PuigFlows

IA y Automatizacion · puigflows.com

Imágenes Docker en Producción: Multi-stage, Caché de BuildKit, Distroless y Builds Reproducibles

Docker Images in Production: Multi-stage Builds, BuildKit Caching, Distroless and Reproducible Builds

Guia · Nivel intermedio · Proyectos Reales

Tiempo de lectura estimado: 57 min / Estimated reading time: 57 min

Edicion bilingue (Espanol / English) · Contenido premium

Josue Garcia · 3 de septiembre de 2026

VERSION EN ESPANOL

Imágenes Docker en Producción: Multi-stage, Caché de BuildKit, Distroless y Builds Reproducibles

El problema: la imagen que nadie mira hasta que duele

Casi todos los proyectos empiezan igual. Alguien escribe un Dockerfile de siete líneas, funciona en local, y ahí se queda durante dos años. El coste aparece despacio y por partes: el pipeline tarda catorce minutos porque cada commit reinstala las dependencias desde cero, el escáner de vulnerabilidades reporta trescientos CVE que vienen del sistema base y no de tu código, el autoescalado tarda cuarenta segundos en levantar un pod porque hay que descargar 1,4 GB, y la factura del registro crece sin que nadie sepa muy bien por qué.

Ninguno de esos problemas se arregla con un truco. Se arreglan entendiendo tres cosas: cómo funciona la caché de capas, qué hace realmente un build multi-etapa, y qué contiene la imagen base que elegiste sin pensar. Esta guía cubre las tres, con Dockerfiles completos para Python, Node y Go que puedes copiar hoy.

Capas y caché: la única regla que importa

Una imagen de contenedor es una pila de capas de solo lectura. Cada instrucción del Dockerfile que modifica el sistema de ficheros produce una capa. Al construir, BuildKit calcula una clave de caché por instrucción; si la clave coincide con una capa previa, la reutiliza y no ejecuta nada.

Para `RUN` la clave es el texto literal del comando más la clave de la capa anterior. Para `COPY` y `ADD` es el contenido de los ficheros copiados. De ahí sale la consecuencia práctica: **una capa invalidada invalida todas las siguientes**. Si copias tu código fuente antes de instalar dependencias, cada cambio de una línea en un fichero Python tira a la basura la instalación completa de paquetes.

```
# Mal: cambiar cualquier fichero reinstala todo
COPY . .
RUN pip install -r requirements.txt

# Bien: el manifiesto cambia poco, el codigo cambia siempre
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

La regla general: ordena el Dockerfile de lo que menos cambia a lo que más cambia. Manifiestos de dependencias primero, código después. Es un cambio de dos líneas que suele recortar más tiempo de CI que cualquier otra optimización.

.dockerignore no es opcional

El contexto de build se envía entero al daemon antes de ejecutar nada. Sin un `.dockerignore`, tu directorio `.git`, tus `node_modules` locales y tu carpeta de datasets viajan en cada build, y además contaminan la clave de caché de `COPY . .`: basta con hacer un commit para invalidarla, aunque no

hayas tocado el código.

```
.git
.gitignore
node_modules
__pycache__
*.pyc
.venv
.env
.env.*
dist
build
coverage
.pytest_cache
.mypy_cache
Dockerfile*
docker-compose*.yml
README.md
tests/fixtures/large/
```

Multi-stage: separar lo que construye de lo que corre

Un build multi-etapa usa varios `FROM` en el mismo Dockerfile. Las etapas intermedias tienen compiladores, cabeceras de desarrollo y herramientas de build; la etapa final copia solo el artefacto terminado. Nada de lo que quedó en las etapas anteriores llega a la imagen publicada, ni siquiera como capa borrada.

Ese matiz es importante desde el punto de vista de seguridad: en una imagen de una sola etapa, un `RUN rm secreto.pem` no elimina el fichero, solo lo oculta con una capa nueva. Cualquiera con la imagen puede extraerlo de la capa anterior. En multi-etapa el secreto que vivió en la etapa de build simplemente no existe en el resultado.

Python con uv, cache mounts y usuario no root

```
# syntax=docker/dockerfile:1.7
FROM python:3.13-slim AS builder

# uv resuelve e instala ordenes de magnitud mas rapido que pip
COPY --from=ghcr.io/astral-sh/uv:0.9.7 /uv /usr/local/bin/uv

WORKDIR /app
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy

# 1) Solo dependencias: esta capa sobrevive a los cambios de codigo
COPY pyproject.toml uv.lock ./
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-install-project --no-dev

# 2) Ahora el proyecto
COPY src/ ./src/
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev

# ---------- runtime ----------
FROM python:3.13-slim AS runtime

RUN useradd --create-home --uid 10001 app
```

```
WORKDIR /app

COPY --from=builder --chown=10001:10001 /app/.venv /app/.venv
COPY --from=builder --chown=10001:10001 /app/src /app/src

ENV PATH="/app/.venv/bin:$PATH" \
    PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1

USER 10001
EXPOSE 8000
CMD ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Tres detalles que hacen el trabajo. `--mount=type=cache` monta un directorio persistente entre builds que **no queda dentro de la imagen**: el índice descargado de paquetes se reutiliza incluso cuando la capa se invalida, y es la diferencia entre noventa segundos y tres. La doble invocación de `uv sync` (primero sin el proyecto, luego con él) mantiene las dependencias en una capa distinta a la de tu código. Y el entorno virtual copiado a la imagen final trae solo lo que necesita el runtime, sin el compilador ni la caché de uv.

Node con pnpm y prune de dependencias

```
# syntax=docker/dockerfile:1.7
FROM node:22-slim AS deps
WORKDIR /app
RUN corepack enable
COPY package.json pnpm-lock.yaml ./
RUN --mount=type=cache,target=/pnpm/store \
    pnpm config set store-dir /pnpm/store && pnpm install --frozen-lockfile

FROM node:22-slim AS build
WORKDIR /app
RUN corepack enable
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN pnpm run build
# Deja solo dependencias de produccion para copiarlas al runtime
RUN --mount=type=cache,target=/pnpm/store \
    pnpm prune --prod

FROM node:22-slim AS runtime
ENV NODE_ENV=production
WORKDIR /app
COPY --from=build --chown=node:node /app/node_modules ./node_modules
COPY --from=build --chown=node:node /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

El paso de pnpm prune --prod es el que casi siempre falta. Sin él te llevas TypeScript, ESLint, Vitest y sus árboles de dependencias a producción: cientos de megabytes y cientos de paquetes que nunca se ejecutan pero sí aparecen en tu informe de vulnerabilidades.

Secretos en tiempo de build sin filtrarlos

Pasar un token privado con `ARG` o `ENV` lo deja escrito en los metadatos de la imagen para siempre.

`docker history` lo enseña. BuildKit resuelve esto con montajes de secreto: el fichero existe durante ese `RUN` y desaparece después, sin producir capa.

```
# En el Dockerfile
RUN --mount=type=secret,id=npm_token,env=NPM_TOKEN \
    npm config set //registry.npmjs.org/:_authToken=${NPM_TOKEN} && \
    npm ci --omit=dev

# En la linea de comandos
docker build --secret id=npm_token,env=NPM_TOKEN -t api:dev .
```

Para dependencias privadas por SSH existe el equivalente `--mount=type=ssh`, que reenvía tu agente al build sin copiar la clave. Si alguna vez ves un `ARG GITHUB_TOKEN` en un Dockerfile, ese token está comprometido en cuanto la imagen sale del registro.

La caché que se evapora en CI

Los runners de CI son efímeros: cada job empieza con un almacén de build vacío, así que la caché local de capas no existe. La solución es exportar e importar la caché contra un registro.

