - 1) \times \text{intervalo}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
build · image	build construye desde una carpeta (imagen proyecto-servicio); image baja una imagen de un registry.	`build: ./gateway · image: prom/prometheus:v3.13.3`
ports	Igual que -p, siempre entre comillas; solo para entrar desde tu máquina.	`"127.0.0.1:9090:9090"`
environment · env_file	Variables del contenedor; si ambas definen la misma, gana environment.	`TRITON_URL: "triton:8000"`
.env del proyecto	Solo sirve para interpolar ${VAR} en el YAML; no entra solo al contenedor.	`image: app:${TAG:-dev}`
volumes	Con ./ es un bind mount; con un nombre, un volumen (declarado en volumes: de primer nivel).	`./monitoring/alerts.yml:/etc/prometheus/alerts.yml:ro`

PIEZA	QUÉ HACE	EJEMPLO
networks	Por defecto, una red del proyecto donde cada servicio se resuelve por su nombre.	`classifier-c01_default`
depends_on + condition	service_started (default), service_healthy o service_completed_successfully: qué esperar antes de arrancar.	`triton: condition: service_healthy`
healthcheck	test, interval, timeout, retries, start_period, start_interval; defaults 30s, 30s, 3, 0s, 5s.	`Triton en C01: 5s, 2s, 3, 60s`
restart	Igual que --restart: mira el exit code, nunca la salud.	`restart: unless-stopped`
profiles	El servicio solo arranca con --profile o si lo nombras en el comando; sin profiles, siempre.	`profiles: [debug]`
stop_grace_period · init	Plazo entre SIGTERM y SIGKILL (10 s por defecto); init: true pone tini como PID 1.	`stop_grace_period: 30s (sugerencia para el gateway)`
develop.watch	Al guardar: sync copia, rebuild reconstruye, sync+restart copia y reinicia. Solo servicios con build.	`docker compose up --watch`
up --build -d · logs -f · exec	Construir y levantar en segundo plano; seguir los logs de un servicio; entrar a uno que corre.	`docker compose exec triton nvidia-smi`
down -v · config	down borra contenedores y red (-v, también volúmenes); config valida y muestra el YAML final.	`docker compose config -q`



El compose de C01: Triton tiene que estar healthy para que arranque el gateway; Prometheus solo espera a que los otros dos arranquen.

EJEMPLO REAL · EL ORDEN DE ARRANQUE DE C01

YAML `reference/c01/docker-compose.yml (extracto: sin la GPU, shm_size ni Prometheus)`

```yaml
name: classifier-c01  # prefijo de contenedores y red: no choca con otros proyectos compose

services:
  triton:
    build: ./triton  # imagen con el modelo horneado (misma que se sube al registry para k8s)
    # (…)
    healthcheck:
      # ready = servidor arriba Y modelo cargado (con --strict-readiness); la imagen de Triton trae curl
      test: ["CMD", "curl", "-fsS", "http://localhost:8000/v2/health/ready"]
      interval: 5s
      timeout: 2s
      retries: 3
      start_period: 60s  # cargar el engine + warmup tarda; no marcar unhealthy mientras tanto

  gateway:
    build: ./gateway  # imagen multi-stage de gateway/Dockerfile
    ports:
      - "8080:8080"  # la API pública: POST /v1/classify
```

```
    environment:  # misma configuración que en k8s, solo cambian los valores
      TRITON_URL: "triton:8000"  # nombre del servicio en la red de compose, sin http://
      # (…)
    depends_on:
      triton:
        condition: service_healthy  # no arranca hasta que el modelo esté listo (evita 503 al inicio)
    restart: unless-stopped  # si el proceso muere, Docker lo levanta (en k8s lo hace el kubelet)
```

- El healthcheck de Triton le pega a `/v2/health/ready`, que con `--strict-readiness` solo da 200 con el modelo cargado: sano = listo para inferir, no solo proceso vivo.
- `start_period: 60s`: mientras carga el engine, los fallos no cuentan. Si nunca queda listo, se declara unhealthy a los ≈ 60 + 2 × 5 = 70 s (el primer fallo que cuenta cae al terminar el start_period) y `up` falla con `dependency failed to start: container classifier-c01-triton-1 is unhealthy`.
- `condition: service_healthy`: Compose espera a que Triton esté healthy antes de arrancar el gateway. Con `depends_on` corto solo esperaría a que el contenedor arranque, y el gateway respondería 503 mientras carga el modelo.
- `restart: unless-stopped` levanta el gateway si su proceso muere; no hace nada si Triton se vuelve unhealthy después: Docker no reinicia por salud.
- Para iterar sobre el gateway: `docker compose up -d --build gateway` reconstruye y recrea el gateway; Triton sigue corriendo si no cambió. Con `develop.watch` y `sync`, ojo: el código del gateway es de root y el proceso corre como 10001, y watch exige que el `USER` pueda escribir el destino; con `rebuild` no hay ese problema.
- Correrlo entero necesita GPU NVIDIA y el NVIDIA Container Toolkit: en tu Mac, practica el patrón con el lab.

CÓMO RECORDARLA

depends_on ordena; la condición decide qué se espera: `service_started` (default) espera al contenedor, `service_healthy` espera a su healthcheck. Ciclo diario: `up --build -d` → `logs -f` → `exec` → `down`, y `-v` solo si quieres perder los datos.

TRAMPA

`depends_on: [db]` sin condición no espera a que la base acepte conexiones, y `restart` no mira la salud: un servicio unhealthy no se reinicia solo. Además, `docker compose down -v` borra los volúmenes del proyecto y la base de desarrollo se va con ellos. Y el `.env` de la carpeta solo interpola el YAML: para que una variable llegue al contenedor, pásala en `environment` o `env_file`.

En el curso: M5 · docker compose para desarrollo local · Docker y kubectl · docker compose · M6 · Los objetos que sí debes conocer · C01 · Archivos de referencia

9 GPU y CUDA en contenedores

¿Qué trae tu imagen, qué pone el host y cómo le pides la GPU a Docker?

SINTAXIS CON VARIABLES

`docker run --rm --gpus` *gpus* *imagen*`:`*tag* `nvidia-smi`

Prueba mínima. Si falla con una imagen chica, el problema está en el host (driver o toolkit), no en tu imagen.

`docker run --rm --device nvidia.com/gpu=`*id* *imagen*`:`*tag* `nvidia-smi -L`

La misma petición con CDI. Desde Engine 29.2, `--gpus` también usa CDI cuando el toolkit generó la especificación.

`devices: [{ driver: nvidia, count:` *n* `, capabilities: [gpu] }]`

En Compose, bajo `deploy.resources.reservations`. `capabilities` es obligatorio; `count` o `device_ids`, nunca los dos.

`pip install torch==`*version* `--index-url https://download.pytorch.org/whl/`*variante*

La variante fija el CUDA que viaja en la imagen: `cu126` (driver ≥ 525), `cu130` (≥ 580) o `cpu` (sin CUDA).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
gpus	qué GPUs ve el contenedor: all, un número, device=0 o '"device=0,2"'	`all` (model/build_engine.sh de C01)
imagen : *tag*	la imagen que pruebas	`nvidia/cuda:12.6.3-base-ubuntu22.04` o `classifier-triton:2026-09-01-resnet50-fp16`
id	nombre del device en la especificación CDI: all, un índice o un UUID	`all`
n	cuántas GPUs reserva el servicio (o all)	`1` (servicio triton del docker-compose.yml de C01)
version	versión de torch fijada con ==	`2.14.0` (model/requirements.txt de C01; desde PyPI trae CUDA 13.0)
variante	índice de wheels de PyTorch: cpu, cu126, cu130 o cu132 (los de torch 2.14.0)	`cu126` si el driver del nodo es R535
$\text{CUDA}_{\text{imagen}}$	el CUDA que trae la imagen: runtime y librerías	`12.6` (tritonserver:24.08-py3)
$\text{CUDA}_{\text{max}}(\text{driver})$	el CUDA más nuevo que soporta el driver: lo que dice nvidia-smi en CUDA Version	`12.2` con un driver R535

FÓRMULAS

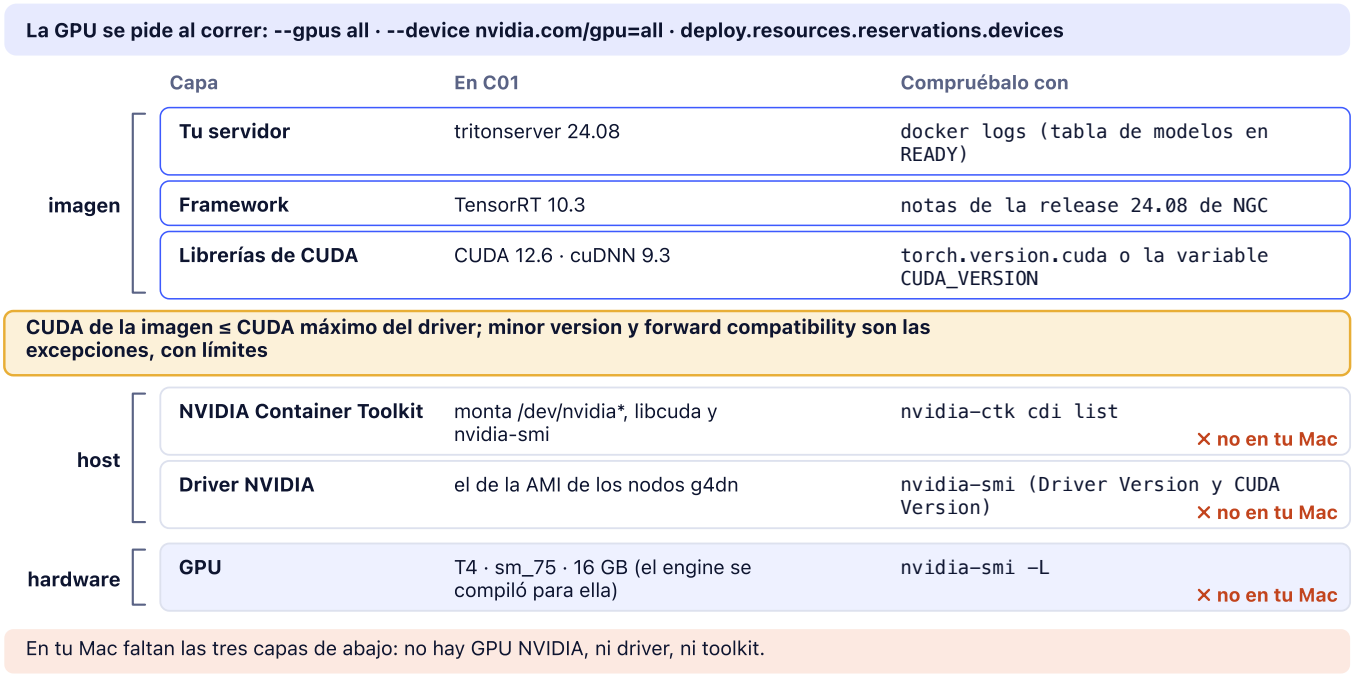
$$\text{CUDA}_{\text{imagen}} \leq \text{CUDA}_{\text{max}}(\text{driver del host})$$

$12.x \Rightarrow \text{driver} \geq 525 \qquad 13.x \Rightarrow \text{driver} \geq 580 \qquad \text{(minor version compatibility)}$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Driver NVIDIA	Módulo del kernel y libcuda del host. Su versión fija el CUDA máximo; nunca va en la imagen.	`nvidia-smi → Driver Version`
NVIDIA Container Toolkit	Se instala en el host; al crear el contenedor monta /dev/nvidia*, libcuda y nvidia-smi.	`sudo nvidia-ctk runtime configure --runtime=docker`
`--gpus all · --gpus 2 · --gpus device=0`	Pide GPUs al crear el contenedor. Sin este flag (ni --device por CDI) no ve ninguna, salvo con runtime nvidia por defecto.	`docker run --rm --gpus device=0 IMAGEN nvidia-smi`
`--gpus '"device=0,2"'`	Lista con coma: comillas dobles dentro de simples, porque la coma separa opciones.	`solo las GPUs 0 y 2`
`--device nvidia.com/gpu=all`	La GPU por CDI: el toolkit (≥ 1.18) genera la especificación en /var/run/cdi.	`--device nvidia.com/gpu=0`

PIEZA	QUÉ HACE	EJEMPLO
`nvidia-ctk cdi list`	Lista los nombres CDI que el host puede inyectar.	`nvidia.com/gpu=0 · nvidia.com/gpu=all`
`NVIDIA_VISIBLE_DEVICES`	Qué GPUs expone el runtime: all, índices, UUIDs, none o void.	`all en nvidia/cuda y en las imágenes de NGC`
`NVIDIA_REQUIRE_CUDA`	Requisito de la imagen: el hook clásico no arranca el contenedor si el driver no cumple; CDI no lo verifica.	`empieza con cuda>=12.6 en nvidia/cuda:12.6.3`
`-runtime vs -devel`	runtime trae librerías para correr; devel suma nvcc, headers y estáticas.	`12.6.3: ≈ 1.5 GB contra ≈ 4.1 GB comprimidas (amd64)`
`torch cpu · cu126 · cu130`	El índice de wheels elige qué CUDA viaja con torch, o ninguno.	`wheel cpu ≈ 0.2 GB; cu126 ≈ 0.9 GB más librerías nvidia-* (amd64)`
`deploy.resources.reservations.devices`	La GPU en Compose: driver, count o device_ids, y capabilities: [gpu].	`count: 1 en C01`
`--shm-size=1g · shm_size: 1g`	/dev/shm es de 64 MB por defecto: corto para Triton y el DataLoader.	`shm_size: 1g en C01`
`torch.cuda.is_available()`	Prueba dentro de TU imagen: driver, toolkit, flag y wheel tienen que estar bien a la vez.	`False = falta alguna de esas piezas`



Arriba, lo que viaja en la imagen; abajo, lo que pone el host y el hardware. La regla vive en la frontera: el CUDA de la imagen tiene que caber debajo del driver.

EJEMPLO REAL · EL DOCKERFILE DE TRITON DE C01

DOCKERFILE `reference/c01/triton/Dockerfile (extracto)`

```
# Release de NGC: DEBE ser el mismo YY.MM que la imagen tensorrt usada en model/build_engine.sh
# (misma versión de TensorRT). Un engine de otra versión no carga.
FROM nvcr.io/nvidia/tritonserver:24.08-py3

# El model repository completo: classifier/config.pbtxt + classifier/1/model.plan
COPY model_repository /models

# Falla el BUILD si falta el engine (no se versiona en git): mejor aquí que un pod en CrashLoop
RUN test -f /models/classifier/1/model.plan

# 8000 HTTP/REST (KServe v2, lo usa el gateway) · 8001 gRPC · 8002 métricas Prometheus (nv_*)
EXPOSE 8000 8001 8002

CMD ["tritonserver", \
     "--model-repository=/models", \
```

```
    "--model-control-mode=none", \
    "--disable-auto-complete-config", \
    "--exit-on-error=true", \
    "--strict-readiness=true", \
    "--http-port=8000", \
    "--grpc-port=8001", \
    "--metrics-port=8002", \
    "--exit-timeout-secs=30", \
    "--log-format=ISO8601"]
```

- `FROM …tritonserver:24.08-py3` : la base trae CUDA 12.6, cuDNN 9.3 y TensorRT 10.3. El driver **no** viene: lo monta el toolkit del host al arrancar el contenedor.
- El Dockerfile no menciona la GPU: se pide al **correr** (`--gpus` o la reserva de Compose). Construir esta imagen no necesita GPU; cargar el engine, sí.
- `RUN test -f …/model.plan` : el engine se compila aparte, en una T4 y con `tensorrt:24.08-py3` (`build_engine.sh` corre `docker run --rm --gpus all …`). Un engine de otra GPU u otro TensorRT no carga.
- Driver del nodo: CUDA 12.6 pide ≥ 560, o las ramas R470, R525, R535 y R545 en GPUs de data center, gracias a la forward compatibility que traen las imágenes de NGC. La T4 es de data center; una GeForce, no.
- `--exit-on-error=true` y `--strict-readiness=true` : sin GPU, el modelo no carga y Triton termina, así que el healthcheck de Compose nunca llega a `healthy` . Es una falla ruidosa a propósito.
- La imagen pesa varios GB comprimida y también se publica para arm64: en tu Mac el pull funcionaría… y no te serviría, porque no hay GPU NVIDIA. No la bajes para practicar.