```
- name: Build and push
  uses: docker/build-push-action@v6
  with:
    context: .
    push: true
    tags: ghcr.io/acme/api:${{ github.sha }}
    cache-from: type=registry,ref=ghcr.io/acme/api:buildcache
    cache-to: type=registry,ref=ghcr.io/acme/api:buildcache,mode=max
    provenance: mode=max
    sbom: true
```

El detalle que arruina la mitad de las configuraciones es `mode=max`. Por defecto el exportador usa `mode=min`, que guarda únicamente las capas presentes en la imagen final. En un build multi-etapa eso significa que las etapas de compilación —justo las caras— **no se cachean**, y cada ejecución vuelve a compilar todo mientras el log dice alegremente que la caché está activa. Con `mode=max` se exportan también las etapas intermedias.

Un aviso de coste: `mode=max` guarda mucho más, así que conviene un registro con política de retención sobre el tag de caché. Y si tu proveedor lo ofrece, un builder remoto persistente (Docker Build Cloud, un builder autoalojado con volumen, o buildx sobre Kubernetes) elimina el problema de raíz porque el almacén de BuildKit deja de ser efímero.

COPY --link para no arrastrar invalidaciones

Por defecto, un `COPY --from=builder` depende del estado del sistema de ficheros de la etapa origen: si esa etapa cambia, la copia se rehace. Con `COPY --link` la capa se crea de forma independiente y se enlaza al resultado, de modo que sigue siendo válida aunque cambie la etapa anterior o la imagen base. Es especialmente útil cuando actualizas la base del runtime y no quieres reconstruir todo lo que copiaste encima.

```
COPY --link --from=builder /app/.venv /app/.venv
```

La contrapartida: --link no ve el sistema de ficheros de destino, así que no puedes usarlo para escribir

sobre un directorio ya existente cuyo contenido esperas conservar, y se combina mal con --chown sobre usuarios creados en la propia etapa. Úsalo para copiar árboles completos a rutas nuevas.

Elegir la imagen base: cuatro familias, cuatro compromisos

Aquí es donde se decide el 90% del tamaño y de la superficie de ataque. Las opciones razonables en 2026:

- **Debian slim** (`python:3.13-slim`, `node:22-slim`). Glibc, gestor de paquetes, shell. Compatibilidad máxima y depuración cómoda. Entre 70 y 150 MB de base y un puñado de CVE heredados que rara vez son explotables pero que tu escáner marcará igual. Es el punto de partida sensato.
- **Alpine**. 5-10 MB, pero usa musl en lugar de glibc. Para Go compilado estáticamente es perfecto. Para Python es una trampa clásica: muchos paquetes publican wheels solo para manylinux (glibc), así que pip compila desde fuente y tu build pasa de dos minutos a veinte. Además hay diferencias reales de rendimiento en el asignador de memoria de musl bajo cargas con muchos hilos.
- **Distroless** (`gcr.io/distroless/*`, de Google, basadas en Debian). Solo el runtime del lenguaje y tu aplicación: sin shell, sin gestor de paquetes, sin `curl`. Reduce drásticamente lo que un atacante puede hacer tras un RCE. Traen un tag `:debug` con BusyBox para cuando necesitas entrar.
- **Wolfi / Chainguard y equivalentes endurecidos**. Filosofía distroless pero con glibc, reconstruidas a diario para mantener cero CVE conocidos, firmadas con Sigstore y con SBOM y procedencia SLSA adjuntas. El catálogo amplio es de pago; hay alternativas en el mismo espacio (Docker Hardened Images, UBI Micro, Chiselled Ubuntu). Si tienes que responder ante un cuestionario de seguridad, esto ahorra mucho tiempo.

Un consejo práctico: no saltes directamente de `:latest` a distroless. Pasa primero a `-slim` con multi-etapa, mide, y solo entonces evalúa si el runtime sin shell te compensa operativamente.

Go: el caso donde el resultado es de 8 MB

```
# syntax=docker/dockerfile:1.7
FROM golang:1.25 AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    CGO_ENABLED=0 go build -trimpath -ldflags="-s -w" -o /out/api ./cmd/api

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/api /api
USER nonroot:nonroot
EXPOSE 8080
ENTRYPOINT ["/api"]
```

`-trimpath` elimina rutas absolutas del binario (necesario para builds reproducibles), `-ldflags="-s -w"` quita la tabla de símbolos y la información de depuración, y `CGO_ENABLED=0` produce un binario estático que no necesita ni libc. La caché de `go-build` montada es lo que evita recompilar todo el árbol en cada cambio.

Que el contenedor se comporte como un buen ciudadano

Tamaño aparte, hay cuatro detalles que separan una imagen que funciona de una que se opera bien:

Usuario no root con UID numérico. Kubernetes no puede evaluar `runAsNonRoot` si el `USER` es un nombre, porque el nombre solo se resuelve dentro del contenedor. Usa `USER 10001`, no `USER app`.

ENTRYPOINT en forma exec. `CMD python app.py` (forma shell) arranca un `/bin/sh -c` como PID 1 que no reenvía SIGTERM a tu proceso. El resultado es que Kubernetes espera el periodo de gracia completo y luego mata con SIGKILL: peticiones perdidas en cada despliegue. Usa siempre la forma de lista: `CMD ["python", "app.py"]`.

Sistema de ficheros de solo lectura. Si la aplicación no escribe nada, declara `readOnlyRootFilesystem: true` y monta un `emptyDir` en `/tmp`. Construir la imagen pensando en eso desde el principio evita descubrir a las tres de la mañana que algo escribía en `/app`.

Metadatos OCI. Etiqueta el origen; te lo agradecerás cuando encuentres una imagen huérfana en el registro.

```
LABEL org.opencontainers.image.source="https://github.com/acme/api" \
      org.opencontainers.image.revision="${GIT_SHA}" \
      org.opencontainers.image.licenses="Apache-2.0"
```

Procedencia, SBOM y fijación por digest

Una imagen sin evidencia de cómo se construyó es un binario anónimo corriendo con acceso a tu red interna. BuildKit genera atestaciones de procedencia y SBOM sin trabajo extra:

```
docker buildx build \
  --provenance=mode=max \
  --sbom=true \
  -t ghcr.io/acme/api:1.4.2 --push .

# Inspeccionar despues
docker buildx imagetools inspect ghcr.io/acme/api:1.4.2 \
  --format '{{ json .Provenance }}'
```

Y el otro lado de la moneda: **fija tus imágenes base por digest**. Un tag es un puntero mutable; `python:3.13-slim` apunta a algo distinto cada semana, así que dos builds del mismo commit pueden producir imágenes diferentes.

```
FROM python:3.13-slim@sha256:1e5f0ba1f52d... AS runtime
```

Sí, eso obliga a actualizar el digest periódicamente; es exactamente el trabajo que Renovate o Dependabot hacen por ti con una regla de `docker`. Lo que ganas es que el build es determinista y que una base comprometida no entra silenciosamente en tu siguiente despliegue.

Para reproducibilidad completa hay una pieza más: las marcas de tiempo. BuildKit respeta `SOURCE_DATE_EPOCH` y reescribe los timestamps de las capas, de modo que dos builds del mismo árbol de fuentes producen el mismo digest.

```
SOURCE_DATE_EPOCH=$(git log -1 --pretty=%ct) \
docker buildx build --output type=image,rewrite-timestamp=true -t api:repro .
```

Builds multiplataforma: por qué tu build de ARM tarda siete veces más

El día que alguien del equipo compra un portátil con chip ARM, o el día que producción se muda a instancias Graviton, el Dockerfile que funcionaba deja de bastar. Necesitas una imagen que sirva para `linux/amd64` y `linux/arm64`, y la forma en que la construyas decide si el pipeline pasa de tres minutos a cinco o de tres minutos a veinte.

Hay tres caminos y conviene entender el compromiso de cada uno antes de elegir.

Emulación con QEMU. Es la opción de una línea: registras el intérprete binario y Docker ejecuta las instrucciones de la otra arquitectura traduciéndolas al vuelo. Funciona para casi todo y no exige tocar el Dockerfile. El problema es el precio. Traducir instrucción a instrucción es caro, y en trabajo intensivo de CPU (compilar C, construir extensiones nativas de Python, transpilar TypeScript) las mediciones publicadas por Docker apuntan a builds del orden de siete veces más lentos en la arquitectura emulada. Si tu etapa emulada supera los cinco minutos, ya has salido del rango en el que QEMU es una buena idea.

```
docker run --privileged --rm tonistiigi/binfmt --install all
docker buildx create --name multi --driver docker-container --use
docker buildx build --platform linux/amd64,linux/arm64 \
  -t registry.example.com/app:1.4.0 --push .
```

Compilación cruzada. En lugar de emular todo el build, ejecutas la etapa de construcción de forma nativa y le dices al compilador que produzca binarios para la otra arquitectura. BuildKit expone dos juegos de variables automáticas que hacen esto viable: `BUILDPLATFORM`, `BUILDOS` y `BUILDARCH` describen la máquina que construye; `TARGETPLATFORM`, `TARGETOS` y `TARGETARCH` describen la imagen que se produce. La clave está en anclar la etapa de build a `--platform=$BUILDPLATFORM` para que no la emule nadie.

```
FROM --platform=$BUILDPLATFORM golang:1.25-alpine AS build
ARG TARGETOS
ARG TARGETARCH
WORKDIR /src
COPY go.mod go.sum ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    CGO_ENABLED=0 GOOS=$TARGETOS GOARCH=$TARGETARCH \
    go build -trimpath -ldflags="-s -w" -o /out/api ./cmd/api

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]
```

Con este patrón, añadir una segunda arquitectura no duplica el tiempo: las etapas compartidas (descarga de módulos, copia de fuentes) se resuelven una sola vez y sólo el enlazado final se repite. Las cifras que publica Docker para este caso hablan de un incremento en torno al 70% frente al 600% de la emulación pura.

Runners nativos. La tercera vía es no emular ni compilar cruzado: tener un builder por arquitectura y

dejar que cada uno construya lo suyo en hardware real. Buildx soporta clústeres de nodos, y GitHub Actions ya ofrece runners ARM gestionados. Es la opción más rápida y la que menos sorpresas da con dependencias nativas, a cambio de infraestructura extra.

```
docker buildx create --name cluster --node amd --platform linux/amd64 \
  --driver remote tcp://builder-amd64.internal:1234
docker buildx create --append --name cluster --node arm --platform linux/arm64 \
  --driver remote tcp://builder-arm64.internal:1234
docker buildx build --builder cluster --platform linux/amd64,linux/arm64 \
  -t app:1.4.0 --push .
```

Una regla práctica: empieza por QEMU porque cuesta cero, mide, y muévete a compilación cruzada cuando la etapa emulada pase de cinco minutos. Reserva los runners nativos para cuando la compilación cruzada no sea viable, que en la práctica significa dependencias con extensiones en C que no cruzan bien: algunas ruedas de Python científico, módulos de Node que pasan por `node-gyp` y bibliotecas del sistema sin paquete cruzado disponible.

El error silencioso de las imágenes multiplataforma

Cuando construyes para dos plataformas, el resultado no es una imagen: es un índice (antes llamado *manifest list*) que apunta a dos imágenes. Eso tiene dos consecuencias que sorprenden a mucha gente. La primera es que `--load` no funciona con múltiples plataformas, porque el almacén de imágenes clásico del daemon no sabe guardar un índice; o usas `--push`, o activas el almacén con soporte de `containerd`. La segunda es que si etiquetas la imagen de una plataforma y la subes por separado, machacas el índice y dejas la otra arquitectura huérfana. Sube siempre el índice completo en una sola operación y comprueba el resultado:

```
docker buildx imagetools inspect registry.example.com/app:1.4.0
```

Bake: describir el build como configuración, no como un script de 40 líneas

Llega un momento en que el comando de build ya no cabe en una línea. Tienes tres imágenes (API, worker, migraciones), dos plataformas, etiquetas que dependen de la rama, argumentos que cambian entre entorno de pruebas y producción, y un bloque de `--cache-from` y `--cache-to` que nadie se atreve a tocar. Ese comando acaba copiado en el Makefile, en el workflow de CI y en el README, y las tres copias divergen.

`docker buildx bake` resuelve eso moviendo la definición del build a un fichero versionado. Soporta herencia entre targets, matrices, variables y grupos, y el mismo fichero se ejecuta idéntico en tu portátil y en CI.

```
variable "REGISTRY" { default = "registry.example.com" }
variable "TAG"      { default = "dev" }

group "default" {
  targets = ["api", "worker"]
}

target "_common" {
  context    = "."
  platforms  = ["linux/amd64", "linux/arm64"]
```

```
  args = {
    PYTHON_VERSION = "3.13"
  }
  cache-from = ["type=registry,ref=${REGISTRY}/cache:buildcache"]
  cache-to   = ["type=registry,ref=${REGISTRY}/cache:buildcache,mode=max"]
  attest = [
    "type=provenance,mode=max",
    "type=sbom"
  ]
}

target "api" {
  inherits = ["_common"]
  target   = "runtime"
  tags     = ["${REGISTRY}/api:${TAG}"]
}

target "worker" {
  inherits = ["_common"]
  target   = "worker-runtime"
  tags     = ["${REGISTRY}/worker:${TAG}"]
}
```

Con eso, construir todo es `docker buildx bake --push`, y construir sólo la API para probar en local es `docker buildx bake api --set api.platforms=linux/arm64 --load`. El bloque `--set` permite sobreescribir cualquier campo desde la línea de comandos sin editar el fichero, que es exactamente lo que necesitas para una prueba puntual.

Dos detalles que ahorran disgustos. El primero: `bake` lee ficheros `docker-bake.hcl`, `docker-bake.json` y también `compose.yaml`, así que si ya tienes Compose puedes empezar sin escribir nada nuevo. El segundo: los targets que empiezan por guión bajo por convención no son inválidos, simplemente no se incluyen en el grupo por defecto salvo que los nombres explícitamente, lo que los convierte en el sitio natural para la configuración compartida.

Probar dentro del build en lugar de después

Una etapa de test que no produce artefacto es una forma barata de garantizar que nadie publica una imagen con la suite en rojo. BuildKit no ejecuta etapas que nadie necesita, así que la etapa de test sólo corre cuando la pides.

```
FROM build AS test
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --group dev
RUN python -m pytest -q --maxfail=1

FROM scratch AS test-report
COPY --from=test /src/reports /
```

```
# En CI: falla el pipeline si los tests fallan, y extrae el informe
docker buildx bake test-report --set test-report.output=type=local,dest=./reports
```

El truco de la etapa `scratch` con `output=type=local` es que BuildKit escribe el contenido de la etapa en tu sistema de ficheros en lugar de en una imagen. Sirve para informes de cobertura, binarios compilados o cualquier artefacto que quieras sacar del build sin publicar una imagen.

Medir antes de optimizar: qué significa de verdad el tamaño de una imagen

Casi todas las guías de Docker terminan con una comparación de megabytes, y casi todas miden mal. El número que devuelve `docker images` es el tamaño descomprimido de todas las capas de esa imagen, contadas como si nadie compartiera nada. Ese no es el número que te importa.

Lo que te importa son tres cosas distintas:

- **Bytes transferidos en un pull en frío.** Es el tamaño *comprimido* de las capas, no el descomprimido. Una capa de 300 MB de texto puede viajar en 40 MB. Determina cuánto tarda un nodo nuevo en arrancar tu servicio.
- **Bytes transferidos en un despliegue normal.** Es sólo el tamaño de las capas que cambiaron. Si tus dependencias están en una capa estable y tu código en otra, un despliegue mueve kilobytes aunque la imagen pese 800 MB. Esta es la métrica que realmente afecta a la velocidad de despliegue, y la que casi nadie mide.
- **Superficie de ataque.** No se mide en megabytes sino en paquetes instalados. Una imagen de 900 MB con veinte paquetes es más segura que una de 120 MB con doscientos.

Para ver la anatomía real de una imagen, `docker history` te da el desglose por capa con el comando que la creó:

```
docker history --no-trunc --format 'table {{.Size}}\t{{.CreatedBy}}' app:1.4.0 | head -20
```

Y para el tamaño comprimido de verdad, que es el que viaja por la red, hay que preguntarle al registro:

```
docker buildx imagetools inspect --raw registry.example.com/app:1.4.0 \
  | jq '[.layers[].size] | add / 1024 / 1024 | round'
```

La diferencia entre ambos números suele ser de un factor de dos a tres. Si estás justificando una optimización ante tu equipo, usa el comprimido: es el que se traduce en segundos de arranque.

Las capas se comparten, y eso cambia la aritmética

Si tus quince servicios usan la misma imagen base, esa base se descarga y se almacena una sola vez por nodo. Reducir la base de 120 MB a 20 MB ahorra 100 MB una vez, no quince veces. En cambio, una capa de dependencias de aplicación de 300 MB que es distinta en cada servicio se paga quince veces.

La consecuencia práctica va contra la intuición: estandarizar la imagen base en toda la organización suele ahorrar más espacio en disco y más tiempo de pull que perseguir la base más pequeña posible en cada repositorio por separado. Y tiene un beneficio adicional: cuando aparece un CVE en la base, hay un solo sitio que actualizar.

Un presupuesto de tamaño en CI

Las imágenes engordan por acumulación de decisiones pequeñas, no por una sola. Un presupuesto que falla el build cuando se cruza el umbral convierte ese crecimiento en una conversación explícita.