CÓMO RECORDARLA

la imagen trae CUDA, el host trae el driver y el flag trae la GPU. Si falta cualquiera de los tres, `torch.cuda.is_available()` da `False` ; la cabecera de `nvidia-smi` dice qué CUDA aguanta el driver, no cuál trae tu imagen.

TRAMPA

un servicio con `"cuda" if torch.cuda.is_available() else "cpu"` arranca feliz sin GPU (olvidaste `--gpus` , el driver es viejo o instalaste la wheel `cpu`) y corre decenas de veces más lento sin un solo error. Si esperas GPU, verifícala al arrancar y termina el proceso si no está. Y nunca instales el driver dentro de la imagen.

En el curso: M5 · Imágenes CUDA: runtime vs devel, drivers y compatibilidad · M5 · GPUs dentro del contenedor: NVIDIA Container Toolkit · Docker y kubectl · GPUs en Docker · C01 · Archivos de referencia · M6 · GPUs en Kubernetes

10 arm64 (tu Mac) vs amd64 (el servidor)

¿Para qué CPU es tu imagen y qué pasa cuando no coincide con la del servidor?

SINTAXIS CON VARIABLES

`docker build --platform linux/amd64 -t` *imagen* **:** *tag* *contexto*

En tu Mac, sin `--platform` la imagen sale `linux/arm64`. Para el servidor, amd64; mejor aún, que la construya CI.

`docker buildx build --platform linux/amd64,linux/arm64 -t` *registry* **/** *imagen* **:** *tag* **`--push`** *contexto*

Multi-arquitectura: un índice con un manifest por plataforma, y cada máquina baja el suyo.

`docker image inspect --format '{{.Os}}/{{.Architecture}}'` *imagen* **:** *tag*

Imagen local: ¿de qué arquitectura es? Con `--platform linux/amd64` miras una variante de una imagen multi-plataforma.

`docker buildx imagetools inspect` *registry* **/** *imagen* **:** *tag*

Imagen en el registry: lista las plataformas del índice leyendo el manifest, sin descargar capas.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
imagen : *tag*	nombre y versión de la imagen	`classifier-gateway:3f9c2ab`
contexto	directorio que se manda al builder	`gateway/ (desde reference/c01)`
registry	dónde vive la imagen para los nodos	`123456789012.dkr.ecr.us-east-1.amazonaws.com`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`linux/amd64 (x86_64)`	La CPU de casi todos los servidores y nodos de nube.	`g4dn y c6i de C01; uname -m → x86_64`
`linux/arm64 (aarch64, arm64/v8)`	Tu Mac con Apple Silicon; también Graviton, Grace y Jetson.	`uname -m → aarch64 dentro de un contenedor en tu Mac`
`--platform`	Elige la variante al construir, bajar o correr. Sin él, Docker usa la del host.	`--platform linux/amd64`
`Índice (manifest list)`	Un tag que apunta a un manifest por plataforma; cada cliente baja el suyo.	`python:3.12-slim: 8 plataformas`
`Attestation manifest`	La provenance que el build agrega por defecto; en el índice aparece como unknown/unknown.	`--provenance=false la quita`
`Emulación (QEMU o Rosetta)`	Docker Desktop traduce binarios amd64 en tu Mac: funciona, pero mucho más lento.	`QEMU viene incluido; Rosetta es opcional`
`WARNING: … does not match the detected host platform`	Corriste una imagen de otra plataforma sin pedirla: va emulada.	`linux/amd64 contra linux/arm64/v8`
`exec format error`	El kernel no reconoce el binario: otra arquitectura sin emulación, o un script sin shebang.	`exec /usr/local/bin/python3: exec format error`
`no matching manifest for … in the manifest list entries`	El índice no trae tu plataforma: el pull falla antes de bajar capas.	`no matching manifest for linux/amd64 …`
`docker image inspect --platform`	Inspecciona una variante de una imagen multi-plataforma local (API 1.49 o más).	`--platform linux/amd64`
`docker buildx imagetools inspect`	Plataformas y digests de una imagen remota, sin hacer pull.	`Platform: linux/amd64`
`Wheels por arquitectura`	pip elige el wheel de la CPU del contenedor; si falta, compila o falla.	`xformers 0.0.35 en Linux: solo x86_64`
`--only-binary=:all:`	Si falta el wheel, el build falla en ese paso, en vez de compilar bajo emulación.	`el pip install del gateway de C01`

Cómo construiste	Imagen	corre en tu Mac (arm64)	corre en servidor (amd64)
docker build en tu Mac, sin --platform	`linux/arm64`	✓ nativa	✗ exec format error, o no matching manifest al bajarla
docker build --platform linux/amd64, o CI en un runner amd64	`linux/amd64`	✓ emulada: mucho más lenta	✓ nativa
docker buildx build --platform linux/amd64,linux/arm64 --push	índice amd64 + arm64	✓ baja arm64: nativa	✓ baja amd64: nativa
imagen de GPU (Triton 24.08 de C01)	índice amd64 + arm64, con CUDA	✗ baja arm64, pero no hay GPU NVIDIA: el modelo no carga	✓ si el driver alcanza

Filas: dónde y cómo construiste la imagen. Columnas: dónde corre. Que corra en tu Mac no prueba nada sobre el servidor.

EJEMPLO REAL · EL GATEWAY DE C01: WHEELS VERIFICADOS PARA X86_64

DOCKERFILE **reference/c01/gateway/Dockerfile (extracto)**

```
# Misma versión de Python que en los tests (3.13); en producción fija además el digest (@sha256:…)
FROM python:3.13-slim AS builder
# PIP_NO_CACHE_DIR: el cache de pip no sirve dentro de una imagen, solo pesa
ENV PIP_NO_CACHE_DIR=1 PIP_DISABLE_PIP_VERSION_CHECK=1
# venv en ruta fija: se copia entero a la etapa final y el PATH lo encuentra
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
# Primero SOLO requirements.txt: esta capa queda en cache mientras no cambien las dependencias,
# así editar main.py no reinstala numpy y Pillow en cada build
COPY requirements.txt .
# --only-binary: solo wheels precompilados; si falta uno, el build falla aquí en vez de intentar
# compilar (y necesitar gcc) en silencio. Verificado: los 37 paquetes tienen wheel cp313 manylinux x86_64.
RUN pip install --only-binary=:all: -r requirements.txt

# ---------------------------------------------------------------- etapa 2: runtime
FROM python:3.13-slim AS runtime
```

- `python:3.13-slim` es un índice multi-plataforma: el mismo `FROM` baja la variante arm64 en tu Mac y la amd64 con `--platform linux/amd64` o en CI.
- El comentario dice **x86_64**: la verificación de wheels es por arquitectura. Un build nativo en tu Mac busca wheels `manylinux…aarch64`; si alguno no existe, `--only-binary` lo hace fallar justo en este paso.
- Con `--platform linux/amd64` en tu Mac, este `RUN` corre emulado. Como solo instala wheels, tarda más pero termina; compilar código nativo bajo emulación es lo que se vuelve lentísimo.
- El venv no es portable entre arquitecturas: trae binarios (`.so`) de numpy y Pillow. Por eso el `.dockerignore` de C01 deja fuera `.venv/`: un venv de macOS no sirve en Linux.
- En CI va el comando del curso: `docker buildx build --platform linux/amd64 -t $REG/classifier-gateway:$SHA --push gateway/`. Se construye una vez, para la arquitectura de los nodos.

CÓMO RECORDARLA

la imagen es de la CPU donde la construiste, salvo que pidas otra con `--platform`. Tu Mac habla arm64, el servidor amd64, y CI construye para el servidor.

TRAMPA

que funcione en tu Mac no prueba nada. Docker Desktop corre amd64 emulado avisando solo con un `WARNING`, y una imagen arm64 corre perfecta en tu Mac y muere en el servidor. Y medir latencia con una imagen amd64 en tu Mac mide al emulador, no a tu código.

En el curso: M5 · Un Dockerfile de ML, decisión por decisión · M5 · Registries y tags inmutables · Docker y kubectl · Docker: construir, correr, depurar · M5 · Por qué contenedores (y qué preguntan de verdad)

11 Registries y tags inmutables

¿Cómo nombras, subes y bajas una imagen para que lo que corre sea siempre lo que probaste?

SINTAXIS CON VARIABLES

registry / *namespace* / *repo* : *tag* **@sha256:** *digest*

La referencia completa. Todo es opcional salvo *repo* ; con digest, el tag queda solo como nombre legible.

aws ecr get-login-password --region *región* **| docker login --username AWS --password-stdin** *registry*

Login a ECR: token de 12 horas, usuario `AWS` . En GHCR: `echo $CR_PAT | docker login ghcr.io -u` *usuario* `--password-stdin` .

docker push *registry* / *repo* : *git_sha*

Termina con *git_sha* `: digest: sha256:… size: …` . Un tag por commit: nunca se reusa.

docker run --rm *registry* / *repo* **@sha256:** *digest*

Por digest corre exactamente ese contenido, hoy y dentro de un año: nunca trae actualizaciones por su cuenta.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
registry	host del registry	`123456789012.dkr.ecr.us-east-1.amazonaws.com`
namespace	usuario u organización (opcional)	`ninguno en el ECR del C01; tu usuario u org en GHCR`
repo	repositorio	`classifier-gateway`
tag	nombre legible que se puede mover	`3f9c2ab`
digest	64 caracteres hex del SHA-256 del manifest	`9b1c… (de docker push o de imagetools inspect)`
región	región de AWS del registry	`us-east-1`
usuario	tu usuario de GitHub	`la cuenta que creó el token`
git_sha	commit corto	`3f9c2ab = $(git rev-parse --short=7 HEAD)`

FÓRMULAS

$$\text{python} \equiv \text{docker.io/library/python:latest}$$

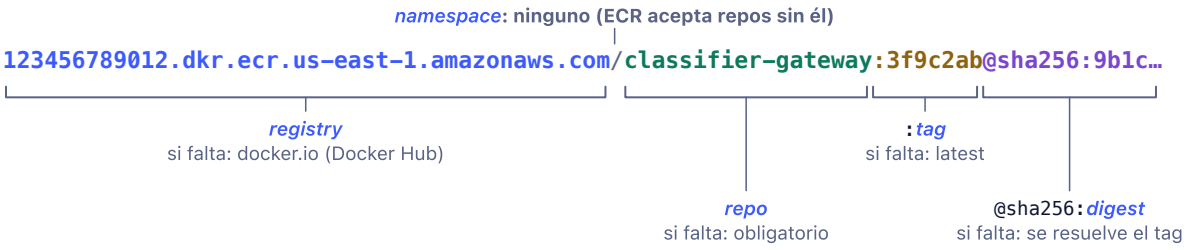
$$\text{digest} = \text{sha256:SHA256(bytes del manifest)}$$

$$\text{tag} \mapsto \text{digest (se puede mover)} \qquad \text{digest} \mapsto \text{contenido (fijo)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
registry	Host donde vive la imagen; sin host en el nombre, Docker usa docker.io.	`123456789012.dkr.ecr.us-east-1.amazonaws.com`
repo	Ruta del repositorio en minúsculas; con mayúsculas falla con invalid reference format: repository name (…) must be lowercase.	`classifier-gateway`
:tag	Nombre legible que apunta a un digest; si falta, latest.	`:3f9c2ab`
@sha256:	Hash del manifest: identifica el contenido y no se puede mover.	`@sha256:9b1c…`
latest	Solo el tag por defecto: no significa la versión más nueva.	`docker pull python baja python:latest`
docker login --password-stdin	Guarda la credencial (en Mac, en el llavero) sin dejar el token en el historial del shell.	`… | docker login --username AWS --password-stdin $REG`
token de ECR	Lo da aws ecr get-login-password y vale 12 horas.	`cada job de CI vuelve a hacer login`
GHCR	ghcr.io/NAMESPACE/IMAGEN; PAT classic con write:packages, o GITHUB_TOKEN en Actions.	`echo $CR_PAT | docker login ghcr.io -u USUARIO --password-stdin`
docker push	Sube las capas que faltan; cada tag termina con tag: digest: sha256:… size: ….	`Layer already exists = ya estaba`
docker pull …@sha256:	Baja exactamente ese contenido; nunca trae parches por su cuenta.	`docker pull $REG/classifier-gateway@sha256:9b1c…`

PIEZA	QUÉ HACE	EJEMPLO
docker run --pull	missing (default): si el tag ya está local, no consulta al registry; always lo fuerza.	`docker run --pull=always …:latest`
tags inmutables	El registry (ECR, Artifact Registry, Docker Hub) rechaza volver a subir un tag; en ECR, docker push falla con tag invalid: … cannot be overwritten because the repository is immutable.	`aws ecr put-image-tag-mutability --repository-name classifier-gateway --image-tag-mutability IMMUTABLE`
RepoDigests	repo@sha256 de la imagen local; con el image store de containerd aparece aunque nunca la hayas subido.	`docker image inspect --format '{{index .RepoDigests 0}}' IMAGEN`
límite de Docker Hub	Sin login: 100 pulls cada 6 horas por IP (o /64 de IPv6); cuenta Personal: 200.	`un clúster detrás de un NAT comparte la IP`



PARTE	EN EL C01	SI FALTA	REGLA
registry	123456789012.dkr.ecr.us-east-1.amazonaws.com	docker.io (Docker Hub)	Host DNS o IP con puerto opcional; Docker lo reconoce si tiene un punto, dos puntos o es localhost
namespace	ninguno (ECR acepta repos sin él)	library, solo en Docker Hub	Usuario u organización, en minúsculas
repo	classifier-gateway	obligatorio	Minúsculas y dígitos, con separadores . _ -
: *tag*	:3f9c2ab	latest	Hasta 128 caracteres: letras, dígitos, _ . -; no empieza con . ni -
@sha256: *digest*	@sha256:9b1c… (64 hex)	se resuelve el tag	SHA-256 del manifest; si está, manda sobre el tag

El nombre, sin el tag, admite hasta 255 caracteres. lab/app sin punto ni dos puntos en el primer componente va a docker.io/lab/app.

Las partes de una referencia de imagen, con los valores del C01 y lo que Docker asume cuando falta cada una.

EJEMPLO REAL · RELEASE DEL GATEWAY: BUILD UNA VEZ, DESPLEGAR POR DIGEST

BASH · ZSH — adaptado de release.sh (js/content/m05-contenedores.js, sección 8) y reference/c01/k8s/gateway.yaml

```
REG=123456789012.dkr.ecr.us-east-1.amazonaws.com
SHA=$(git rev-parse --short=7 HEAD)

# Login: el token de ECR vale 12 h (GHCR: echo $CR_PAT | docker login ghcr.io -u USUARIO --password-stdin)
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin $REG

# Build una vez, para la arquitectura de los nodos, con el commit como tag; --push sube al terminar
docker buildx build --platform linux/amd64 -t $REG/classifier-gateway:$SHA --push gateway/

# El digest es la identidad: es lo que se firma, se escanea y se despliega
docker buildx imagetools inspect $REG/classifier-gateway:$SHA     # línea Digest: sha256:…

# Desplegar por digest (k8s/gateway.yaml del C01 usa el tag :3f9c2ab)
kubectl -n classifier set image deployment/gateway gateway=$REG/classifier-gateway@sha256:9b1c…
```

- El tag es el commit (`3f9c2ab`): de un pod llegas al código exacto, y en un repo con tags inmutables nadie lo puede reusar.
- `--platform linux/amd64` y `--push` : CI construye una vez, para los nodos, y sube en el mismo paso.
- `docker buildx imagetools inspect` lee el índice en el registry sin bajar la imagen: la línea `Digest:` es lo que firmas y despliegas.
- `k8s/gateway.yaml` del C01 referencia el tag; desplegar por `@sha256:` te blinda contra un tag reescrito en un repo mutable.
- El rollback es volver al digest anterior: la lifecycle policy del registry tiene que conservar las últimas releases.

CÓMO RECORDARLA

El tag es un nombre; el digest es el contenido. Etiqueta con el commit, despliega por `@sha256:` y deja `latest` para humanos.

TRAMPA

`latest` no es "la última versión": es el tag que se usa cuando no pones ninguno. `docker run` no lo vuelve a bajar si ya lo tienes (`--pull=missing`), Kubernetes lo consulta en cada arranque (`imagePullPolicy: Always`) y no hay rollback: ya apunta a la versión rota.