```
#!/usr/bin/env bash
set -euo pipefail
IMAGE="${1:?uso: check-size.sh imagen:tag}"
```

```
BUDGET_MB="${2:-250}"

SIZE_MB=$(docker image inspect "$IMAGE" --format '{{.Size}}' | awk '{print int($1/1048576)}')
echo "Tamaño descomprimido: ${SIZE_MB} MB (presupuesto: ${BUDGET_MB} MB)"

if [ "$SIZE_MB" -gt "$BUDGET_MB" ]; then
  echo "La imagen supera el presupuesto. Capas más grandes:"
  docker history --format 'table {{.Size}}\t{{.CreatedBy}}' "$IMAGE" | head -8
  exit 1
fi
```

Ponlo en el pipeline de la rama principal, no en cada pull request: quieres una señal cuando algo entra, no un bloqueo constante mientras alguien experimenta.

Escanear vulnerabilidades sin ahogarse en ruido

El primer escaneo de una imagen basada en Debian devuelve entre doscientos y trescientos hallazgos. La reacción habitual es una de dos: ignorarlos todos, o abrir una tarea por cada uno y quemar dos semanas. Ninguna sirve. El escaneo sólo aporta valor cuando produce una lista corta de cosas accionables, y para llegar ahí hacen falta tres decisiones.

Primera: separar la base de tu código. Los CVE del sistema operativo se arreglan actualizando la imagen base, y eso es trabajo de mantenimiento programado. Los CVE de tus dependencias de aplicación se arreglan actualizando el lockfile, y eso es trabajo del equipo que escribe el código. Mezclar ambos en el mismo informe garantiza que nadie mire ninguno.

```
# Sólo lo que depende de ti: dependencias de aplicación, no del SO
trivy image --scanners vuln --vuln-type library \
  --severity HIGH,CRITICAL --ignore-unfixed \
  --exit-code 1 registry.example.com/app:1.4.0

# Informe separado de la base, para el ciclo de mantenimiento
trivy image --scanners vuln --vuln-type os \
  --severity HIGH,CRITICAL --format table \
  registry.example.com/app:1.4.0
```

Segunda: `--ignore-unfixed` **por defecto.** Un CVE sin parche disponible no es accionable: no puedes hacer nada excepto esperar o cambiar de base. Sacarlo del informe que bloquea el build no es esconder el problema, es separar lo que se puede arreglar hoy de lo que no. Mantén un informe aparte, sin ese filtro, que alguien revise una vez por semana.

Tercera: VEX para los falsos positivos estructurales. Muchos hallazgos son ciertos y a la vez irrelevantes: la biblioteca vulnerable está en la imagen pero tu código nunca llama al camino afectado. VEX (Vulnerability Exploitability eXchange) es el formato estándar para declarar eso de forma auditable, en lugar de mantener una lista de exclusiones en un fichero de configuración que nadie sabe justificar seis meses después.

```
{
  "@context": "https://openvex.dev/ns/v0.2.0",
  "author": "seguridad@example.com",
  "timestamp": "2026-09-02T09:00:00Z",
  "statements": [
    {
      "vulnerability": { "name": "CVE-2026-11841" },
```

```
      "products": [
        { "@id": "pkg:oci/app@sha256:9f2c1e...", "identifiers": { "purl": "pkg:oci/app" } }
      ],
      "status": "not_affected",
      "justification": "vulnerable_code_not_in_execute_path",
      "impact_statement": "El parser XML afectado sólo se usa en el script de importación,
que no forma parte de la imagen de runtime."
    }
  ]
}
```

```
trivy image --vex ./vex/app.openvex.json --severity HIGH,CRITICAL \
  --exit-code 1 registry.example.com/app:1.4.0
```

La diferencia entre una exclusión y una declaración VEX es que la segunda lleva autor, fecha y justificación de un vocabulario cerrado. Eso la hace revisable, y hace posible revocarla cuando el análisis deja de ser cierto.

Dónde poner la puerta

Escanear en el pull request y escanear la imagen publicada resuelven problemas distintos. En el pull request quieres retroalimentación rápida sobre lo que la persona acaba de cambiar; ahí el escaneo debe fallar sólo por dependencias de aplicación con parche disponible. Sobre la imagen publicada quieres el inventario completo, incluido lo que no tiene arreglo, porque necesitas saber a qué estás expuesto aunque no puedas actuar hoy.

Y hay un tercer momento que casi nadie cubre: el reescaneo periódico de imágenes ya desplegadas. Una imagen que era limpia el martes puede tener tres críticos el viernes porque se publicó un CVE nuevo, y ningún build va a avisarte porque nadie ha vuelto a construir. Un trabajo diario que escanee las etiquetas en producción y avise es probablemente el control con mejor relación entre esfuerzo y riesgo evitado de toda esta sección.

```
name: rescan-produccion
on:
  schedule: [{ cron: "0 6 * * *" }]
  workflow_dispatch:
jobs:
  scan:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        image: [api, worker, scheduler]
    steps:
      - uses: aquasecurity/trivy-action@master
        with:
          image-ref: registry.example.com/${{ matrix.image }}:production
          severity: HIGH,CRITICAL
          ignore-unfixed: true
          exit-code: "1"
          format: sarif
          output: trivy-${{ matrix.image }}.sarif
      - uses: github/codeql-action/upload-sarif@v3
        if: always()
        with:
          sarif_file: trivy-${{ matrix.image }}.sarif
```

Firmar la imagen y verificar la firma donde importa

Generar una SBOM y una atestación de procedencia responde a la pregunta *qué hay dentro y de dónde salió*. Falta la otra mitad: *quién dice eso, y cómo sé que no me lo han cambiado*. Esa es la función de la firma.

La manera moderna de firmar contenedores no usa claves de larga vida. Con el flujo *keyless* de Sigstore, el proceso de CI se identifica ante Fulcio con un token OIDC, recibe un certificado efímero de diez minutos, firma con él, y la firma queda anotada en Rekor, un registro público de transparencia y sólo-añadir. No hay clave privada que rotar ni que se pueda filtrar, y la verificación no comprueba una clave sino una identidad: *esta imagen fue firmada por el workflow release.yml del repositorio example/app*.

```
      - uses: sigstore/cosign-installer@v3
      - name: Firmar la imagen
        env:
          COSIGN_EXPERIMENTAL: "1"
        run: |
          cosign sign --yes \
            registry.example.com/app@${{ steps.build.outputs.digest }}
```

Fíjate en que se firma el *digest*, no la etiqueta. Firmar `app:1.4.0` no significa nada: la etiqueta es un puntero mutable y mañana puede apuntar a otra cosa. El digest es el contenido.

La verificación se ve así:

```
cosign verify \
  --certificate-identity-regexp
'^https://github.com/example/app/.github/workflows/release.yml@refs/heads/main$' \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
  registry.example.com/app@sha256:9f2c1e...
```

Ese comando es el que separa una firma decorativa de un control real. Si sólo compruebas que existe una firma, cualquiera con una cuenta puede firmar cualquier cosa y pasar tu verificación. Lo que hace el control efectivo es fijar la identidad concreta: ese repositorio, ese workflow, esa rama.

Verificar en admisión, no en el pipeline

Un `cosign verify` dentro del mismo pipeline que acaba de firmar la imagen comprueba muy poco. El sitio donde la verificación tiene valor es el punto en el que el clúster decide si arranca un pod, porque ahí atrapa también las imágenes que llegaron por caminos que nadie planificó: un despliegue manual, un chart de terceros, una etiqueta cambiada a mano.

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: exigir-firma-e-identidad
spec:
  validationFailureAction: Enforce
  background: false
  rules:
    - name: verificar-firma
      match:
        any:
```

```
          - resources:
              kinds: [Pod]
              namespaces: ["produccion", "staging"]
      verifyImages:
        - imageReferences:
            - "registry.example.com/*"
          mutateDigest: true
          required: true
          attestors:
            - entries:
                - keyless:
                    subject:
"https://github.com/example/*/.github/workflows/release.yml@refs/heads/main"
                    issuer: "https://token.actions.githubusercontent.com"
                    rekor:
                      url: https://rekor.sigstore.dev
```

El campo `mutateDigest: true` merece un comentario: reescribe la referencia de la imagen del pod para sustituir la etiqueta por el digest que se acaba de verificar. Sin eso, verificas un digest y el kubelet podría descargar otro si la etiqueta cambia entre la admisión y el arranque. Es una ventana pequeña, pero es exactamente el hueco que un atacante con acceso al registro necesitaría.