En el curso: M5 · Registries y tags inmutables · M5 · Imagen, contenedor y capas · M11 · Rollback: criterios automáticos y qué guardar para volver · M15 · Supply chain: imágenes, dependencias y pesos verificados

12 Diagnóstico: las 10 fallas típicas

¿Qué ves, con qué comando lo confirmas y qué lo arregla?

SINTAXIS CON VARIABLES

`docker ps -a --filter name=` *nombre*

¿Sigue vivo? STATUS dice `Up` o `Exited (código)`, y hace cuánto.

`docker logs --tail` *n* *nombre*

Lo último que dijo tu proceso. Funciona con el contenedor parado: por eso no lo borres antes de leerlo.

`docker inspect --format '{{.State.ExitCode}} {{.State.OOMKilled}} {{.State.Error}}'` *nombre*

Lo que el log no dice: el código, si lo mató el OOM killer y el error del runtime.

`docker run --rm -it --entrypoint sh` *imagen* **`:`** *tag*

La imagen sin su CMD: para mirar archivos, `PATH` y usuario cuando el contenedor muere al arrancar.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre del contenedor; en Compose es proyecto-servicio-índice	`gw`, o `classifier-c01-triton-1`
n	cuántas líneas finales del log	`50`
imagen : *tag*	la imagen que quieres mirar por dentro	`classifier-gateway:3f9c2ab`
exit code	lo que muestran docker ps -a y docker inspect	`137`
$n_{\text{señal}}$	número de la señal que terminó el proceso	`2 = SIGINT (Ctrl+C); 9 = SIGKILL (OOM, o docker stop agotó su plazo: 10 s por defecto); 15 = SIGTERM`

FÓRMULAS

$$\text{exit code} = 128 + n_{\text{señal}}$$

$137 = 128 + 9$ (SIGKILL) $\qquad 143 = 128 + 15$ (SIGTERM)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`docker ps -a`	Todos los contenedores, también los muertos, con STATUS y exit code.	`Exited (137) 2 minutes ago`
`docker logs`	stdout y stderr del contenedor, esté corriendo o parado.	`docker logs --tail 50 gw`
`docker inspect --format '{{.State…}}'`	Estado exacto: ExitCode, OOMKilled, Error y Health.	`137 true`
`docker events`	Eventos del daemon en vivo: die, oom, kill, health_status.	`docker events --filter container=gw`
`docker stats`	CPU y memoria de cada contenedor contra su límite, en vivo.	`MEM USAGE / LIMIT`
`docker port`	Qué puertos del host apuntan al contenedor.	`8080/tcp -> 0.0.0.0:8080`
`docker exec -it` *nombre* **`sh`**	Proceso nuevo dentro de un contenedor que corre: env, archivos, id.	`docker exec gw id`
`docker run --entrypoint sh`	La imagen sin su CMD, para cuando el contenedor muere al arrancar.	`docker run --rm -it --entrypoint sh IMAGEN`
`docker build --progress=plain`	Log completo del build: qué pasos salieron CACHED y cuál se rehízo.	`#7 CACHED`
`docker history`	Capas con su tamaño y la instrucción que las creó.	`la capa de 2 GB y su COPY`
`docker system df`	Espacio de imágenes, contenedores, volúmenes y cache de build.	`Build Cache 12 GB`
`docker network inspect`	Qué contenedores están en esa red y con qué IP.	`docker network inspect classifier-c01_default`
`docker compose ps -a · logs`	Estado y health de cada servicio; logs con el nombre del servicio.	`triton (health: starting)`

FALLA	SÍNTOMA	CONFÍRMALO CON	CAUSA TÍPICA	ARREGLO
Sale en el acto	docker ps no lo muestra; docker ps -a dice Exited (0), (1) o (255) a los pocos segundos, o Created si ni llegó a arrancar (comando no encontrado: 127 en inspect)	`docker ps -a` · `docker logs NOMBRE` · `docker inspect -f '{{.State.ExitCode}}' NOMBRE`	El CMD terminó (un script, un python sin servidor) o la app falló al arrancar: variable faltante, archivo, módulo o comando no encontrado	Un proceso en primer plano que no termine (servidor en forma exec); lee el traceback; entra con docker run --rm -it --entrypoint sh IMAGEN
Exit 137 (OOM)	Exited (137) sin traceback; en un RUN del build, Killed	`docker inspect -f '{{.State.OOMKilled}} {{.State.ExitCode}}' NOMBRE` → `true 137` · `docker stats`	Pico de RAM sobre --memory, o sobre la memoria de la VM de Docker Desktop: pesos cargados a RAM, batch grande, workers que duplican el modelo	Mide el pico y fija --memory con margen; menos workers o batch. Con OOMKilled false, lo más probable es un SIGKILL externo (docker stop agotó su plazo, 10 s por defecto); si nadie hizo stop ni kill, sospecha memoria igual: el evento OOM a veces se pierde
exec format error	Muere al instante con exec /ruta: exec format error en el log y exit 255	`docker image inspect --format '{{.Os}}/{{.Architecture}}' IMAGEN` contra `uname -m` del servidor · `head -1` del script	Imagen arm64 (construida en tu Mac) en un servidor amd64, o un script de entrypoint sin #! en la primera línea; si el #! termina en CRLF, el error es no such file or directory	--platform linux/amd64 (mejor: build en CI); #!/bin/sh como primera línea y finales de línea LF
Puerto no accesible	curl localhost:8080 desde tu Mac falla (Connection refused, Empty reply o se cuelga) aunque la app responde adentro	`docker port NOMBRE` · `docker logs NOMBRE` (¿escucha en `127.0.0.1`?)	Faltó -p; -p al revés (es HOST:CONTENEDOR); la app escucha en 127.0.0.1 dentro del contenedor, el default de uvicorn y Flask	-p 8080:8080 y --host 0.0.0.0, como el CMD del gateway de C01
Connection refused entre contenedores	El gateway de C01 responde 503 Triton no disponible (status 503 en su log); otro cliente dice [Errno 111] Connection refused o no resuelve el nombre	`docker compose ps` (¿healthy?) · `docker network inspect RED` · `docker exec NOMBRE env | grep URL`	Usa localhost (es el propio contenedor); el destino aún no escucha (Triton abre puertos tras cargar el modelo); usa el puerto publicado en vez del interno; redes distintas	Nombre del servicio y puerto del contenedor (TRITON_URL=triton:8000 en C01); depends_on con condition: service_healthy; reintentos
Permission denied en un volumen	PermissionError: [Errno 13] Permission denied: '/data/...' o touch: /data/x: Permission denied	`docker exec NOMBRE id` · `docker exec NOMBRE ls -ldn /data`	La imagen corre sin root (USER 10001, como el gateway de C01) y el volumen es de root, o el bind mount trae el UID del host (en Linux)	Crea el directorio en la imagen con chown 10001:10001 (un volumen nombrado nuevo copia ese dueño) o usa --user con el UID dueño; nunca chmod 777 ni root
Cache de build que no se usa	Cambias una línea de código y el build reinstala todas las dependencias	`docker build --progress=plain .` (qué pasos dicen CACHED y cuál fue el primero en rehacerse)	COPY . . antes de pip install; .dockerignore sin .git ni datos; arriba, un ARG que cambia en cada build; en CI, un runner sin cache	requirements.txt → pip install → código, como C01; .dockerignore estricto; los ARG que cambian, abajo; en CI, --cache-from y --cache-to
Imagen enorme	Pesa GB para un servicio CPU, los pulls tardan minutos, o aparece no space left on device	`docker image ls IMAGEN` · `docker history --no-trunc IMAGEN` · `docker system df`	torch con CUDA en un servicio CPU (varios GB de librerías nvidia-*), base completa o -devel, cache de pip, un checkpoint copiado, borrar en un RUN posterior	Base slim y multi-stage; wheel cpu; PIP_NO_CACHE_DIR=1; .dockerignore; limpiar en el mismo RUN; pesos grandes fuera de la imagen

FALLA	SÍNTOMA	CONFÍRMALO CON	CAUSA TÍPICA	ARREGLO
CUDA no disponible	torch.cuda.is_available() da False y el servicio va lento; o could not select device driver	`nvidia-smi` en el host · `docker run --rm --gpus all ubuntu nvidia-smi` · `torch.version.cuda` en tu imagen	Faltó --gpus o la reserva de Compose; host sin toolkit (o tu Mac); driver más viejo que el CUDA de la imagen; wheel cpu	Pide la GPU; instala el toolkit; elige el CUDA de la imagen según el driver (cu126 con R535); falla al arrancar si no hay GPU
Funciona en local, no en CI	El build o los tests pasan en tu Mac y fallan en el runner; a veces un COPY con not found	`Log del runner con --progress=plain` · `uname -m` en ambos · `git status --ignored` · `docker build --no-cache` en local	Otra arquitectura (wheels y binarios distintos); archivos que no están en git; tu cache esconde un paso roto; credenciales o límite de pulls en el runner	Reproduce con --platform linux/amd64 --no-cache; fija versiones y digests; commitea o genera lo que falta; secretos del runner, nunca en la imagen

Las 10 fallas, más o menos en el orden en que aparecen. Cada fila: síntoma, cómo confirmarlo, causa típica y arreglo.

EJEMPLO REAL · C01: EL GATEWAY NO ARRANCA PORQUE TRITON NUNCA QUEDA HEALTHY

BASH · ZSH **reference/c01/docker-compose.yml (healthcheck y depends_on) + comandos del curso**

```
cd reference/c01
docker compose up -d --build
# dependency failed to start: container classifier-c01-triton-1 is unhealthy

docker compose ps -a                 # triton: unhealthy o Exited; gateway: Created, nunca arrancó
docker compose logs --tail 50 triton  # ¿cargó el modelo? ¿ve la GPU?
docker inspect --format '{{json .State.Health}}' classifier-c01-triton-1   # salidas del healthcheck
docker inspect --format '{{.State.ExitCode}} {{.State.OOMKilled}}' classifier-c01-triton-1
docker compose exec triton nvidia-smi # solo si sigue corriendo: ¿hay GPU adentro?
```

- Decide el healthcheck de Triton: `curl -fsS …/v2/health/ready` cada 5 s, 3 reintentos y `start_period: 60s`. El gateway espera con `condition: service_healthy`.
- `.State.Health` guarda las últimas salidas del chequeo: un error de conexión de curl mientras Triton carga (abre sus puertos al terminar) o un fallo si el modelo no quedó listo con `--strict-readiness`.
- Si el log de Triton dice que no ve la GPU o que el engine no carga, el arreglo está en el host o en `build_engine.sh`, no en el compose.
- Exit 137 con `OOMKilled true` es RAM, no VRAM: mira `docker stats` mientras carga el engine.
- Si la carga es lenta pero sana, sube `start_period` en vez de quitar la condición: sin ella, el gateway arranca y responde 503.

CÓMO RECORDARLA

ps -a → logs → inspect, en ese orden. El STATUS dice si murió y con qué código; los logs, por qué; `inspect`, lo que los logs no cuentan (OOMKilled, Health, Error). Recién después cambias algo.

TRAMPA

borrar el contenedor (o haberlo corrido con `--rm`) antes de leer su log y su exit code: se va la evidencia. Y no arregles a ciegas subiendo memoria o corriendo como root: un 137 con `OOMKilled false` suele ser un stop que agotó el plazo (aunque, si nadie hizo stop ni kill, puede ser memoria: el evento OOM a veces se pierde), y root tapa un problema de permisos que reaparece en producción.

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Docker: construir, correr, depurar · M5 · Imagen, contenedor y capas · M5 · Reducir tamaño (y por qué importa para el cold start) · M5 · docker compose para desarrollo local · Docker y kubectl · docker compose

13 Errores de un vistazo

Lees el mensaje: ¿qué significa, qué corres primero y dónde está la explicación?

SINTAXIS CON VARIABLES

`docker: Error response from daemon:` *mensaje*

Tu proceso nunca corrió: lo rechazó el daemon, o runc al crear el contenedor. `docker run` sale con 125; con 126 si el comando no es ejecutable y con 127 si no existe.

`ERROR: failed to build: failed to solve:` *paso* **`:`** *causa*

Falló el build. Lo útil está al final o en las líneas de arriba del paso que falló: míralas con `--progress=plain`.

`exec` *ruta* **`:`** *errno*

El contenedor ya había arrancado y el kernel rechazó el exec de tu entrypoint: runc lo escribe en el log y sale con 255. El errno dice por qué: `exec format error` o `no such file or directory`.

`docker logs` *nombre*

Todo lo demás lo dijo tu proceso: un Traceback, `Connection refused`, `Killed`. Ahí no hay prefijo de Docker.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
mensaje	el error que devolvió el daemon	`could not select device driver "" with capabilities: [[gpu]]`
paso	el paso del build que falló	`process "/bin/sh -c pip install --only-binary=:all: -r requirements.txt" did not complete successfully`
causa	la razón concreta, al final de la cadena	`exit code: 1`
ruta	el ejecutable o script que runc intentó correr	`/entrypoint.sh`
errno	el error del kernel al ejecutarlo	`exec format error`
nombre	el contenedor cuyo proceso habló	`gw (el gateway de C01)`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`exit 0`	Terminó bien… y por eso el contenedor se fue.	`un CMD que no se queda sirviendo`
`exit 1 (u otro)`	Lo devolvió tu proceso; 1 es el error genérico. En Python, casi siempre hay un Traceback en docker logs.	`ModuleNotFoundError`
`exit 125`	Falló Docker: flag inválido, imagen que no existe o pull denegado, error del daemon. El contenedor nunca corrió.	`unknown flag: --gpu`
`exit 126`	El comando existe pero no se puede ejecutar (sin permiso, o es un directorio). runc lo detecta al crear el contenedor.	`docker run busybox /etc`
`exit 127`	El comando no existe en la imagen o no está en el PATH. runc lo detecta al crear el contenedor.	`executable file not found in $PATH`
`exit 130`	128 + 2, SIGINT: lo cortaste con Ctrl+C.	`Ctrl+C en docker run -it`
`exit 137`	128 + 9, SIGKILL: OOM si OOMKilled es true; si es false, casi siempre docker stop agotó su plazo (10 s por defecto), pero no descarta OOM.	`OOMKilled true`
`exit 143`	128 + 15: un SIGTERM que tu proceso no manejó. Si lo maneja y sale limpio, verás 0.	`docker stop`
`exit 255`	Fuera de la tabla de Docker: el contenedor arrancó y el kernel rechazó el exec del entrypoint (runc ≥ 1.2).	`exec format error · CRLF en el shebang`
`docker: Error response from daemon:`	Prefijo de los rechazos del daemon.	`port is already allocated`
`ERROR: failed to build: failed to solve:`	Prefijo de los fallos de BuildKit.	`"/requirements.txt": not found`

LOS 26 MENSAJES

`Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?`

El socket existe pero nadie responde: el daemon está apagado o todavía arrancando (en tu Mac, Docker Desktop). Antes de Docker 29 salía también si el socket no existía.

PRIMERO: `docker context ls`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

`failed to connect to the docker API at unix:///Users/…/.docker/run/docker.sock; check if the path is correct and if the daemon is running`

La versión de Docker 29 cuando el socket no existe: Docker Desktop no está corriendo o el contexto apunta a otro lado.

PRIMERO: `docker context ls`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

`permission denied while trying to connect to the docker API at unix:///var/run/docker.sock`

Linux: el daemon corre, pero tu usuario no puede abrir el socket (antes decía …the Docker daemon socket at…).

PRIMERO: `ls -l /var/run/docker.sock && id -nG`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

`exec /usr/local/bin/python3: exec format error`

El kernel no reconoce el binario: imagen de otra arquitectura (arm64 de tu Mac en un servidor amd64) o script sin shebang. Exit 255.

PRIMERO:

```
docker image inspect --format '{{.Os}}/{{.Architecture}}' IMAGEN
```

→ Sección 10 · arm64 (tu Mac) vs amd64 (el servidor)

`exec: "uvicorn": executable file not found in $PATH`

El comando del CMD o ENTRYPOINT no está en el PATH de la imagen (no se instaló, o el PATH no incluye el venv). Exit 127.

PRIMERO:

```
docker run --rm --entrypoint sh IMAGEN -c 'echo $PATH; command -v uvicorn'
```

→ Sección 1 · Anatomía de un Dockerfile

`exec /entrypoint.sh: no such file or directory`

El archivo existe, pero falta su intérprete: el del shebang (CRLF en #!/bin/sh\r, o #!/bin/bash en Alpine) o el loader de un binario glibc en Alpine. Exit 255.

PRIMERO: `head -1 entrypoint.sh | od -c`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

`Bind for 0.0.0.0:8080 failed: port is already allocated · bind: address already in use`

Ese puerto del host ya está tomado: por otro contenedor (allocated) o por otro proceso de tu Mac (address already in use; AirPlay usa el 5000).

PRIMERO:

```
docker ps --filter publish=8080; lsof -nP -iTCP:8080 -sTCP:LISTEN
```

→ Sección 6 · Redes y puertos

`pull access denied for mi-imagen, repository does not exist or may require 'docker login'`

El registry no te da ese repositorio: nombre mal escrito, repo privado sin login, o una imagen que solo existe en tu máquina.

PRIMERO: `docker image ls mi-imagen`

→ Sección 11 · Registries y tags inmutables

`manifest for python:3.99-slim not found: manifest unknown · failed to resolve reference "docker.io/library/python:3.99-slim": …: not found`

El repositorio existe, pero ese tag no. El primer texto es del image store clásico; el segundo, del de containerd, el default de Docker Desktop y de Engine 29.

PRIMERO:
`docker buildx imagetools inspect python:3.99-slim` (si también falla, el tag no existe en el registry)

→ Sección 11 · Registries y tags inmutables

`no matching manifest for linux/arm64/v8 in the manifest list entries`

El índice de la imagen no trae variante para tu plataforma: en tu Mac, una imagen solo amd64; en el servidor, una solo arm64.

PRIMERO: `docker buildx imagetools inspect IMAGEN`

→ Sección 10 · arm64 (tu Mac) vs amd64 (el servidor)

`could not select device driver "" with capabilities: [[gpu]]`

Pediste GPU (--gpus o la reserva de Compose) y el daemon no tiene driver de GPU: falta el NVIDIA Container Toolkit, o estás en tu Mac. Desde Engine 29.3, con CDI activo, el mismo caso puede decir failed to discover GPU vendor from CDI.

PRIMERO: `nvidia-smi`

→ Sección 9 · GPU y CUDA en contenedores

`failed to compute cache key: failed to calculate checksum of ref …: "/requirements.txt": not found`

Un COPY pide algo que no está en el contexto: ruta fuera del contexto, mal escrita o excluida por .dockerignore.

PRIMERO:
`ls gateway/requirements.txt && cat gateway/.dockerignore`

→ Sección 2 · Capas, caché y .dockerignore

`failed to solve: process "/bin/sh -c pip install -r requirements.txt" did not complete successfully: exit code: 1`

Un RUN falló dentro del build. La causa está en las líneas de arriba: un wheel que no existe para tu arquitectura, sin red, algo que hay que compilar.

PRIMERO:
`docker build --progress=plain . 2>&1 | tail -n 40`

→ Sección 10 · arm64 (tu Mac) vs amd64 (el servidor)

`no space left on device`

Se llenó el disco donde Docker guarda imágenes, capas y cache de build (en tu Mac, el disco virtual de la VM de Docker Desktop).

PRIMERO: `docker system df`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

"OOMKilled": true · Exited (137) · Killed

El kernel mató el proceso por pasar el límite de memoria del contenedor o de la VM. Sin traceback; Killed lo imprime el shell padre.

PRIMERO:

```
docker inspect --format '{{.State.OOMKilled}}
{{.State.ExitCode}}' NOMBRE
```

→ Sección 4 · docker run y el ciclo de vida

the input device is not a TTY · cannot attach stdin to a TTY-enabled container because stdin is not a terminal

Usaste -t (de -it) sin una terminal: en CI, en un pipe o en cron. El primer texto es de Docker CLI hasta 29.3; el segundo, desde 29.4.

PRIMERO: `docker run -i IMAGEN …` (sin -t)

→ Sección 4 · docker run y el ciclo de vida

Conflict. The container name "/gw" is already in use by container "…"

Ya existe un contenedor con ese nombre, quizá parado: los nombres son únicos aunque el contenedor no corra.

PRIMERO: `docker ps -a --filter name=gw`

→ Sección 12 · Diagnóstico: las 10 fallas típicas

requested access to the resource is denied

Al hacer push: no tienes permiso de escritura en ese repositorio. Falta docker login, el tag no lleva tu registry o tu usuario (sin prefijo, va a docker.io), o tu cuenta no puede escribir ahí.

PRIMERO:

```
docker image ls --format '{{.Repository}}:{{.Tag}}' |
grep NOMBRE
```

→ Sección 11 · Registries y tags inmutables

dependency failed to start: container classifier-c01-triton-1 is unhealthy

Compose esperaba service_healthy y el healthcheck de la dependencia agotó sus reintentos, o el start_period no alcanzó para cargar el modelo.

PRIMERO:

```
docker inspect --format '{{json .State.Health}}'
classifier-c01-triton-1
```

→ Sección 8 · Docker Compose

Name or service not known · Temporary failure in name resolution

El primero: el resolver respondió que ese nombre no existe (typo, el servicio no está en tu red o el contenedor no corre). El segundo: no respondió o falló (sin red, --network none, DNS de la VM); según el DNS de arriba, también sale con un nombre inexistente. En los dos, revisa primero el nombre y la red.

PRIMERO:

```
docker network inspect RED --format '{{range
.Containers}}{{.Name}} {{end}}'
```

→ Sección 6 · Redes y puertos

[Errno 111] Connection refused (a localhost, desde un contenedor)

Dentro de un contenedor, localhost es el propio contenedor. Usa el nombre del servicio (triton:8000) o host.docker.internal para llegar a tu Mac.

PRIMERO: `docker exec NOMBRE env | grep URL`

→ Sección 6 · Redes y puertos

ModuleNotFoundError: No module named 'app'

Python no encuentra tu paquete: el WORKDIR o el COPY no coinciden con lo que asume el CMD, el código quedó fuera por .dockerignore, o falta la dependencia.

PRIMERO:

```
docker run --rm --entrypoint sh IMAGEN -c 'pwd && ls'
```

→ Sección 1 · Anatomía de un Dockerfile

CUDA driver version is insufficient for CUDA runtime version

El driver del host es más viejo que el CUDA de tu imagen, y no aplica ni minor version ni forward compatibility.

PRIMERO: `nvidia-smi`

→ Sección 9 · GPU y CUDA en contenedores

error getting credentials - err: exec: "docker-credential-desktop": executable file not found in $PATH

Tu ~/.docker/config.json pide un credential helper que no está en el PATH: típico con un PATH recortado o tras desinstalar Docker Desktop.

PRIMERO: `cat ~/.docker/config.json`

→ Sección 11 · Registries y tags inmutables

invalid reference format: repository name (library/MiImagen) must be lowercase

El nombre de imagen no es válido: mayúsculas, espacios, o un tag vacío (gateway:$TAG con TAG sin definir).

PRIMERO: `echo "gateway:$TAG"`

→ Sección 11 · Registries y tags inmutables

unknown flag: --gpu

Ese subcomando no conoce el flag: un typo (es --gpus), un flag de otro subcomando o de otra versión. Lo rechaza el CLI: exit 125.

PRIMERO: `docker run --help | grep -i gpu`

→ Sección 9 · GPU y CUDA en contenedores

EJEMPLO REAL · CÓMO LEER UN ERROR LARGO: UVICORN NO ESTÁ EN EL PATH

TEXTO — Forma típica del mensaje con Docker 29 y runc; el CMD es el de reference/c01/gateway/Dockerfile

```
# Una variante rota del gateway de C01: su etapa runtime perdió ENV PATH="/opt/venv/bin:$PATH"
$ docker run --rm classifier-gateway:roto
docker: Error response from daemon: failed to create task for container: failed to create shim task: OCI runtime create
failed: runc create failed: unable to start container process: error during container init: exec: "uvicorn": executable
file not found in $PATH: unknown

Run 'docker run --help' for more information
```

```
$ echo $?
127
```

- Léelo de derecha a izquierda: lo último es la causa (`exec: "uvicorn": executable file not found in $PATH`); lo de antes son las capas que lo reenvían: daemon, containerd, shim y runc. El texto del medio cambia entre versiones; el final, no.
- En el Dockerfile real de C01, uvicorn vive en `/opt/venv/bin` y la etapa runtime lo pone en el `PATH`. Si se perdiera esa línea, o el `COPY --from=builder /opt/venv /opt/venv`, este sería el error.
- Exit 127: Docker lo traduce a comando no encontrado. Falla al crear el contenedor, así que no hay nada en `docker logs`.
- Primer comando: `docker run --rm --entrypoint sh classifier-gateway:roto -c 'echo $PATH; ls /opt/venv/bin | head'`.

CÓMO RECORDARLA

de dónde viene el mensaje dice dónde mirar. `docker:` es el CLI o el daemon; `failed to solve`, el build; `exec …`, runc al arrancar; el resto lo dijo tu proceso y está en `docker logs`.

TRAMPA

buscar en Google la primera línea larga del error. La parte útil está al final (la causa concreta) o en las líneas de arriba (en el build). Y en Docker 29 algunos textos cambiaron: un tutorial viejo cita `Cannot connect to the Docker daemon` o `the input device is not a TTY`, y tú puedes ver otra redacción del mismo problema.

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Docker: construir, correr, depurar · Docker y kubectl · GPUs en Docker · Docker y kubectl · docker compose

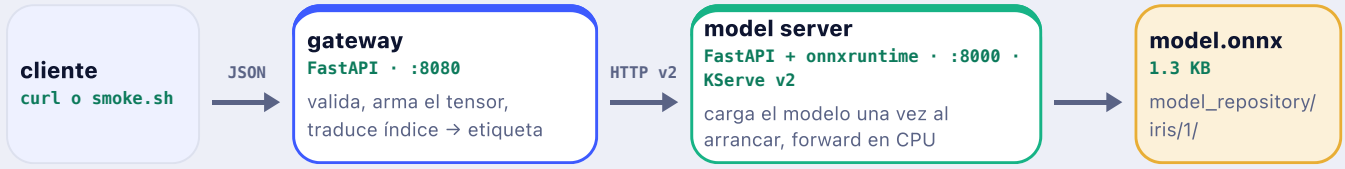
Lab Laboratorio completo: Iris con gateway y model server

Tu laboratorio de punta a punta: cada archivo, cada comando, lo que salió y lo que falta probar.

0 Qué es el proyecto

contexto

QUÉ ES Un clasificador de Iris (3 especies, 4 medidas en cm) servido con **dos servicios**, para practicar el ciclo completo de Docker: dependencias fijadas → Dockerfiles → build → caché → push → run → Compose → romperlo → AWS.



El model server habla el protocolo KServe v2, el mismo de Triton. Por eso el gateway no cambia si luego lo reemplazas por Triton: solo cambian `MODEL_SERVER_URL` y la imagen.

POR QUÉ NO TRITON EN LOCAL

La imagen arm64 pesa ~10 GB comprimida y ~30 GB en el pico del pull: llenó el disco de la Mac (trampa 12.1).

MÁQUINA

MacBook Air M4 (arm64), Docker Desktop. Todo corre local; nada en la nube todavía.

PUERTOS

El 8000 de tu Mac lo ocupa otro contenedor tuyo: por eso el model server se publica en el 9000.

LAS CARPETAS

iris-lab/

- `training/train.py` entrena el MLP y exporta ONNX — repo · 0f14740 — paso 1
- `model_repository/iris/1/model.onnx` 1.3 KB (se versiona: .gitignore tiene excepción) — repo · 0f14740 — paso 1
- **model_server/**
  - `server.py` — repo · 0f14740 — paso 2
  - `requirements.in / requirements.txt` — repo · 0f14740 — paso 3
  - `Dockerfile` variante A: modelo DENTRO de la imagen (contexto = iris-lab/) — prompt · probado en carpeta temporal — paso 6
  - `Dockerfile.dockerignore` — prompt · probado en carpeta temporal — paso 6
  - `Dockerfile.volume` variante B: modelo FUERA (volumen) (contexto = model_server/) — prompt · probado en carpeta temporal — paso 6
  - `.dockerignore` para la variante B — prompt · probado en carpeta temporal — paso 6
- **gateway/**
  - `app.py` — repo · 0f14740 — paso 2
  - `requirements.in / requirements.txt` — repo · 0f14740 — paso 3
  - `Dockerfile` — repo · sin commit — paso 4
  - `.dockerignore` — repo · sin commit — paso 4
- `scripts/smoke.sh` prueba end-to-end — repo · 0f14740 — paso 2

Además: `.gitignore` de iris-lab (paso 1), `compose.yaml` (borrador, paso 7) y `model_repository/iris/config.pbtxt` (referencia para Triton, paso 9; se borró del repo).

DE DÓNDE SALE CADA ARCHIVO

repo · 0f14740 En el repo: copiado byte por byte de iris-lab, igual al commit 0f14740 de la rama claude/classification-docker-setup-1528ea.

repo · sin commit En la carpeta iris-lab, todavía sin commit: copiado byte por byte.

prompt · probado en carpeta temporal Solo existe en CHEATSHEET_PROMPT.md: se probó en una carpeta temporal y falta crearlo en el repo. Es el bloque de código del prompt, tal cual.

prompt · borrador Solo existe en CHEATSHEET_PROMPT.md: borrador sin probar. Es el bloque de código del prompt, tal cual.

Copiados el 2026-10-29 de la rama `claude/classification-docker-setup-1528ea`; el build avisa si alguno ya no es igual a su fuente.

ESTADO DE CADA PASO

PASO	ESTADO
1 Entrenar/exportar	✅
2 Correr sin Docker	✅
3 requirements fijados	✅

PASO	ESTADO
4 Dockerfile gateway + build + caché	✅
5 Usuario no-root	✅
6 Dockerfile model server (A y B)	✅ probado en carpeta temporal falta crearlo y construirlo en el repo
7 Run/Compose	🟡 pendiente
8 Push	🟡 pendiente
9 Romperlo	🟡 parcial
10 AWS	🟡 plan, no probado

✅ **probado**: lo corriste y anotaste lo que salió. 🟡 borrador, pendiente o plan: está en tus notas, pero no se ha corrido. Las salidas que ves son solo las que anotaste; donde no hay, no se inventa ninguna.

TRAMPA · TE PASÓ

12.5 **Carpetas ocultas en macOS** `.claude` no se ve en Finder → `Cmd+Shift+.` o `Cmd+Shift+G` y pegar la ruta, o `code RUTA` desde la terminal.

Tu iris-lab vive en `Caso1/.claude/worktrees/classification-docker-setup-1528ea/iris-lab/` : sin ver las carpetas ocultas, en Finder no llegas.

1 Entrenar y exportar el modelo

✅ probado

QUÉ HACE Entrena un MLP chiquito (4 → 16 → 3) sobre Iris y lo exporta a ONNX en `model_repository/iris/1/model.onnx` , la carpeta que leerá el model server. Corre una vez, en tu Mac, fuera de cualquier imagen.