Despliega la política primero en modo `Audit` durante un par de semanas. Descubrirás imágenes en producción que nadie recordaba, y ese inventario ya justifica el ejercicio aunque nunca llegues a activar `Enforce`.

Depurar una imagen que no tiene shell

La objeción más habitual contra distroless es real: cuando algo falla a las tres de la mañana, no puedes entrar. No hay `sh`, no hay `ls`, no hay `curl`. Merece la pena responderla bien, porque la respuesta hace que la objeción deje de aplicar.

La técnica es la misma en Docker y en Kubernetes: no metas herramientas en la imagen de producción, adjunta un contenedor efímero que las traiga y comparta los espacios de nombres del contenedor roto.

```
# Docker: un contenedor con herramientas que comparte red y PID con el roto
docker run --rm -it \
  --pid=container:api \
  --network=container:api \
  --cap-add SYS_PTRACE \
  nicolaka/netshoot

# Dentro: el proceso del contenedor roto es visible y su red es la tuya
ps aux
ss -tlnp
curl -sv localhost:8080/healthz
cat /proc/1/environ | tr '\0' '\n'
```

```
# Kubernetes: contenedor efímero en el pod que ya está corriendo
kubectl debug -it pod/api-7d9f8b6c5-xk2p9 \
  --image=nicolaka/netshoot \
  --target=api \
  --profile=general

# Ver el sistema de ficheros del contenedor objetivo desde el efímero
ls /proc/1/root/app
```

Ese `/proc/1/root` es la parte que la gente no conoce y la que convierte la técnica en completa: desde el contenedor efímero puedes leer todo el sistema de ficheros del contenedor sin shell, incluidos ficheros de configuración, logs en disco y binarios. La imagen de producción se queda mínima y tú conservas todas las herramientas.

Para el caso concreto de que el contenedor ni siquiera arranque, hay dos trucos más. Uno: `kubectl debug pod/api --copy-to=api-debug --set-image=api=busybox --share-processes -it` crea una copia del pod con otra imagen y la misma configuración, red y volúmenes, sin tocar el original. Dos: en el Dockerfile, mantener una etapa `debug` que parte de la de producción y añade un shell, publicada con la misma etiqueta más el sufijo `-debug`. Cuesta un minuto de build y evita la tentación de meter un shell en la imagen real.

```
FROM gcr.io/distroless/static-debian12:nonroot AS runtime
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]

FROM gcr.io/distroless/static-debian12:debug-nonroot AS debug
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]
```

La caché de CI a fondo: los tres backends y cuándo usar cada uno

La sección anterior sobre la caché que se evapora explicaba el principio. Vale la pena bajar al detalle, porque la elección de backend cambia el resultado más que cualquier otro ajuste del pipeline.

Caché inline. `--cache-to type=inline` mete los metadatos de caché dentro de la propia imagen publicada. No requiere infraestructura y no cuesta nada, pero sólo guarda las capas de la etapa final. En un build multi-etapa, que es el caso normal, eso significa que las etapas caras (compilar, instalar dependencias) no se cachean en absoluto. Sirve para builds de una sola etapa y para poco más.

Caché en registro. `--cache-to type=registry,ref=...,mode=max` publica la caché como un artefacto separado en tu registro. Guarda todas las etapas intermedias, funciona entre runners efímeros y entre repositorios, y es la opción correcta para la mayoría de equipos. El `mode=max` no es opcional: sin él el comportamiento por defecto es `mode=min`, que guarda sólo la etapa final y te deja exactamente donde estabas.

```
docker buildx build \
  --cache-from type=registry,ref=registry.example.com/cache:main \
  --cache-from type=registry,ref=registry.example.com/cache:pr-${PR_NUMBER} \
  --cache-to   type=registry,ref=registry.example.com/cache:pr-${PR_NUMBER},mode=max \
  -t registry.example.com/app:${SHA} --push .
```

Ese patrón de dos `--cache-from` es el que hace que los pull requests sean rápidos: la rama lee primero su propia caché y, si no existe porque es el primer build, cae a la de la rama principal. Y escribe en su propia clave para no contaminar la caché compartida con estados intermedios.

Caché en almacenamiento de objetos. `type=s3`, `type=azblob` y `type=gha` guardan la caché fuera del registro. La variante de GitHub Actions es cómoda porque no requiere credenciales adicionales, pero está sujeta a la cuota de caché del repositorio y a su política de expulsión: cuando se llena, se borra lo menos usado recientemente, y eso puede significar que tu caché desaparezca un lunes sin explicación. Para pipelines grandes, S3 con una política de ciclo de vida propia da un comportamiento mucho más

predecible.

```
      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: registry.example.com/app:${{ github.sha }}
          cache-from: type=gha,scope=${{ github.ref_name }}
          cache-to: type=gha,mode=max,scope=${{ github.ref_name }}
          provenance: mode=max
          sbom: true
```

Por qué la caché no acierta aunque el código no haya cambiado

Cuando un build que debería ser instantáneo tarda diez minutos, la causa está casi siempre en una de estas cuatro:

1. **Una marca de tiempo en el contexto.** BuildKit calcula la clave de caché de un `COPY` a partir del contenido y los metadatos de los ficheros. Un fichero regenerado en cada ejecución de CI (un `build-info.json` con la fecha, un `.git` completo) invalida todo lo que viene después. Esto lo arregla el `.dockerignore`.
2. **Un `ARG` que cambia siempre.** Si declaras `ARG GIT_SHA` arriba del Dockerfile, todas las instrucciones posteriores dependen de un valor distinto en cada commit. Declara los argumentos volátiles lo más abajo posible, idealmente justo antes de la instrucción que los usa.
3. **El builder no persiste.** Un `docker buildx create` nuevo en cada job crea un builder vacío. La caché local vive en el builder; si lo tiras, la tiras.
4. **Plataformas distintas.** La caché es por plataforma. Un build de `linux/arm64` no reutiliza nada de un build de `linux/amd64`, y si construyes las dos en jobs separados con la misma clave de caché, se pisan.

Cuando no está claro cuál de las cuatro es, la herramienta correcta es el modo de depuración de BuildKit, que muestra el grafo de ejecución y marca qué pasos vinieron de caché:

```
BUILDKIT_PROGRESS=plain docker buildx build --no-cache-filter test . 2>&1 | grep -E
'CACHED|DONE'
```

El `--no-cache-filter` merece mención aparte: invalida la caché de una etapa concreta y respeta la del resto. Es lo que quieres cuando necesitas volver a ejecutar los tests sin reconstruir las dependencias.

El coste de la caché y cuándo limpiarla

Una caché de registro con `mode=max` crece. En un monorepo con seis imágenes y ramas activas, es normal ver decenas de gigabytes en el repositorio de caché al cabo de unos meses. Merece la pena una política explícita: expiración de las claves de rama a los siete días, la de la rama principal indefinida, y un límite de tamaño en el propio backend.

```
# Limpiar la caché local del builder, conservando lo usado en la última semana
docker buildx prune --filter 'until=168h' --keep-storage 20GB -f

# Configuración persistente del builder
cat <<'TOML' > buildkitd.toml
[worker.oci]
  gc = true
  [[worker.oci.gcpolicy]]
```

```
    keepBytes = "20GB"
    keepDuration = "168h"
TOML
docker buildx create --name ci --driver docker-container --config buildkitd.toml --use
```

Arranque, señales y salud: el contrato entre la imagen y el orquestador

Una imagen bien construida que se comporta mal en Kubernetes suele fallar en el mismo sitio: el contrato de arranque y parada. Son cuatro detalles pequeños con consecuencias grandes.

La forma exec no es una preferencia de estilo. `CMD python app.py` (forma shell) arranca `/bin/sh -c "python app.py"`. El PID 1 es el shell, y el shell no reenvía SIGTERM a su hijo. Cuando Kubernetes pide una parada limpia, tu proceso no se entera, agota el periodo de gracia y el kubelet lo mata con SIGKILL. Las peticiones en vuelo se pierden. `CMD ["python", "app.py"]` (forma exec) hace que tu proceso sea el PID 1 y reciba la señal directamente.

El PID 1 tiene responsabilidades. El proceso 1 de un espacio de nombres adopta los huérfanos y debe recogerlos. Si tu aplicación lanza subprocesos y no los espera, acumulas zombis. Si además ignoras las señales por defecto (el kernel no aplica las acciones por defecto al PID 1), un proceso que no maneje SIGTERM explícitamente lo ignorará. Cuando tu runtime no está preparado para ser PID 1, un init mínimo lo resuelve.

```
FROM python:3.13-slim
RUN apt-get update && apt-get install -y --no-install-recommends tini \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["python", "-m", "app"]
```

Si usas distroless, la alternativa sin instalar nada es `ENTRYPOINT ["/app", "--"]` con un binario Go que ya gestiona sus goroutines, o el paquete `dumb-init` copiado desde una etapa de build.

STOPSIGNAL cuando tu proceso no habla SIGTERM. Algunos servidores usan otra convención: nginx hace parada rápida con SIGTERM y parada elegante con SIGQUIT. Si no lo declaras, el orquestador envía SIGTERM y cortas conexiones a medias.

```
STOPSIGNAL SIGQUIT
```

HEALTHCHECK del Dockerfile frente a probes de Kubernetes. Son mecanismos distintos y Kubernetes ignora el primero por completo. La instrucción `HEALTHCHECK` sólo la interpreta el daemon de Docker y Compose. Si despliegas en Kubernetes, define las probes en el manifiesto; si además ejecutas la imagen en local con Compose, el `HEALTHCHECK` sigue siendo útil para que `depends_on: condition: service_healthy` funcione. Tener ambos no es duplicación: cubren entornos distintos.

```
HEALTHCHECK --interval=10s --timeout=2s --start-period=20s --retries=3 \
  CMD ["/app", "healthcheck"]
```

Fíjate en que el healthcheck invoca un subcomando del propio binario en lugar de `curl`. Eso evita tener que instalar curl en la imagen final sólo para comprobar la salud, que es una de las razones más habituales por las que una imagen distroless acaba convertida en una imagen con Debian entero dentro.

Variables de entorno del runtime que casi nadie configura

Un contenedor con un límite de memoria de 512 MiB no le dice nada a tu runtime salvo que se lo digas. La JVM, Node y Go leen la memoria de la máquina, no el cgroup, y descubren el límite cuando el kernel los mata.

```
# Go: el recolector de basura respeta este límite y hace GC antes del OOM
ENV GOMEMLIMIT=450MiB
ENV GOGC=100

# Node: el heap viejo por debajo del límite del contenedor
ENV NODE_OPTIONS="--max-old-space-size=384"

# Python: sin buffering para que los logs salgan al momento
ENV PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1
```

La aritmética importa: deja entre un 10% y un 20% de margen entre el límite del runtime y el límite del contenedor, porque el heap no es toda la memoria del proceso. Hay pilas de hilos, buffers de red, memoria del asignador y las propias bibliotecas nativas.

El registro también es infraestructura

Las imágenes se acumulan. Un pipeline que publica una etiqueta por commit genera varios miles de imágenes al año por servicio, y cada una arrastra sus capas. Si nadie define una política, el coste crece de forma lineal hasta que alguien pregunta por la factura.

Una política razonable tiene tres reglas: conservar indefinidamente las etiquetas de versión semántica y las que están desplegadas, conservar treinta días las etiquetas de rama, y borrar a los siete días las de pull request. Lo importante es la segunda parte: **borrar la etiqueta no libera espacio**. Las capas siguen ahí hasta que el recolector de basura del registro pasa, y en muchos registros gestionados eso ocurre en una ventana que no controlas.

```
{
  "rules": [
    {
      "rulePriority": 1,
      "description": "PRs: siete días",
      "selection": {
        "tagStatus": "tagged",
        "tagPrefixList": ["pr-"],
        "countType": "sinceImagePushed",
        "countUnit": "days",
        "countNumber": 7
      },
      "action": { "type": "expire" }
    },
    {
      "rulePriority": 2,
      "description": "Sin etiqueta: un día",
      "selection": {
        "tagStatus": "untagged",
        "countType": "sinceImagePushed",
        "countUnit": "days",
        "countNumber": 1
      },
      "action": { "type": "expire" }
```

```
    }
  ]
}
```

Hay una trampa concreta con las políticas de expiración y las imágenes multiplataforma: si la regla borra por antigüedad una imagen de arquitectura que todavía forma parte de un índice vigente, el índice queda roto y los pulls de esa arquitectura fallan con un error poco descriptivo. Las políticas que operan sobre imágenes sin etiqueta son las que más a menudo provocan esto, porque las imágenes que componen un índice no llevan etiqueta propia. Comprueba que tu registro entiende los índices antes de activar una regla agresiva sobre lo no etiquetado.

Caché de extracción y límites de descarga

Si tus nodos descargan imágenes públicas directamente de Docker Hub, estás a un pico de tráfico de encontrarte con errores de límite de descarga en mitad de un escalado. Un registro con caché de extracción (pull-through) resuelve las dos cosas a la vez: elimina la dependencia de la disponibilidad de un tercero y elimina el límite.

```
# containerd: redirigir docker.io al espejo interno
version = 2
[plugins."io.containerd.grpc.v1.cri".registry]
  config_path = "/etc/containerd/certs.d"
```

```
# /etc/containerd/certs.d/docker.io/hosts.toml
server = "https://registry-1.docker.io"

[host."https://registry.example.com/v2/dockerhub"]
  capabilities = ["pull", "resolve"]
```

Es media hora de trabajo y elimina una clase entera de incidentes cuya causa raíz siempre tarda en aparecer, porque el error se manifiesta como pods en `ImagePullBackOff` sin ninguna relación aparente con el cambio que se acaba de desplegar.

Caso práctico: de 14 minutos y 1,4 GB a 95 segundos y 180 MB

Los números de esta sección vienen de un servicio real: una API en Python con FastAPI, SQLAlchemy, un par de dependencias con extensiones compiladas y un frontend de administración construido con Vite. El punto de partida era el Dockerfile de siempre.

```
FROM python:3.13
WORKDIR /app
COPY . .
RUN apt-get update && apt-get install -y build-essential libpq-dev nodejs npm
RUN pip install -r requirements.txt
RUN cd frontend && npm install && npm run build
EXPOSE 8000
CMD uvicorn app.main:app --host 0.0.0.0 --port 8000
```

Medición inicial: 14 minutos 20 segundos de build en CI en frío, 11 minutos 40 segundos en caliente (la caché casi nunca acertaba), 1,41 GB descomprimidos, 512 MB comprimidos, 287 CVE de los que 9 eran críticos, y 38 segundos de pull en un nodo nuevo.

Paso 1: `.dockerignore` y orden de copia. El `COPY . .` arriba del todo significaba que cualquier

cambio en cualquier fichero invalidaba la instalación de dependencias. Añadir un `.dockerignore` (que excluía `.git`, `node_modules`, `.venv`, `__pycache__` y los ficheros de test) y copiar primero los manifiestos bajó el build en caliente a 3 minutos 10 segundos. Media hora de trabajo, 73% menos de tiempo. Esta es siempre la primera optimización porque es la de mejor relación entre esfuerzo y resultado.

Paso 2: multi-etapa. Separar la construcción del frontend, la instalación de dependencias de Python y el runtime en tres etapas eliminó de la imagen final `build-essential`, Node, npm y las cabeceras de desarrollo. 1,41 GB pasaron a 340 MB. Los CVE bajaron de 287 a 61, casi todos por la desaparición del toolchain de compilación.