ARCHIVOS

PYTHON **training/train.py** repo · 0f14740 78 líneas

```python
"""Entrena un MLP chiquito sobre Iris y lo exporta a ONNX dentro del model repository de Triton.

Uso (desde iris-lab/):
    python training/train.py

Resultado: model_repository/iris/1/model.onnx
"""

from pathlib import Path

import numpy as np
import torch
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from torch import nn

SEED = 0
OUT = Path(__file__).resolve().parents[1] / "model_repository" / "iris" / "1" / "model.onnx"


class IrisNet(nn.Module):
    """La normalización vive DENTRO del modelo: el gateway manda features crudas.

    Así no hay dos copias de media/desviación (una en training y otra en serving)
    que se puedan desincronizar: anti training/serving skew.
    """

    def __init__(self, mean: np.ndarray, std: np.ndarray):
        super().__init__()
        self.register_buffer("mean", torch.tensor(mean, dtype=torch.float32))
        self.register_buffer("std", torch.tensor(std, dtype=torch.float32))
        self.net = nn.Sequential(nn.Linear(4, 16), nn.ReLU(), nn.Linear(16, 3))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = (x - self.mean) / self.std
        # Softmax adentro: Triton devuelve probabilidades listas, no logits.
        return torch.softmax(self.net(x), dim=1)
```

```python
def main() -> None:
    torch.manual_seed(SEED)
    X, y = load_iris(return_X_y=True)
    X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=SEED, stratify=y)

    model = IrisNet(X_tr.mean(axis=0), X_tr.std(axis=0))
    opt = torch.optim.Adam(model.parameters(), lr=0.05)
    loss_fn = nn.NLLLoss()  # el modelo ya devuelve softmax → NLL sobre log(probs)

    Xt = torch.tensor(X_tr, dtype=torch.float32)
    yt = torch.tensor(y_tr)
    for _ in range(200):
        opt.zero_grad()
        loss = loss_fn(torch.log(model(Xt) + 1e-9), yt)
        loss.backward()
        opt.step()

    model.eval()
    with torch.inference_mode():
        pred = model(torch.tensor(X_te, dtype=torch.float32)).argmax(dim=1).numpy()
    print(f"accuracy test: {(pred == y_te).mean():.3f}  (n={len(y_te)})")

    OUT.parent.mkdir(parents=True, exist_ok=True)
    torch.onnx.export(
        model,
        (torch.zeros(1, 4),),
        OUT,
        input_names=["input"],
        output_names=["probs"],
        # El batch es dinámico: Triton junta requests (dynamic batching) en un solo forward.
        dynamic_axes={"input": {0: "batch"}, "probs": {0: "batch"}},
        opset_version=17,
        dynamo=False,
    )
    print(f"exportado: {OUT}")


if __name__ == "__main__":
    main()
```

TEXTO **.gitignore** repo · 0f14740 5 líneas

```
.venv/
__pycache__/

# El modelo pesa 1.3 KB y el model server lo necesita: sí se versiona (la raíz ignora *.onnx).
!model_repository/**/*.onnx
```

El de `iris-lab/`, no el de la raíz: su excepción deja entrar al modelo en el commit (la trampa de este paso).

COMANDOS

DESDE `iris-lab`

```
uv run --no-project --python 3.13 --with torch --with scikit-learn --with onnx python training/train.py
```

SALIDA ESPERADA

- `accuracy test: 1.000 (n=30)`
- `exportado: .../model_repository/iris/1/model.onnx`

POR QUÉ

- `uv run --with …` arma un entorno temporal con `torch`, `scikit-learn` y `onnx` solo para esta corrida, y `--no-project` evita que uv use el `pyproject.toml` de Caso1 (iris-lab vive dentro de ese repo).
- `register_buffer("mean"/"std")`: la normalización vive **dentro** del modelo → anti training/serving skew; se exporta como constante al ONNX.
- Softmax dentro del modelo → el server devuelve probabilidades. Por eso la loss es `NLLLoss` sobre `torch.log(probs + 1e-9)` (con logits se usaría `CrossEntropyLoss`). Nota: lo más estable en proyectos serios es entrenar con logits y añadir softmax solo al exportar.
- `stratify=y` en `train_test_split`: conserva la proporción de clases.

- `torch.onnx.export(model, (torch.zeros(1, 4),), ...)`: el segundo argumento es un **input de ejemplo** (tracing), no la forma de la salida. `dynamic_axes` marca el batch como variable.
- `opset_version=17` es una elección conservadora (versión del «diccionario» de operaciones ONNX; debe soportarla el runtime). `dynamo=False` fuerza el exportador legacy (el nuevo usa `dynamic_shapes`).
- Las dependencias de entrenamiento (`torch`, `scikit-learn`, `onnx`) **no** van en ninguna imagen: el servidor solo necesita `onnxruntime`. Esa es una razón para exportar a ONNX.

TRAMPA · TE PASÓ

12.7 `.gitignore` **raíz ignora** `*.onnx` el modelo no entraba al commit → excepción en `iris-lab/.gitignore` (`!model_repository/**/*.onnx`).

2 Correr los dos servicios SIN Docker

✅ probado

QUÉ HACE Levanta el model server y el gateway como procesos normales en tu Mac, cada uno con su `.venv`, y los prueba de punta a punta con `smoke.sh`. Así, cuando algo falle dentro de Docker, ya sabes que el código sí funciona.

ARCHIVOS

PYTHON **model_server/server.py** repo · 0f14740 140 líneas

```python
"""Model server liviano: sirve un ONNX con onnxruntime y habla el protocolo KServe v2.

Es el mismo protocolo que Triton, así que el gateway no sabe (ni le importa) cuál de los dos
tiene atrás: cambiar a Triton después es cambiar la imagen en el compose.

Lee el modelo de: $MODEL_REPOSITORY/<MODEL_NAME>/<MODEL_VERSION>/model.onnx
"""

import math
import os
from contextlib import asynccontextmanager
from pathlib import Path

import numpy as np
import onnxruntime as ort
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

MODEL_REPOSITORY = Path(os.getenv("MODEL_REPOSITORY", "../model_repository"))
MODEL_NAME = os.getenv("MODEL_NAME", "iris")
MODEL_VERSION = os.getenv("MODEL_VERSION", "1")
MAX_BATCH_SIZE = int(os.getenv("MAX_BATCH_SIZE", "64"))
# 1 hilo por defecto: en un contenedor con límite de CPU, más hilos que cores solo pelean entre sí.
ORT_THREADS = int(os.getenv("ORT_THREADS", "1"))

# Tipos del protocolo v2 → numpy. Este modelo solo usa FP32.
V2_TO_NUMPY = {"FP32": np.float32}
ONNX_TO_V2 = {"tensor(float)": "FP32"}


class InferInput(BaseModel):
    name: str
    shape: list[int]
    datatype: str
    data: list[float]  # tensor aplanado (row-major), como lo manda el cliente


class RequestedOutput(BaseModel):
    name: str


class InferRequest(BaseModel):
    inputs: list[InferInput]
    outputs: list[RequestedOutput] | None = None  # None = devolver todas


def load_session() -> ort.InferenceSession:
    path = MODEL_REPOSITORY / MODEL_NAME / MODEL_VERSION / "model.onnx"
    opts = ort.SessionOptions()
    opts.intra_op_num_threads = ORT_THREADS
```

```
    return ort.InferenceSession(str(path), opts, providers=["CPUExecutionProvider"])


@asynccontextmanager
async def lifespan(app: FastAPI):
    # El modelo se carga UNA vez al arrancar, no por request.
    # Si el archivo no existe, el proceso muere aquí: fail fast, el orquestador lo ve en seguida.
    app.state.session = load_session()
    yield


app = FastAPI(title="iris-model-server", lifespan=lifespan)


def _check_model(name: str) -> ort.InferenceSession:
    if name != MODEL_NAME:
        raise HTTPException(404, f"modelo '{name}' no existe; este server sirve '{MODEL_NAME}'")
    return app.state.session


@app.get("/v2/health/live")
def live():
    return {"live": True}


@app.get("/v2/health/ready")
def ready():
    # Si llegamos aquí, lifespan ya cargó el modelo (si no, el proceso no estaría vivo).
    return {"ready": True}


@app.get("/v2/models/{name}/ready")
def model_ready(name: str):
    _check_model(name)
    return {"name": name, "ready": True}


@app.get("/v2/models/{name}")
def model_metadata(name: str):
    """Qué espera y qué devuelve el modelo: útil para depurar desde curl."""
    sess = _check_model(name)

    def describe(t):
        return {"name": t.name, "datatype": ONNX_TO_V2.get(t.type, t.type), "shape": [-1 if not isinstance(d, int)
          else d for d in t.shape]}

    return {
        "name": MODEL_NAME,
        "versions": [MODEL_VERSION],
        "platform": "onnxruntime_onnx",
        "inputs": [describe(t) for t in sess.get_inputs()],
        "outputs": [describe(t) for t in sess.get_outputs()],
    }


# `def` (no `async def`): onnxruntime bloquea mientras calcula; FastAPI corre los `def` en su
# threadpool, así el event loop sigue atendiendo /health mientras hay un forward en curso.
@app.post("/v2/models/{name}/infer")
def infer(name: str, req: InferRequest):
    sess = _check_model(name)
    expected = {t.name for t in sess.get_inputs()}

    feeds = {}
    for inp in req.inputs:
        if inp.name not in expected:
            raise HTTPException(400, f"input '{inp.name}' desconocido; el modelo espera {sorted(expected)}")
        if inp.datatype not in V2_TO_NUMPY:
            raise HTTPException(400, f"datatype '{inp.datatype}' no soportado; usa {list(V2_TO_NUMPY)}")
        if len(inp.data) != math.prod(inp.shape):
            raise HTTPException(400, f"shape {inp.shape} pide {math.prod(inp.shape)} valores y llegaron
              {len(inp.data)}")
        if inp.shape and inp.shape[0] > MAX_BATCH_SIZE:
            raise HTTPException(400, f"batch {inp.shape[0]} > max_batch_size {MAX_BATCH_SIZE}")
        feeds[inp.name] = np.asarray(inp.data, dtype=V2_TO_NUMPY[inp.datatype]).reshape(inp.shape)

    if missing := expected - feeds.keys():
        raise HTTPException(400, f"faltan inputs: {sorted(missing)}")
```

```
    out_names = [o.name for o in req.outputs] if req.outputs else [t.name for t in sess.get_outputs()]
    try:
        results = sess.run(out_names, feeds)
    except Exception as e:  # p. ej. dims incorrectas: onnxruntime explica qué esperaba
        raise HTTPException(400, f"onnxruntime: {e}")

    return {
        "model_name": MODEL_NAME,
        "model_version": MODEL_VERSION,
        "outputs": [
            {"name": n, "datatype": "FP32", "shape": list(r.shape), "data": r.ravel().tolist()}
            for n, r in zip(out_names, results)
        ],
    }
```

PYTHON **gateway/app.py** repo · 0f14740 117 líneas

```
"""Gateway: la puerta de entrada del sistema.

Cliente --JSON--> gateway (valida, arma el tensor) --HTTP v2--> model server (forward) --> gateway (etiqueta) -->
  cliente

El gateway NO carga el modelo: no necesita onnxruntime ni numpy. Eso lo hace el model server.
"""

import os
from contextlib import asynccontextmanager

import httpx
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

# Config por variables de entorno: la misma imagen corre en la laptop, en compose o en k8s.
MODEL_SERVER_URL = os.getenv("MODEL_SERVER_URL", "http://localhost:8000")
MODEL_NAME = os.getenv("MODEL_NAME", "iris")
MODEL_SERVER_TIMEOUT_S = float(os.getenv("MODEL_SERVER_TIMEOUT_S", "2.0"))

CLASSES = ["setosa", "versicolor", "virginica"]


class Flower(BaseModel):
    # Rangos físicos razonables en cm: rechazamos basura antes de gastar un forward.
    sepal_length: float = Field(gt=0, lt=20)
    sepal_width: float = Field(gt=0, lt=20)
    petal_length: float = Field(gt=0, lt=20)
    petal_width: float = Field(gt=0, lt=20)


class PredictRequest(BaseModel):
    flowers: list[Flower] = Field(min_length=1, max_length=64)  # 64 = MAX_BATCH_SIZE del model server


class Prediction(BaseModel):
    label: str
    score: float
    probabilities: dict[str, float]


class PredictResponse(BaseModel):
    model: str
    predictions: list[Prediction]


@asynccontextmanager
async def lifespan(app: FastAPI):
    # Un solo cliente HTTP para toda la vida del proceso: reutiliza conexiones (keep-alive).
    app.state.client = httpx.AsyncClient(base_url=MODEL_SERVER_URL, timeout=MODEL_SERVER_TIMEOUT_S)
    yield
    await app.state.client.aclose()


app = FastAPI(title="iris-gateway", lifespan=lifespan)


@app.get("/healthz")
```

```python
async def healthz():
    """Liveness: el proceso está vivo. No depende del model server (si no, k8s reiniciaría el gateway por culpa de
      otro)."""
    return {"status": "ok"}


@app.get("/readyz")
async def readyz():
    """Readiness: ¿puedo recibir tráfico? Solo si el model server tiene el modelo cargado."""
    try:
        r = await app.state.client.get(f"/v2/models/{MODEL_NAME}/ready")
    except httpx.HTTPError as e:
        raise HTTPException(503, f"model server no responde en {MODEL_SERVER_URL}: {type(e).__name__}")
    if r.status_code != 200:
        raise HTTPException(503, f"modelo '{MODEL_NAME}' no está listo en el model server")
    return {"status": "ready", "model_server": MODEL_SERVER_URL, "model": MODEL_NAME}


@app.post("/v1/predict", response_model=PredictResponse)
async def predict(req: PredictRequest):
    rows = [[f.sepal_length, f.sepal_width, f.petal_length, f.petal_width] for f in req.flowers]

    # Protocolo KServe v2 (el que hablan Triton y nuestro model server): tensor aplanado + shape + dtype.
    payload = {
        "inputs": [
            {
                "name": "input",
                "shape": [len(rows), 4],
                "datatype": "FP32",
                "data": [x for row in rows for x in row],
            }
        ],
        "outputs": [{"name": "probs"}],
    }

    try:
        r = await app.state.client.post(f"/v2/models/{MODEL_NAME}/infer", json=payload)
    except httpx.TimeoutException:
        raise HTTPException(504, "el model server tardó demasiado")
    except httpx.HTTPError as e:
        raise HTTPException(503, f"model server no disponible: {type(e).__name__}")
    if r.status_code != 200:
        # 502: el que falló fue el servicio de atrás, no el cliente.
        raise HTTPException(502, f"model server respondió {r.status_code}: {r.text[:200]}")

    out = r.json()["outputs"][0]
    n_classes = out["shape"][1]
    flat = out["data"]

    predictions = []
    for i in range(len(rows)):
        probs = flat[i * n_classes : (i + 1) * n_classes]
        best = max(range(n_classes), key=probs.__getitem__)
        predictions.append(
            Prediction(
                label=CLASSES[best],
                score=round(probs[best], 4),
                probabilities={c: round(p, 4) for c, p in zip(CLASSES, probs)},
            )
        )
    return PredictResponse(model=MODEL_NAME, predictions=predictions)
```

BASH **scripts/smoke.sh** repo · 0f14740 33 líneas

```bash
#!/usr/bin/env bash
# Prueba de humo end-to-end: model server directo y luego a través del gateway.
set -euo pipefail

MODEL_SERVER=${MODEL_SERVER:-http://localhost:9000}
GATEWAY=${GATEWAY:-http://localhost:8080}

echo "== Model server: ¿modelo listo?"
curl -sf -o /dev/null -w "HTTP %{http_code}\n" "$MODEL_SERVER/v2/models/iris/ready"

echo "== Model server: metadata (qué espera el modelo)"
curl -sf "$MODEL_SERVER/v2/models/iris"
echo

echo "== Model server: inferencia cruda (protocolo v2)"
curl -sf "$MODEL_SERVER/v2/models/iris/infer" -H 'Content-Type: application/json' -d '{
  "inputs": [{"name": "input", "shape": [1, 4], "datatype": "FP32", "data": [5.1, 3.5, 1.4, 0.2]}]
}'
echo

echo "== Gateway: readiness"
curl -sf "$GATEWAY/readyz"
echo

echo "== Gateway: predicción (3 flores en un request)"
curl -sf "$GATEWAY/v1/predict" -H 'Content-Type: application/json' -d '{
  "flowers": [
    {"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.2},
    {"sepal_length": 5.9, "sepal_width": 3.0, "petal_length": 4.2, "petal_width": 1.5},
    {"sepal_length": 6.7, "sepal_width": 3.0, "petal_length": 5.2, "petal_width": 2.3}
  ]
}'
echo
```

COMANDOS

Entornos locales (la primera vez, sin versiones fijadas; luego se reemplaza por el paso 3)

DESDE `iris-lab/model_server`

```
uv venv --python 3.13 && uv pip install fastapi "uvicorn[standard]" onnxruntime numpy
```

DESDE `iris-lab/gateway`

```
uv venv --python 3.13 && uv pip install fastapi "uvicorn[standard]" httpx
```

Arrancar (puerto 9000 porque el 8000 del host lo ocupa otro contenedor tuyo; cada server se queda corriendo en su terminal)

DESDE `iris-lab/model_server` · terminal 1

```
.venv/bin/uvicorn server:app --port 9000
```

DESDE `iris-lab/gateway` · terminal 2 · la variable va en la MISMA línea que el comando

```
MODEL_SERVER_URL=http://localhost:9000 .venv/bin/uvicorn app:app --port 8080
```

DESDE `iris-lab` · terminal 3

```
./scripts/smoke.sh
```

Comprobaciones manuales con curl

DESDE cualquier carpeta

```
curl -s -m 3 -w ' [%{http_code}]\n' localhost:8080/healthz
```

```
curl -s -m 3 -w ' [%{http_code}]\n' localhost:8080/readyz
```

```
curl -s -m 3 -w ' [%{http_code}]\n' localhost:9000/v2/health/ready
```

```
lsof -nP -iTCP:8080 -iTCP:9000 -sTCP:LISTEN
```

SALIDA ESPERADA

- `smoke.sh` : model server `HTTP 200` , la metadata, la inferencia cruda, el gateway `{"status":"ready",...}` y la predicción de 3 flores (setosa 1.0, versicolor 0.9996, virginica 1.0).
- Para los `curl` no anotaste salidas: el número entre corchetes es el código HTTP de cada respuesta.

POR QUÉ

- Cada servicio tiene su `.venv` , su alacena: así cada imagen llevará solo lo que usa (el gateway no necesita `onnxruntime` ni `numpy`).
- Pydantic: `Flower` / `PredictRequest` / `PredictResponse` validan entrada y salida (422 si falla; `response_model` define el contrato de salida).
- `lifespan` + `yield` : antes del `yield` = arranque (abrir `httpx.AsyncClient` , cargar el modelo); después = apagado (`aclose()`).
- `/healthz` (liveness, no depende de nadie) vs `/readyz` (readiness, pregunta al model server `GET /v2/models/iris/ready`).
- Protocolo v2: tensor aplanado + `shape` + `datatype` ; `"outputs": [{"name": "probs"}]` pide qué salida devolver (el nombre viene de `output_names` en el export).
- Decodificación en el gateway: `best = max(range(n_classes), key=probs.__getitem__)` es un argmax; `CLASSES` vive en el gateway (si cambia el orden en training, las etiquetas saldrían mal sin error).
- `server.py` usa `def` (no `async def`) en `/infer` porque onnxruntime bloquea; FastAPI lo corre en su threadpool.
- Config por variables de entorno: `MODEL_SERVER_URL` (default `http://localhost:8000`), `MODEL_NAME` , `MODEL_SERVER_TIMEOUT_S` , `MODEL_REPOSITORY` , `MODEL_VERSION` , `MAX_BATCH_SIZE` , `ORT_THREADS` .
- `curl -s` no muestra la barra de progreso, `-m 3` se rinde a los 3 s y `-w ' [%{http_code}]\n'` agrega el código HTTP al final. `lsof … -sTCP:LISTEN` dice quién escucha en esos puertos (`-n` y `-P` : IP y puerto en número, sin traducir a nombres).

TRAMPA · TE PASÓ

12.3 **La variable de entorno en una línea aparte** no se la pasa al programa.

12.2 **El puerto 8000 del host ocupado** por otro contenedor → el gateway «conectaba» ahí y daba `RemoteProtocolError` . Diagnóstico: `lsof -nP -iTCP:8000 -sTCP:LISTEN` y `docker ps` .

Sin `export` , `MODEL_SERVER_URL=...` sola en su línea es una variable del shell: el proceso que arrancas después no la ve y usa el default `localhost:8000` , donde contestaba tu otro contenedor. Las dos trampas te pasaron juntas (paso 9, experimento 1).

3 Dependencias fijadas: requirements.in → requirements.txt

✅ probado

QUÉ HACE Fija las versiones exactas, con un par de archivos por servicio (cada imagen lleva solo lo que usa). El `.in` es la lista del súper: lo que quieres. El `.txt` es el ticket: lo que te llevaste exactamente, con versión, y quién pidió cada cosa.

ARCHIVOS

TEXTO **gateway/requirements.in** repo · 0f14740 3 líneas

```
fastapi
uvicorn[standard]
httpx
```

TEXTO **model_server/requirements.in** repo · 0f14740 4 líneas

```
fastapi
uvicorn[standard]
onnxruntime
numpy
```

TEXTO **gateway/requirements.txt** repo · 0f14740 65 líneas

```
# This file was autogenerated by uv via the following
  command:
#    uv pip compile requirements.in -o requirements.txt
  --universal --python-version 3.13
annotated-doc==0.0.5
    # via fastapi
annotated-types==0.8.0
    # via pydantic
anyio==4.15.1
    # via
    #   httpx
    #   starlette
    #   watchfiles
certifi==2026.10.23
    # via
    #   httpcore
    #   httpx
click==8.5.0
    # via uvicorn
fastapi==0.162.0
```

```
    # via -r requirements.in
h11==0.16.0
    # via
    #   httpcore
    #   uvicorn
httpcore==1.0.9
    # via httpx
httptools==0.9.0
    # via uvicorn
httpx==0.28.1
    # via -r requirements.in
idna==3.20
    # via
    #   anyio
    #   httpx
opentelemetry-api==1.45.1
    # via fastapi
pydantic==2.14.1
    # via fastapi
pydantic-core==2.50.1
    # via pydantic
python-dotenv==1.2.4
    # via uvicorn
pyyaml==6.0.3
    # via uvicorn
starlette==1.10.0
    # via fastapi
typing-extensions==4.16.0
    # via
    #   anyio
    #   fastapi
    #   opentelemetry-api
    #   pydantic
    #   pydantic-core
    #   typing-inspection
typing-inspection==0.4.4
    # via
    #   fastapi
    #   pydantic
uvicorn==0.54.0
    # via -r requirements.in
uvloop==0.23.0 ; platform_python_implementation !=
  'PyPy' and sys_platform != 'cygwin' and sys_platform
  != 'win32'
    # via uvicorn
watchfiles==1.3.0
    # via uvicorn
websockets==17.3
    # via uvicorn
```

TEXTO **model_server/requirements.txt** repo · 0f14740 64 líneas

```
# This file was autogenerated by uv via the following
  command:
#    uv pip compile requirements.in -o requirements.txt
  --universal --python-version 3.13
annotated-doc==0.0.5
    # via fastapi
annotated-types==0.8.0
    # via pydantic
anyio==4.15.1
    # via
    #   starlette
    #   watchfiles
click==8.5.0
    # via uvicorn
fastapi==0.162.0
    # via -r requirements.in
flatbuffers==25.12.19
    # via onnxruntime
h11==0.16.0
    # via uvicorn
httptools==0.9.0
    # via uvicorn
idna==3.20
    # via anyio
numpy==2.5.4
    # via
    #   -r requirements.in
    #   onnxruntime
onnxruntime==1.31.1
    # via -r requirements.in
opentelemetry-api==1.45.1
    # via fastapi
packaging==26.3
    # via onnxruntime
protobuf==7.36.2
    # via onnxruntime
pydantic==2.14.1
    # via fastapi
pydantic-core==2.50.1
    # via pydantic
python-dotenv==1.2.4
    # via uvicorn
pyyaml==6.0.3
    # via uvicorn
starlette==1.10.0
    # via fastapi
typing-extensions==4.16.0
    # via
    #   anyio
    #   fastapi
    #   opentelemetry-api
    #   pydantic
    #   pydantic-core
    #   typing-inspection
typing-inspection==0.4.4
    # via
    #   fastapi
    #   pydantic
uvicorn==0.54.0
    # via -r requirements.in
uvloop==0.23.0 ; platform_python_implementation !=
  'PyPy' and sys_platform != 'cygwin' and sys_platform
  != 'win32'
    # via uvicorn
watchfiles==1.3.0
    # via uvicorn
websockets==17.3
    # via uvicorn
```

COMANDOS

Generar el `.txt`

DESDE `iris-lab/gateway` · y otra vez desde iris-lab/model_server

```
uv pip compile requirements.in -o requirements.txt --universal --python-version 3.13
```

Verificar que el `.txt` reproduce el entorno (detén los servidores primero)

DESDE `iris-lab/gateway` · y otra vez desde iris-lab/model_server

```
rm -rf .venv && uv venv --python 3.13 && uv pip sync requirements.txt && uv pip freeze | wc -l
```

→ 23 (gateway) y 25 (model_server)

SALIDA ESPERADA

- Resultado: gateway 23 paquetes, model server 25. Versiones obtenidas: fastapi 0.162.0, uvicorn 0.54.0, httpx 0.28.1, onnxruntime 1.31.1, numpy 2.5.4.
- El conteo con `wc -l`: 23 en gateway y 25 en model_server.

POR QUÉ

- `--universal`: el resultado sirve en Mac **y** Linux (el contenedor es Linux). `--python-version 3.13`: misma versión que `python:3.13-slim`.
- Mira la línea de `uvloop` en los dos `.txt`: lleva un marcador (`; platform_python_implementation != 'PyPy' and sys_platform != 'cygwin' and sys_platform != 'win32'`). Así funciona `--universal`: un solo ticket, con condiciones por plataforma.
- El `.txt` trae comentarios `# via` que dicen quién pidió cada paquete.
- Se commitean los dos archivos (`.in` y `.txt`); el `.txt` no se edita a mano.
- Actualizar un paquete: `uv pip compile ... --upgrade-package fastapi`.
- `sync` deja el entorno **exactamente** igual al archivo (instala lo que falta y quita lo que sobra); `install` solo agrega. Es vaciar la alacena y volver a llenarla solo con el ticket.
- `uv pip freeze | wc -l`: `freeze` lista paquetes (uno por línea), el pipe `|` se los pasa a `wc -l`, que cuenta líneas → número de paquetes instalados.

TRAMPA · TE PASÓ

12.8 **Pegar una plantilla sobre** `pyproject.toml` (otro proyecto, Caso1) desincronizó el lock: el lock se deriva del archivo de dependencias, nunca al revés.

Aquí pasa lo mismo: el ticket sale de la lista, nunca al revés. Si cambias el `.in`, vuelves a compilar el `.txt`.

4 Gateway: Dockerfile, build, caché

✅ probado

QUÉ HACE Empaqueta el gateway en una imagen: Python base, las dependencias del ticket, el código y un usuario sin privilegios. Después mide la imagen y prueba cuándo Docker reusa la caché y cuándo no.

ARCHIVOS

DOCKERFILE **gateway/Dockerfile** repo · sin commit 16 líneas

```
FROM python:3.13-slim

ENV PYTHONUNBUFFERED=1

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

RUN useradd --create-home appuser
USER appuser

EXPOSE 8080
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8080"]
```

TEXTO **gateway/.dockerignore** repo · sin commit 3 líneas

```
.venv
__pycache__
*.pyc
```

Guárdalo con contenido: una vez quedó vacío por no guardar y el build habría mandado el `.venv`.

COMANDOS

Build (el `.` final es el build context)

DESDE `iris-lab/gateway`

```
docker build --progress=plain -t iris-gateway:0.1 .
```

```
docker images iris-gateway
```

```
docker history iris-gateway:0.1
```

Experimentos de caché (todos ✅)

DESDE `iris-lab/gateway`

```
docker build --no-cache --progress=plain -t iris-gateway:0.1 .
```

→ todo se rehace

```
docker build --progress=plain -t iris-gateway:0.1 .
```

→ todo CACHED (<1 s); mismo manifest sha256

Después de los experimentos A y B (abajo)

DESDE `iris-lab` · las rutas empiezan en gateway/

```
git restore gateway/app.py gateway/requirements.txt
```

DESDE **cualquier carpeta**

```
docker rmi iris-gateway:0.2 iris-gateway:0.3
```

SALIDA ESPERADA

- Números reales: contexto 5.6 kB (con `.dockerignore`), `pip install` ≈ 5.5 s, imagen 274 MB en disco / 59.6 MB comprimida (`DISK USAGE` vs `CONTENT SIZE` : lo comprimido es lo que viaja en push/pull).
- Capas (`docker history`): `pip install` 56.3 MB, `COPY app.py` 16 kB; el resto es la imagen base.
- **A.** Agregar un comentario a `app.py` y construir `:0.2` → `pip install` sale `CACHED` ; se rehacen `COPY app.py` **y** `useradd` (cuando una capa cambia, todas las de abajo se invalidan aunque su instrucción no cambie).
- **B.** Agregar un comentario a `requirements.txt` y construir `:0.3` → `pip install` se rehace (5.7 s). Docker compara bytes, no significado.

POR QUÉ, LÍNEA POR LÍNEA

- `FROM python:3.13-slim` : la imagen base; ya trae Python 3.13.
- `ENV PYTHONUNBUFFERED=1` : los logs salen al instante, sin quedarse en un buffer.
- `WORKDIR /app` : la carpeta de trabajo; el `.` de cada `COPY` apunta ahí.
- `COPY requirements.txt .` **antes** que el código, y luego `RUN pip install` : caché de capas. Si el ticket no cambió, Docker no vuelve al súper: reusa la capa de dependencias.
- `--no-cache-dir` : pip no guarda su caché de descargas dentro de la capa.
- `RUN useradd --create-home appuser` + `USER appuser` : el proceso no corre como root (paso 5).
- `EXPOSE 8080` : solo documenta; no publica. Publicas con `-p` al correrlo.
- `CMD ["uvicorn", …]` en forma exec y `--host 0.0.0.0` : con `127.0.0.1` nadie desde fuera del contenedor puede entrar.
- `.dockerignore` : lo que no viaja en el build context (`.venv` , `__pycache__` , `*.pyc`). Con él, el contexto pesa 5.6 kB.
- Mejora opcional: mover `useradd` antes de instalar dependencias para que solo se rehaga `COPY app.py` .
- Nota: el warning `Running pip as the 'root' user` es normal en el build (los paquetes quedan de root y `appuser` solo puede leerlos, lo cual es bueno).
- Nota: `WORKDIR` salió `CACHED` incluso con `--no-cache` (curiosidad, sin efecto).
- Nota: los wheels se descargan `aarch64` porque Docker Desktop en M4 construye Linux arm64.

TRAMPA · TE PASÓ

12.4 `.dockerignore` **vacío por no guardar** el archivo (`wc -c` lo delata).

12.6 **Todo** `CACHED` **en el «primer» build** porque una prueba previa con los mismos archivos dejó el caché: el caché se basa en el **contenido**, no en el nombre de la imagen; usar `--no-cache` para ver el build real.

En esta hoja: Sección 1 · Anatomía de un Dockerfile | Sección 2 · Capas, caché y .dockerignore

5 Usuario no-root: qué es y qué no es

✅ probado

QUÉ HACE Comprueba con qué identidad corre el proceso dentro de la imagen del gateway y qué puede tocar, con y sin `--user root` .

COMANDOS

Comprobación (usa la imagen del paso 4)

DESDE cualquier carpeta

```
docker run --rm iris-gateway:0.1 id
```

→ `uid=1000(appuser)`

```
docker run --rm --user root iris-gateway:0.1 id
```

→ `uid=0(root)`

```
docker run --rm iris-gateway:0.1 sh -c 'touch /etc/prueba && echo SI || echo NO'
```

→ `NO` (Permission denied)

```
docker run --rm --user root iris-gateway:0.1 sh -c 'touch /etc/prueba && echo SI || echo NO'
```

→ `SI`

SALIDA ESPERADA

- Sin `--user`: `uid=1000(appuser)`, y no puede escribir en `/etc` (`NO`, Permission denied).
- Con `--user root`: `uid=0(root)`, y sí puede (`SI`).

FRAGMENTO

Versión endurecida (la usa el model server, paso 6)

DOCKERFILE **fragmento del prompt** prompt · probado en carpeta temporal 3 líneas

```
RUN groupadd --system --gid 10001 app && \
    useradd --system --uid 10001 --gid app --no-create-home --shell /usr/sbin/nologin app
USER 10001:10001
```

POR QUÉ

- **Nadie hace login en un contenedor.** `USER` es con qué identidad corre el proceso.
- El riesgo no es un login, es una vulnerabilidad (en tu código o en una librería) que deja ejecutar comandos a través de un request; el atacante hereda los permisos del proceso. Por eso `USER` limita el daño. Solo es alcanzable lo que publicas con `-p`.

  NOTA DE REVISIÓN

  Matiz: «solo es alcanzable lo que publicas con `-p` » vale desde fuera de Docker. La documentación de Docker dice que, en una red bridge, los contenedores conectados a la misma red se alcanzan en todos sus puertos, publicados o no; `-p` los abre hacia fuera del host y hacia otras redes. Por eso, en el paso 7, el gateway llega a `model-server:8000` aunque tu Mac solo vea el 9000.

  Fuente: docs.docker.com/engine/network/port-publishing
- Versión endurecida: `--system` cuenta de servicio, `--no-create-home` sin carpeta personal, `--shell /usr/sbin/nologin` sin shell interactivo, `&& \` una sola capa.
- UID alto y numérico: Kubernetes `runAsNonRoot` necesita un UID numérico para verificarlo, y evita coincidir con usuarios reales del host. El número es convención, no regla.
- `sh -c '… && echo SI || echo NO'`: si `touch` funciona imprime SI; si falla, NO. `--rm` borra cada contenedor de prueba al terminar.

TRAMPA TÍPICA

Creer que `USER` blinda la imagen: `--user root` lo anula al correrla (lo viste: `SI`). `USER` limita lo que hereda quien entra por un request; quien controla `docker run` ya manda.

En esta hoja: Sección 3 · Imágenes chicas y seguras | Sección 1 · Anatomía de un Dockerfile

6 Model server: Dockerfile, DOS caminos para los pesos

✅ probado en carpeta temporal — falta crearlo y construirlo en el repo

QUÉ HACE Empaqueta el model server de dos maneras: con el modelo DENTRO de la imagen (variante A) o con el modelo FUERA, montado como volumen al arrancar (variante B). Las dos se probaron en una carpeta temporal; falta crearlas y construirlas en el repo.

Variante A — pesos DENTRO de la imagen ✅ probado en carpeta temporal

Problema: `model_repository/` está **fuera** de `model_server/` y Docker solo copia lo que está dentro del build context → el contexto es `iris-lab/` y se indica el Dockerfile con `-f`.

ARCHIVOS

DOCKERFILE **model_server/Dockerfile** prompt · probado en carpeta temporal 20 líneas

```
FROM python:3.13-slim

ENV PYTHONUNBUFFERED=1 \
    MODEL_REPOSITORY=/models

WORKDIR /app

RUN groupadd --system --gid 10001 app && \
    useradd --system --uid 10001 --gid app --no-create-home --shell /usr/sbin/nologin app

COPY model_server/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY model_repository/ /models/
COPY model_server/server.py .

USER 10001:10001

EXPOSE 8000
CMD ["uvicorn", "server:app", "--host", "0.0.0.0", "--port", "8000"]
```

TEXTO **model_server/Dockerfile.dockerignore** prompt · probado en carpeta temporal 7 líneas

```
**/.venv
**/__pycache__
**/*.pyc
gateway
training
scripts
README.md
```

BuildKit: aplica solo a ese Dockerfile; en `.dockerignore` los patrones son relativos a la raíz del contexto, por eso `**/`.

COMANDOS

DESDE `iris-lab` · el `.` es iris-lab/

```
docker build --progress=plain -f model_server/Dockerfile -t iris-model-server:0.1 .
```

SALIDA ESPERADA

Resultados de la prueba: contexto ≈ 8 kB, imagen 453 MB (≈180 MB más que el gateway por onnxruntime y sus dependencias), `id` → `uid=10001(app)`, `/v2/health/ready` → `{"ready":true}`, inferencia correcta.

POR QUÉ

- Orden de capas: usuario → dependencias → modelo → código (de lo que menos cambia a lo que más).
- Las rutas de `COPY` son relativas al contexto (`iris-lab/`): por eso `COPY model_server/requirements.txt .` y `COPY model_repository/ /models/`.
- `ENV MODEL_REPOSITORY=/models` : `server.py` busca el modelo en `/models/iris/1/model.onnx`, justo donde lo deja el `COPY`.
- `Dockerfile.dockerignore` : `**/.venv` saca el `.venv` de cualquier carpeta, y fuera quedan `gateway`, `training`, `scripts` y `README.md`. Por eso el contexto baja a ≈ 8 kB.
- `USER 10001:10001` al final: instalar y copiar se hace como root; el proceso corre como 10001 (paso 5).

Variante B — pesos FUERA de la imagen (volumen) ✅ probado en carpeta temporal

ARCHIVOS

DOCKERFILE **model_server/Dockerfile.volume** prompt · probado en carpeta temporal 19 líneas

```
FROM python:3.13-slim

ENV PYTHONUNBUFFERED=1 \
    MODEL_REPOSITORY=/models

WORKDIR /app

RUN groupadd --system --gid 10001 app && \
    useradd --system --uid 10001 --gid app --no-create-home --shell /usr/sbin/nologin app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY server.py .

USER 10001:10001

EXPOSE 8000
CMD ["uvicorn", "server:app", "--host", "0.0.0.0", "--port", "8000"]
```

Igual, pero sin `COPY model_repository` y con contexto `model_server/`.

TEXTO **model_server/.dockerignore** prompt · probado en carpeta temporal 4 líneas

```
.venv
__pycache__
*.pyc
Dockerfile*
```

COMANDOS

DESDE `iris-lab/model_server`

```
docker build -f Dockerfile.volume -t iris-model-server:vol .
```

DESDE `iris-lab` · `-v origen:destino:ro` → solo lectura

```
docker run --rm -p 9000:8000 -v "$PWD/model_repository:/models:ro" iris-model-server:vol
```

SALIDA ESPERADA

- Sin `-v` el contenedor **muere al arrancar**: `NoSuchFile: Load model from /models/iris/1/model.onnx failed` + `Application startup failed. Exiting.` → fail fast (el orquestador lo ve de inmediato).
- Tamaño igual (453 MB) porque el modelo pesa 1.3 KB.

POR QUÉ

- `-v origen:destino:ro`: monta la carpeta de tu Mac en `/models`, solo lectura. `$PWD` vuelve completa la ruta de origen.
- `Dockerfile*` en el `.dockerignore`: aunque algún día agregues `COPY . .`, los Dockerfiles no entran a la imagen.
- Mismo usuario, mismo orden de capas y mismo `MODEL_REPOSITORY=/models` que la variante A: solo cambia de dónde llega el modelo.

CUÁNDO CUÁL

	PESOS DENTRO	PESOS FUERA
Qué es la imagen	paquete cerrado código+modelo	solo código; el modelo llega al arrancar
Versión del modelo	= tag de la imagen	se lleva aparte; riesgo de desincronizar código y modelo
Rollback	cambias de tag	revertir imagen **y** modelo
Modelos de varios GB	cada versión = imagen enorme, push/pull lentos	imagen chica, el modelo se actualiza sin rebuild

- **Regla:** modelo chico → dentro (reproducible); modelo pesado o que cambia seguido → fuera.
- **Frase de entrevista:** «I copy the dependency manifest and install before copying the source, so code changes reuse the cached dependency layer.»

TRAMPA · TE PASÓ

9.3 Correr la variante B sin `-v` : el contenedor muere al arrancar. Es a propósito: `server.py` carga el modelo en el `lifespan` y, si el archivo no está, el proceso se cae en seguida y el orquestador lo ve.

En esta hoja: Sección 7 · Volúmenes y permisos · Sección 2 · Capas, caché y .dockerignore

7 Correr los dos contenedores y conectarlos

borrador sin probar · pendiente

QUÉ HACE Corre los dos contenedores y los conecta: primero a mano con `docker run`, luego con Docker Compose. Todavía no se ha probado.

Idea clave: dentro de un contenedor `localhost` es el propio contenedor. El gateway debe llegar al model server por **nombre** en una red de Docker, pasando `MODEL_SERVER_URL` con `-e` . El puerto 8000 del host está ocupado por tu contenedor `plant_pathology_zero_model-api-1` → el model server se publica como 9000.

7a. Con `docker run` (variante A)

borrador sin probar

COMANDOS

DESDE `iris-lab` · `smoke.sh` se llama con ruta relativa

```
docker network create iris-net
```

```
docker run -d --rm --name model-server --network iris-net -p 9000:8000 iris-model-server:0.1
```

```
docker run -d --rm --name gateway --network iris-net -p 8080:8080 \
  -e MODEL_SERVER_URL=http://model-server:8000 iris-gateway:0.1
```

```
./scripts/smoke.sh
```

```
docker logs gateway
```

```
docker stop gateway model-server && docker network rm iris-net
```

POR QUÉ

- `docker network create iris-net` : una red propia; dentro, cada contenedor se encuentra por su `--name` .
- `-p 9000:8000` es host:contenedor. El gateway, dentro de la red, usa el puerto del contenedor: `http://model-server:8000` .
- `-e MODEL_SERVER_URL=…` : la misma imagen del paso 4, con otra configuración.
- `-d` corre en segundo plano y `--rm` borra el contenedor al detenerlo: por eso al final basta `docker stop` .

7b. Con Docker Compose (variante A) borrador sin probar

ARCHIVOS

YAML **compose.yaml** prompt · borrador 24 líneas

```yaml
services:
  model-server:
    build:
      context: .
      dockerfile: model_server/Dockerfile
    image: iris-model-server:0.1
    ports:
      - "9000:8000"
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request;
        urllib.request.urlopen('http://localhost:8000/v2/health/ready')"]
      interval: 5s
      timeout: 3s
      retries: 10

  gateway:
    build: ./gateway
    image: iris-gateway:0.1
    ports:
      - "8080:8080"
    environment:
      MODEL_SERVER_URL: http://model-server:8000
    depends_on:
      model-server:
        condition: service_healthy
```

Va en `iris-lab/compose.yaml`.

COMANDOS

DESDE `iris-lab` · donde está `compose.yaml`; `up` se queda en primer plano, así que lo demás va en otra terminal

```
docker compose up --build
```

```
docker compose ps
```

```
./scripts/smoke.sh
```

```
docker compose down
```

POR QUÉ

- `build` + `image`: construye con ese contexto y ese Dockerfile, y le pone ese nombre a la imagen.
- `healthcheck`: `slim` no trae `curl`; por eso el healthcheck usa Python.
- `depends_on` con `condition: service_healthy`: el gateway arranca cuando pasa el healthcheck del model server.
- Compose crea por su cuenta la red del proyecto: los servicios se encuentran por su nombre (`model-server`), sin `docker network create`.
- `docker compose down`: detiene y borra los contenedores y la red.

7c. Compose con pesos fuera (variante B) borrador sin probar

FRAGMENTO

Cambia solo el servicio `model-server`:

YAML **fragmento del prompt** prompt · borrador 10 líneas

```yaml
  model-server:
    build:
      context: ./model_server
      dockerfile: Dockerfile.volume
    image: iris-model-server:vol
    volumes:
      - ./model_repository:/models:ro
    ports:
      - "9000:8000"
    healthcheck: # igual que arriba
```

POR QUÉ

- `volumes: - ./model_repository:/models:ro` es el `-v` del paso 6 escrito en Compose; aquí la ruta relativa vale, porque es relativa al `compose.yaml`.
- `# igual que arriba` es un comentario: en el archivo real va el bloque `healthcheck` completo de 7b.

SALIDA ESPERADA

Sin salidas anotadas: es un borrador. La meta de 7a y de 7b es que `./scripts/smoke.sh` pase igual que en el paso 2.

TRAMPA TÍPICA

Darle al gateway en contenedor la URL que funcionaba sin Docker (`http://localhost:9000`) o ninguna (default `localhost:8000`): dentro del contenedor, `localhost` es el gateway mismo y `/readyz` da 503. Lo vas a provocar en el paso 9, experimento 4.

En esta hoja: Sección 6 · Redes y puertos | Sección 8 · Docker Compose

8 Subir las imágenes (push)

borrador sin probar

QUÉ HACE Sube la imagen a un registry. Aún no se decidió a cuál, así que van las dos opciones: uno local (no sale de la Mac) o Docker Hub (requiere cuenta propia; sale de la Mac).

COMANDOS

Registry local (no sale de la Mac)

DESDE cualquier carpeta

```
docker run -d -p 5000:5000 --name registry registry:2
```

```
docker tag iris-gateway:0.1 localhost:5000/iris-gateway:0.1
```

```
docker push localhost:5000/iris-gateway:0.1
```

```
docker rmi localhost:5000/iris-gateway:0.1 iris-gateway:0.1
```

```
docker pull localhost:5000/iris-gateway:0.1
```

Docker Hub (requiere cuenta propia; sale de la Mac)

DESDE cualquier carpeta · `TU_USUARIO` = tu usuario de Docker Hub

```
docker login
```

```
docker tag iris-gateway:0.1 TU_USUARIO/iris-gateway:0.1
```

```
docker push TU_USUARIO/iris-gateway:0.1
```

SALIDA ESPERADA

Sin salidas anotadas: es un borrador. El dato que ya tienes: lo que viaja son las capas comprimidas (≈60 MB el gateway).

POR QUÉ

- Nombre completo de una imagen = `registry/usuario/repo:tag` . `docker tag` solo le pone otro nombre a la misma imagen; no copia nada.
- Lo que viaja son las capas comprimidas (≈60 MB el gateway: el `CONTENT SIZE` del paso 4); las capas ya existentes en el registry no se vuelven a subir.
- `registry:2` es un registry de verdad, corriendo en un contenedor de tu Mac, en el puerto 5000.
- `docker rmi` y después `docker pull` : borras tus copias locales y la bajas del registry; así compruebas que de verdad quedó allá.

TRAMPA TÍPICA

Hacer `docker push iris-gateway:0.1` sin el nombre completo: sin registry delante, el nombre apunta a Docker Hub, no a `localhost:5000` . Primero `docker tag` con el nombre completo y después el push de ese nombre.

En esta hoja: Sección 11 · Registries y tags inmutables

9 Romperlo (experimentos)

🟡 parcial — pendientes, salvo los marcados ✅

QUÉ HACE Rompe el sistema a propósito para reconocer cada falla cuando te pase de verdad. Para cada uno: qué esperas → qué pasa → por qué → cómo se arregla. Tres ya los viviste; el resto está pendiente.

COMANDOS

No hay comandos nuevos: cada experimento reusa los de los pasos 2, 4, 6 y 7 con el cambio que dice.

EXPERIMENTOS

9.1 ✅ probado — Arrancar el gateway con la variable en una **línea aparte**

QUÉ PASÓ: `MODEL_SERVER_URL=...` en una línea y el comando en la siguiente: el gateway usó el default `localhost:8000` y `/readyz` dio 503 `RemoteProtocolError` (en ese puerto contestaba tu otro contenedor).

CÓMO SE ARREGLA: La variable debe ir en la misma línea que el comando.

TRAMPA: 12.3 La variable de entorno en una línea aparte · 12.2 El puerto 8000 del host ocupado

9.2 ✅ probado — `.dockerignore` vacío

QUÉ PASÓ: 0 bytes, sin guardar: el build habría mandado el `.venv`.

CÓMO SE ARREGLA: Guárdalo con contenido; `wc -c` lo delata.

TRAMPA: 12.4 `.dockerignore` vacío por no guardar

9.3 ✅ probado — Correr la variante B sin `-v`

QUÉ PASÓ: El contenedor muere al arrancar (fail fast).

POR QUÉ: `server.py` carga el modelo en el `lifespan`: sin archivo, no hay proceso.

CÓMO SE ARREGLA: Montar el modelo con `-v` (paso 6).

9.4 🟡 pendiente — Gateway en contenedor **sin** `-e MODEL_SERVER_URL`

QUÉ ESPERAS: Usa `localhost:8000` → 503 en `/readyz`.

QUÉ PASA: pendiente

POR QUÉ: Dentro del contenedor, `localhost` es el gateway mismo (paso 7).

CÓMO SE ARREGLA: `-e MODEL_SERVER_URL=http://model-server:8000`.

9.5 🟡 pendiente — Cambiar `--host 0.0.0.0` por `127.0.0.1` en el `CMD`

QUÉ ESPERAS: El contenedor corre pero el `-p` no llega (connection reset/empty reply).

QUÉ PASA: pendiente

POR QUÉ: Con `127.0.0.1` nadie desde fuera del contenedor puede entrar (paso 4).

CÓMO SE ARREGLA: Volver a `--host 0.0.0.0`.

EN ESTA HOJA: Sección 6 · Redes y puertos

9.6 🟡 pendiente — Puerto del host ocupado

QUÉ ESPERAS: `-p 8000:8000` con tu otro contenedor → error `port is already allocated`.

QUÉ PASA: pendiente

CÓMO SE ARREGLA: Otro puerto del host, como el `-p 9000:8000` de este lab. Para ver quién lo ocupa: `lsof -nP -iTCP:8000 -sTCP:LISTEN` y `docker ps`.

EN ESTA HOJA: Sección 6 · Redes y puertos

9.7 🟡 pendiente — `docker stop` y SIGTERM

QUÉ ESPERAS: Ver en los logs el apagado limpio (código tras el `yield`) vs `docker kill`.

QUÉ PASA: pendiente

POR QUÉ: `docker stop` manda SIGTERM y espera; `docker kill` manda SIGKILL, que no deja correr nada después del `yield`.

EN ESTA HOJA: Sección 5 · ENTRYPOINT vs CMD, PID 1 y señales

9.8 🟡 pendiente `requirements.in` **sin fijar versiones vs** `.txt`

QUÉ ESPERAS: Mostrar la deriva.

QUÉ PASA: pendiente

POR QUÉ: La lista no dice versiones: cada instalación toma las más nuevas. El ticket sí: siempre las mismas (paso 3).

9.9 ✅ en el paso 5 **Correr como root vs** `appuser`

QUÉ PASÓ: Ya en el paso 5: `uid=1000(appuser)` contra `uid=0(root)`, y `NO` contra `SI` al escribir en `/etc`.

9.10 🟡 pendiente **Plataforma**

QUÉ ESPERAS: Construir con `--platform linux/amd64` y compararlo con arm64.

QUÉ PASA: pendiente

EN ESTA HOJA: Sección 10 · arm64 (tu Mac) vs amd64 (el servidor)

9.11 🟡 pendiente **Reemplazar el model server por Triton (solo cambiar la imagen/URL)**

QUÉ ESPERAS: Requiere `config.pbtxt` (borrado del repo; aquí abajo, como referencia). Triton ya trae los endpoints v2 (`/v2/health/ready`, `/v2/models/iris/ready`, `/infer`) y métricas Prometheus en el puerto 8002; el model server propio los implementa a mano.

QUÉ PASA: pendiente

ARCHIVOS

TEXTO **model_repository/iris/config.pbtxt** prompt · borrador 7 líneas

```
name: "iris"
backend: "onnxruntime"
max_batch_size: 64
input [ { name: "input", data_type: TYPE_FP32, dims: [ 4 ] } ]
output [ { name: "probs", data_type: TYPE_FP32, dims: [ 3 ] } ]
instance_group [ { kind: KIND_CPU, count: 1 } ]
dynamic_batching { }
```

El que se usó para Triton: guárdalo como referencia. **No se corrió** porque no cabía en el disco.

SALIDA ESPERADA

Solo los marcados ✅ tienen resultado anotado (en cada experimento). Los pendientes muestran lo que esperas, no lo que pasó.

POR QUÉ, EN EL CONFIG.PBTXT

- `name: "iris"` coincide con la carpeta `model_repository/iris/`, y `backend: "onnxruntime"` le dice a Triton con qué correr el `model.onnx`.
- `max_batch_size: 64`: el mismo 64 del `MAX_BATCH_SIZE` del model server y del `max_length=64` del gateway.
- `dims: [ 4 ]` y `dims: [ 3 ]` sin el batch: como `max_batch_size` es mayor que 0, Triton agrega esa dimensión.
- `input` y `output` se llaman como en el export (`input_names=["input"]`, `output_names=["probs"]`).
- `dynamic_batching { }` junta requests en un solo forward; `instance_group`: una copia del modelo, en CPU.

TRAMPA · TE PASÓ

12.1 Disco lleno al bajar Triton (≈10 GB comprimida, ≈30 GB de pico con el containerd store): Docker Desktop se cayó (`Docker Desktop is unable to start`) y hasta Claude Code dejó de poder crear archivos temporales. Antes de bajar imágenes grandes: `df -h /` y `docker system df`. Recuperación: liberar espacio (`uv cache prune`), matar y reabrir Docker Desktop, `docker image prune -af`.

Ojo antes del experimento 11: los comandos para medir el disco y limpiar están en el paso 11.

En esta hoja: Sección 13 · Errores de un vistazo | Sección 5 · ENTRYPOINT vs CMD, PID 1 y señales | Sección 10 · arm64 (tu Mac) vs amd64 (el servidor)

10 AWS

plan, no probado — cada comando: verificar con la doc oficial

QUÉ HACE El plan para llevar los dos servicios a AWS: las imágenes a ECR y los contenedores a ECS con Fargate. No tienes cuenta todavía y nada de esto se ha corrido: cada comando lleva «verificar con la doc oficial».

CONTEXTO Y COSTOS

- No tienes cuenta todavía. Las cuentas nuevas reciben créditos ($100 + hasta $100 por actividades) y un plan gratuito de máximo 6 meses; no se pudo confirmar si piden tarjeta al registrarse.

  > **NOTA DE REVISIÓN**
  >
  > Las páginas oficiales no coinciden. El anuncio de AWS del 16 de septiembre de 2026 dice que a la mayoría de las cuentas nuevas no se les pide tarjeta; las preguntas frecuentes del registro dicen que sí piden una tarjeta para verificar la cuenta y cobran 1 USD que después reembolsan. Lo vas a saber en la pantalla de registro.
  >
  > Fuentes: aws.amazon.com · anuncio del 16-sep-2026 · aws.amazon.com/free/registration-faqs

- Costos aproximados: ECR centavos; Fargate ≈ $0.04 por vCPU-hora (un contenedor de 0.25 vCPU / 0.5 GB ≈ $9/mes encendido 24/7; no verificado contra la página oficial); el balanceador de carga (ALB) es probablemente lo más caro y **su precio no está verificado**.

  > **NOTA DE REVISIÓN**
  >
  > Revisado el 29 de octubre de 2026 en las páginas de precios de AWS (us-east-1). Fargate Linux/x86: $0.000011244 por vCPU-segundo y $0.000001235 por GB-segundo → 0.25 vCPU + 0.5 GB durante 730 h ≈ $9.0 al mes, como estimaste (en ARM64, ≈ $7.2). ALB: $0.0225 por hora (≈ $16.4 al mes) más $0.008 por LCU-hora: sí, es lo más caro. Y algo que falta en el plan: sin ALB, la task necesita una IP pública, y cada IPv4 pública cuesta $0.005 por hora (≈ $3.65 al mes).
  >
  > Fuentes: aws.amazon.com/fargate/pricing · aws.amazon.com/elasticloadbalancing/pricing · aws.amazon.com/vpc/pricing

- Recomendaciones: alerta de presupuesto desde el inicio, no usar ALB al principio (exponer el gateway con un security group limitado a tu IP) y **borrar todo al terminar**.

QUÉ CAMBIA RESPECTO A LOCAL

PIEZA	EN TU MAC	EN AWS
requirements, Dockerfiles, caché, usuario, `0.0.0.0`, variables de entorno	los de los pasos 3 a 6	**no cambian**
Arquitectura	M4 = arm64	Fargate x86 (construir para amd64) o Fargate ARM64 (Graviton)
Registry	local o Docker Hub (paso 8)	ECR
Dónde corre	Docker Desktop	ECS con Fargate
Cómo se encuentran los servicios	por nombre, en una red de Docker (paso 7)	misma task: `localhost`; tasks separadas: service discovery / Service Connect
Cómo llega el modelo si va fuera de la imagen	`-v` desde tu Mac (paso 6)	S3 al arrancar, un volumen EFS o, en Kubernetes, un volumen persistente o un init container

PASOS (ESQUELETO)

La persona dueña de la cuenta crea las credenciales y configura el acceso: **no pegues claves en el chat**.

1. Crear cuenta, alerta de presupuesto, instalar la CLI, configurar acceso y comprobarlo.

   DESDE cualquier carpeta

   VERIFICAR CON LA DOC OFICIAL

   ```
   brew install awscli
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws configure sso
   ```

   o, con claves de acceso:

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws configure
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws sts get-caller-identity
   ```

2. Arquitectura. Opción 1: Fargate en x86 → construir para amd64 (en la Mac se emula y es más lento). Opción 2: Fargate en ARM64 (Graviton, más barato) → sirve la imagen arm64 tal cual; en la task definition se indica `cpuArchitecture: ARM64`. El `requirements.txt` universal sirve en ambas.

   DESDE la carpeta de cada imagen, como en los pasos 4 y 6 · en lugar de `...` va el nombre de la imagen

   VERIFICAR CON LA DOC OFICIAL

   ```
   docker buildx build --platform linux/amd64 -t ... --load .
   ```

3. ECR: un repositorio por imagen, el login de Docker contra ECR, tag y push (igual para `iris-model-server`).

   DESDE cualquier carpeta · `ACCOUNT_ID` = el número de tu cuenta

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws ecr create-repository --repository-name iris-gateway --region us-east-1
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws ecr create-repository --repository-name iris-model-server --region us-east-1
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin ACCOUNT_ID
   .dkr.ecr.us-east-1.amazonaws.com
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   docker tag iris-gateway:0.1 ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/iris-gateway:0.1
   ```

   VERIFICAR CON LA DOC OFICIAL

   ```
   docker push ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/iris-gateway:0.1
   ```

4. Correr en ECS con Fargate: cluster, task definition (los dos contenedores, `MODEL_SERVER_URL`, logs a CloudWatch), servicio. Cómo se encuentran: dos contenedores en la **misma task** se hablan por `localhost` (entonces `MODEL_SERVER_URL=http://localhost:8000`), o en tasks separadas con service discovery / Service Connect. La diferencia con tu Mac: ahí, `localhost` dentro de un contenedor es solo ese contenedor (paso 7); en una task de Fargate, los contenedores comparten la red de la task, y `localhost` llega al otro.

5. Health checks: `/readyz` del gateway se usa como health check del balanceador / del contenedor.

   NOTA DE REVISIÓN

   En ECS no hay un «readiness» aparte: si una task de un servicio falla el health check del balanceador, ECS la detiene y la reinicia; si falla el de un contenedor esencial, la marca UNHEALTHY y la reemplaza. Con el gateway y el model server en tasks separadas, `/readyz` ahí reiniciaría al gateway cada vez que falle el model server, justo lo que tu `app.py` evita con `/healthz`.

   Fuentes: docs.aws.amazon.com · Use an Application Load Balancer for Amazon ECS · docs.aws.amazon.com · HealthCheck (API de ECS)

6. Red: el model server **privado**, solo el gateway accesible desde fuera.

7. Logs: stdout → CloudWatch (por eso `PYTHONUNBUFFERED=1`).

8. **Limpieza obligatoria**: eliminar servicio, cluster, repositorios ECR y cualquier balanceador.

LOS PESOS EN AWS

- **Pesos dentro de la imagen:** no cambia nada. Se sube la imagen variante A a ECR y el modelo viaja con ella.
- **Pesos fuera de la imagen** (el `-v` de la Mac no existe): (a) bajar el modelo de **S3** al arrancar, con un script de entrada o un contenedor init que copie a un volumen compartido (el rol de la task necesita permiso de lectura al bucket); (b) montar un volumen **EFS** en la task definition en `/models`; (c) en Kubernetes, un volumen persistente o un init container. `server.py` ya falla rápido si el modelo no está, lo cual es deseable.
- **Argumento de diseño:** separar gateway y model server permite reemplazar el segundo por Triton (en una instancia con GPU o con disco suficiente) cambiando solo la imagen y `MODEL_SERVER_URL`.

SALIDA ESPERADA

Ninguna todavía: es un plan, no se ha corrido nada.

POR QUÉ

- **ECR** es el registry de AWS: como Docker Hub, pero dentro de tu cuenta. **ECS** es el servicio que corre contenedores, y **Fargate** es el modo en que AWS pone las máquinas: tú solo pides CPU y memoria por task.
- La **task definition** es la receta de la task: imágenes, CPU, memoria, variables de entorno, health checks y a dónde van los logs.
- `aws ecr get-login-password | docker login …`: la CLI de AWS saca una contraseña temporal y Docker la usa para entrar al registry, sin que la escribas.

TRAMPA TÍPICA

Construir en tu Mac (arm64) y correr en Fargate x86 sin `--platform linux/amd64`: la imagen no arranca. O construyes para amd64, o pides ARM64 en la task definition. Y la otra: olvidar la limpieza; lo que dejas encendido sigue cobrando.

En esta hoja: Sección 10 · arm64 (tu Mac) vs amd64 (el servidor) | Sección 11 · Registries y tags inmutables

11 Comandos de inspección y limpieza

✅ probado — referencia rápida

QUÉ HACE Una referencia rápida para ver qué hay en tu Mac (imágenes, contenedores, capas, logs y disco) y para limpiar. Lo que va en mayúsculas lo cambias tú.

COMANDOS

DESDE **cualquier carpeta**

```
docker images
```

tus imágenes, con su tamaño

```
docker ps -a
```

los contenedores, también los detenidos

```
docker history IMAGEN:TAG
```

las capas de una imagen y cuánto pesa cada una

```
docker logs NOMBRE
```

lo que el contenedor escribió en stdout y stderr

```
docker exec -it NOMBRE sh
```

una shell dentro de un contenedor que está corriendo

```
docker system df
```

cuánto ocupa Docker: imágenes, contenedores, volúmenes y caché de build

```
docker rmi IMAGEN:TAG
```

borra una imagen

```
docker image prune -af
```

borra todas las imágenes que ningún contenedor usa, sin preguntar

```
docker builder prune -af
```

borra la caché de build, sin preguntar

```
df -h /
```

cuánto espacio le queda al disco de tu Mac

SALIDA ESPERADA

Sin salidas anotadas: cada comando muestra el estado de tu Mac en ese momento.

POR QUÉ

- En `prune`, `-a` quiere decir todas las imágenes sin uso, no solo las «colgadas» (las que perdieron su tag), y `-f`, sin pedir confirmación.
- Antes de bajar una imagen grande: `df -h /` y `docker system df` (trampa 12.1).

TRAMPA TÍPICA

`docker image prune -af` también borra tus `iris-gateway:0.1` e `iris-model-server:0.1` si ningún contenedor las usa: después toca reconstruirlas.

En esta hoja: Sección 12 · Diagnóstico: las 10 fallas típicas

12 Errores reales que te pasaron

8 errores reales

QUÉ HACE Los errores que de verdad te pasaron. Cada uno aparece como trampa en el paso donde ocurre; aquí están juntos.

12.1 **Disco lleno al bajar Triton** (≈10 GB comprimida, ≈30 GB de pico con el containerd store): Docker Desktop se cayó (`Docker Desktop is unable to start`) y hasta Claude Code dejó de poder crear archivos temporales. Antes de bajar imágenes grandes: `df -h /` y `docker system df`. Recuperación: liberar espacio (`uv cache prune`), matar y reabrir Docker Desktop, `docker image prune -af`.
→ Paso 9 · Romperlo (experimentos)

12.2 **El puerto 8000 del host ocupado** por otro contenedor → el gateway «conectaba» ahí y daba `RemoteProtocolError`. Diagnóstico: `lsof -nP -iTCP:8000 -sTCP:LISTEN` y `docker ps`.
→ Paso 2 · Correr los dos servicios SIN Docker

12.3 **La variable de entorno en una línea aparte** no se la pasa al programa.
→ Paso 2 · Correr los dos servicios SIN Docker

12.4 `.dockerignore` **vacío por no guardar** el archivo (`wc -c` lo delata).
→ Paso 4 · Gateway: Dockerfile, build, caché

12.5 **Carpetas ocultas en macOS** `.claude` no se ve en Finder → `Cmd+Shift+.` o `Cmd+Shift+G` y pegar la ruta, o `code RUTA` desde la terminal.
→ Paso 0 · Qué es el proyecto

12.6 **Todo** `CACHED` **en el «primer» build** porque una prueba previa con los mismos archivos dejó el caché: el caché se basa en el **contenido**, no en el nombre de la imagen; usar `--no-cache` para ver el build real.
→ Paso 4 · Gateway: Dockerfile, build, caché

12.7 `.gitignore` **raíz ignora** `*.onnx` el modelo no entraba al commit → excepción en `iris-lab/.gitignore` (`!model_repository/**/*.onnx`).
→ Paso 1 · Entrenar y exportar el modelo

12.8 **Pegar una plantilla sobre** `pyproject.toml` (otro proyecto, Caso1) desincronizó el lock: el lock se deriva del archivo de dependencias, nunca al revés.
→ Paso 3 · Dependencias fijadas: requirements.in → requirements.txt

Comandos para Docker Engine 29.x, Compose v2 o posterior y el NVIDIA Container Toolkit, revisados contra la documentación oficial (docs.docker.com, la especificación de Compose y la documentación de NVIDIA). Los laboratorios no se ejecutaron al escribir esta hoja: si tu terminal dice otra cosa, créele a tu terminal. Los ejemplos salen de `reference/` del curso. El laboratorio Iris es otra cosa: es el tuyo, con tus archivos copiados tal cual y solo las salidas que anotaste; cada paso dice si se probó.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error