Paso 3: cache mounts y uv. Sustituir `pip install` por `uv sync` con un cache mount para el directorio de caché de uv, y `npm install` por `pnpm install --frozen-lockfile` con su propio cache mount, bajó el build en frío de 14 minutos a 4 minutos 5 segundos y el build en caliente a 95 segundos. El cambio con más impacto de toda la lista sobre el build en frío, que es el que sufres cuando el runner es nuevo.

```
FROM node:22-alpine AS frontend
WORKDIR /fe
RUN corepack enable
COPY frontend/package.json frontend/pnpm-lock.yaml ./
RUN --mount=type=cache,target=/root/.local/share/pnpm/store \
    pnpm install --frozen-lockfile
COPY frontend/ ./
RUN pnpm run build

FROM python:3.13-slim AS deps
COPY --from=ghcr.io/astral-sh/uv:0.9 /uv /usr/local/bin/uv
WORKDIR /app
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy
RUN apt-get update && apt-get install -y --no-install-recommends \
      build-essential libpq-dev && rm -rf /var/lib/apt/lists/*
COPY pyproject.toml uv.lock ./
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev --no-install-project

FROM python:3.13-slim AS runtime
RUN apt-get update && apt-get install -y --no-install-recommends libpq5 \
    && rm -rf /var/lib/apt/lists/* \
    && useradd --uid 10001 --no-create-home --shell /usr/sbin/nologin app
WORKDIR /app
COPY --from=deps /app/.venv /app/.venv
COPY --from=frontend /fe/dist ./static
COPY --chown=10001:10001 app/ ./app/
ENV PATH="/app/.venv/bin:$PATH" PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1
USER 10001:10001
EXPOSE 8000
ENTRYPOINT ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Paso 4: base más delgada en el runtime. El paso de `python:3.13` (1,02 GB) a `python:3.13-slim` (154 MB) fue lo que llevó la imagen final de 340 MB a 180 MB. Se intentó también `python:3.13-alpine` y se descartó: las dependencias con extensiones compiladas no tenían ruedas para musl y había que compilarlas, lo que devolvía el build en frío a más de siete minutos para ahorrar 30 MB. Es el compromiso clásico de Alpine con Python y casi nunca sale a cuenta.

Paso 5: caché de registro en CI. Los runners de CI son efímeros, así que la caché local no sobrevivía entre ejecuciones. Con `--cache-to type=registry,mode=max`, el build de un pull request típico pasó

a 95 segundos de forma consistente en lugar de depender de la suerte.

Paso 6: procedencia, SBOM y firma. Añadir `--provenance=mode=max --sbom=true` y la firma con cosign sumó 11 segundos al pipeline. Ese es el coste real de la parte de cadena de suministro, y es la razón por la que no hay ningún argumento razonable para dejarla para después.

Resumen de la migración

- **Build en frío**: 14 min 20 s !’ 4 min 05 s
- **Build en caliente**: 11 min 40 s !’ 95 s
- **Tamaño descomprimido**: 1,41 GB !’ 180 MB
- **Tamaño comprimido**: 512 MB !’ 68 MB
- **Pull en un nodo nuevo**: 38 s !’ 6 s
- **CVE de severidad alta y crítica**: 9 críticos y 41 altos !’ 0 críticos y 3 altos
- **Capas que cambian en un despliegue**: todas !’ una sola, la del código, de 2,1 MB

La fila más interesante es la última. Antes, cada despliegue movía 512 MB por nodo porque el `COPY . .` invalidaba todo. Después, un despliegue normal mueve 2,1 MB. Eso no aparece en ninguna comparativa de tamaños de imagen y es, con diferencia, lo que más se nota en la velocidad de despliegue diario.

Coste total del trabajo: unas nueve horas repartidas en tres sesiones. El paso 1 se hizo en la primera media hora y ya justificaba el resto.

Construir sin demonio: rootless, Kubernetes y builders remotos

Todo lo anterior asume que el build ocurre en tu portátil. En producción casi nunca es así: ocurre en un runner de CI que muy a menudo vive dentro de Kubernetes, y ahí la conversación cambia de tema. Montar `/var/run/docker.sock` en el pod que construye es lo primero que encuentra cualquiera en un buscador, y es también la forma más rápida de convertir un pull request en acceso root al nodo: quien habla con el socket puede lanzar un contenedor privilegiado con el sistema de ficheros del host montado y leer los secretos de los demás pods. Si tu pipeline construye código que aún nadie ha revisado —y eso es exactamente lo que hace un pipeline de pull requests— el socket no es una opción defendible.

La respuesta habitual durante años fue Kaniko, que construía imágenes sin demonio desde dentro de un contenedor. Google archivó el repositorio en junio de 2025: sigue funcionando, pero ya no recibe correcciones de errores ni de CVE, y varios proyectos han retirado su documentación al respecto. Hay forks mantenidos por terceros, aunque construir tu cadena de suministro sobre un fork de un proyecto abandonado es una decisión que conviene tomar a propósito y no por inercia. Las dos alternativas con mantenimiento real son BuildKit en modo rootless y Buildah.

BuildKit tiene una ventaja concreta aquí: es el mismo motor que ya usas en local, así que el Dockerfile, los cache mounts, los secretos de build y `bake` se comportan igual. Buildx sabe arrancar el builder dentro del clúster:

```
# Un builder que corre en el cluster, sin privilegios
docker buildx create \
  --name k8s-builder \
  --driver kubernetes \
  --driver-opt namespace=ci,replicas=2,rootless=true \
  --driver-opt requests.cpu=2,requests.memory=4Gi,limits.memory=8Gi \
  --use
```

```
# El build se ejecuta en los pods, no en el runner
docker buildx build \
  --cache-from type=registry,ref=registry.example.com/app:buildcache \
  --cache-to   type=registry,ref=registry.example.com/app:buildcache,mode=max \
  --push -t registry.example.com/app:$GIT_SHA .
```

La opción `rootless=true` crea los pods sin `securityContext.privileged`, apoyándose en espacios de nombres de usuario. Dos detalles que se descubren tarde: en algunas distribuciones hace falta relajar el perfil de seccomp y AppArmor del pod para que el *userns* funcione, y el overlayfs sin privilegios requiere un kernel razonablemente moderno; si no está disponible, BuildKit cae a fuse-overlayfs, que funciona pero es notablemente más lento escribiendo capas grandes. Merece la pena comprobar en los logs del builder cuál de los dos está usando antes de culpar al Dockerfile de un build lento.

El problema serio de un builder efímero es la caché. Un pod que nace y muere con el build empieza siempre en frío, con lo que pierdes justo lo que hace rápido a BuildKit. Hay tres salidas, y no son excluyentes. La primera es la caché de registro con `mode=max`, que ya vimos: sobrevive al pod porque no vive en el pod. La segunda es dar al builder un volumen persistente y rutar los builds del mismo repositorio siempre al mismo réplica, para que la caché local acierte; funciona, pero introduce afinidad y un estado que hay que operar. La tercera es dejar de tener builders efímeros: un builder remoto de vida larga (el driver `remote` apuntando a un buildkitd propio, o un servicio gestionado tipo Docker Build Cloud) mantiene la caché caliente entre builds y, si además ofrece máquinas ARM nativas, elimina de golpe el problema de QEMU que vimos en la sección de multiplataforma.

Buildah es la alternativa cuando ya vives en el ecosistema de Podman: construye sin demonio por diseño, entiende Dockerfiles y permite además escribir el build como un script de shell paso a paso, lo que resulta cómodo para imágenes que se generan a partir de lógica y no de un fichero estático. Lo que no obtienes es la paridad exacta con BuildKit: los cache mounts y la caché de registro con `mode=max` son territorio de BuildKit, y si tu Dockerfile depende de ellos la migración no es un cambio de binario.

Cuándo no escribir un Dockerfile: buildpacks, ko y jib

Hay una pregunta que casi nunca se hace en voz alta: ¿de verdad necesitas un Dockerfile? Mantener uno bien hecho por servicio tiene un coste recurrente —cada vez que sale una versión del runtime, cada vez que aparece un CVE en la base, cada vez que alguien copia el Dockerfile de hace tres años a un servicio nuevo— y en organizaciones con muchos repositorios ese coste se multiplica por el número de equipos. Las tres herramientas que evitan el Dockerfile no son intercambiables; cada una resuelve un caso distinto.

Cloud Native Buildpacks invierte el planteamiento: en lugar de describir cómo se construye la imagen, describes nada y el *builder* detecta el lenguaje, instala el runtime adecuado, compila y produce una imagen con usuario no root, etiquetas OCI y SBOM ya incluidos. Es convención frente a configuración, con la ventaja operativa de que la convención la mantiene otro:

```
# Sin Dockerfile: el builder decide como se construye
pack build registry.example.com/app:$GIT_SHA \
  --builder paketobuildpacks/builder-jammy-base \
  --env BP_JVM_VERSION=21 \
  --publish

# Actualizar solo el sistema base de cientos de imagenes,
# sin recompilar la aplicacion: se reescriben las capas de abajo
pack rebase registry.example.com/app:$GIT_SHA --publish
```

Ese `pack rebase` es el argumento fuerte y el que casi nadie menciona. Cuando aparece un CVE en la imagen base, un parque de Dockerfiles obliga a reconstruir cada aplicación, con lo que eso implica de pipelines, tests y riesgo de que algo se mueva. Con buildpacks, las capas de la aplicación y las del sistema están separadas por contrato, así que puedes cambiar el sistema por debajo y publicar una imagen nueva en segundos, sin tocar el código compilado. A cambio: los builds consumen más memoria y suelen ser más lentos que un Dockerfile bien cacheado, las imágenes resultantes son más grandes que una distroless afinada a mano, y cuando necesitas algo que la convención no contempla —una librería nativa concreta, un paso de build raro— acabas escribiendo un buildpack, que es más trabajo que escribir el Dockerfile que querías evitar.

ko es el caso más limpio de todos, y sólo sirve para Go. Compila el binario en tu máquina o en el runner, lo coloca sobre una base estática mínima y sube la imagen directamente al registro, sin demonio de Docker y sin Dockerfile:

```
export KO_DOCKER_REPO=registry.example.com/app
ko build ./cmd/api --bare --sbom=spdx --platform=linux/amd64,linux/arm64

# Devuelve la referencia por digest, lista para el manifiesto
# registry.example.com/app@sha256:...
```

Para un servicio en Go, ko elimina de un golpe la mitad de este artículo: no hay caché de capas que optimizar porque no hay capas de dependencias, no hay QEMU porque el compilador de Go cruza-compila de forma nativa, y la salida es una referencia por digest que puedes inyectar directamente en el manifiesto. El límite es obvio: si tu binario necesita cgo, certificados extra, ficheros de configuración o una zona horaria, ya estás peleando con las opciones de ko y probablemente estabas mejor con un multi-stage de tres líneas.

jib hace lo equivalente para la JVM desde Maven o Gradle: separa dependencias, recursos y clases en capas distintas —que es justo el orden de frecuencia de cambio correcto—, usa una base distroless por defecto y no necesita demonio. En un monorepo Java con decenas de módulos, la diferencia frente a un Dockerfile por módulo es difícil de exagerar.

El criterio práctico, resumido: si tienes muchos servicios en pocos lenguajes y quieres centralizar el mantenimiento del sistema base, buildpacks. Si es Go, ko. Si es JVM, jib. Si tienes un servicio con requisitos poco convencionales, o necesitas control fino sobre lo que entra en la imagen final por motivos de tamaño o de auditoría, el Dockerfile multi-etapa sigue ganando —y ahora ya sabes escribirlo bien.

Dependencias privadas y builds que no salen a internet

El Dockerfile de ejemplo de cualquier tutorial instala dependencias públicas. El de tu empresa, casi seguro, instala al menos un paquete de un índice privado, y ahí es donde aparecen las credenciales dentro del build. Ya vimos por qué `ARG` es la peor opción posible: queda escrito en `docker history` para siempre y viaja con la imagen. La forma correcta es un secreto montado, que existe sólo durante la ejecución de ese `RUN` y no deja capa:

```
# syntax=docker/dockerfile:1.7
FROM python:3.13-slim AS deps
WORKDIR /app
COPY pyproject.toml uv.lock ./
# El .netrc existe durante este RUN y en ningun otro sitio
RUN --mount=type=secret,id=netrc,target=/root/.netrc,mode=0400 \
    --mount=type=cache,target=/root/.cache/uv \
```

```
    uv sync --frozen --no-dev --no-install-project
```

```
docker buildx build --secret id=netrc,src=$HOME/.netrc -t app:dev .

# En CI, desde una variable de entorno en lugar de un fichero
docker buildx build --secret id=netrc,env=NETRC_CONTENT -t app:ci .
```

El patrón es el mismo para los demás ecosistemas, cambiando el fichero de destino: `/root/.npmrc` para npm y pnpm, `/root/.m2/settings.xml` para Maven, `/root/.config/pip/pip.conf` si prefieres configurar pip por fichero. Una precaución que se olvida: si el gestor de paquetes escribe la URL con credenciales dentro del lockfile o de un fichero de caché que luego copias a la etapa final, el secreto se filtra por la puerta de atrás. Conviene revisar con `grep` la imagen resultante la primera vez que montas esto, y no la décima.

Go tiene su propia trampa. `GOPRIVATE` le dice al toolchain que ciertos módulos no pasan por el proxy público ni por la base de datos de sumas de comprobación, lo cual es necesario para un repositorio interno. El atajo tentador es `GOPRIVATE=*`, y con eso acabas de desactivar la verificación criptográfica de *todas* tus dependencias, incluidas las públicas. Lo correcto es enumerar los prefijos reales:

```
# Bien: solo lo interno se salta el proxy y la suma de comprobacion
ENV GOPRIVATE=github.com/acme/*,gitlab.acme.internal/*
ENV GOFLAGS=-mod=readonly

# Mal: desactiva sum.golang.org para todo el arbol de dependencias
# ENV GOPRIVATE=*
```

Probar que un paso no necesita red

Un build que descarga cosas en pasos donde no debería es un build no reproducible esperando a que el índice remoto tenga un mal día. BuildKit permite afirmarlo de forma comprobable con `--network=none`, que ejecuta ese `RUN` sin interfaz de red. Si el paso funcionaba porque en realidad se descargaba algo, falla en el momento en que lo declaras hermético, que es exactamente cuando quieres enterarte:

```
FROM deps AS build
COPY . .
# Compilar no necesita internet. Si lo necesita, quiero saberlo ahora.
RUN --network=none python -m compileall -q src/
```

La otra mitad del problema es de infraestructura y no de Dockerfile. Un build que depende de PyPI o del registro público de npm depende de la disponibilidad de un tercero para poder desplegar un arreglo urgente. Un espejo interno con caché de extracción —Artifactory, Nexus, devpi, Verdaccio, o el proxy de tu proveedor de nube— convierte esa dependencia externa en una interna y, de paso, te da un punto único donde aplicar el periodo de enfriamiento de versiones y el bloqueo de paquetes retirados. En entornos aislados de verdad es el único camino: el índice interno se sincroniza desde fuera con una política explícita, y los builds nunca tienen ruta a internet.

Etiquetas, promoción por digest y el ciclo de vida de la imagen base

Una etiqueta de Docker es un puntero mutable. Esa frase, que parece un detalle de implementación, es la causa de una familia entera de incidentes: el pod que se reinicia y arranca con un código distinto al de

sus hermanos, el *rollback* a `v1.4.2` que trae el binario equivocado porque alguien reescribió la etiqueta, la imagen que en staging funcionaba y en producción no siendo *la misma etiqueta*. Un digest, en cambio, es el hash del manifiesto: si el digest coincide, el contenido coincide, y no hay conversación posible.

De ahí sale una regla operativa sencilla: **construye una vez, etiqueta para las personas, despliega por digest**. Las etiquetas sirven para que un humano encuentre la imagen (`v1.4.2`, `main-a3f9c1`, `staging`); el manifiesto que aplica el orquestador lleva el digest. Y la promoción entre entornos no reconstruye nada: mueve la referencia.

```
# Promocionar sin reconstruir: es exactamente el mismo artefacto
DIGEST=$(crane digest registry.example.com/app:main-$GIT_SHA)
crane tag registry.example.com/app@$DIGEST staging

# Y despues, tras las pruebas, a produccion
crane tag registry.example.com/app@$DIGEST v1.4.2

# El manifiesto referencia el digest, no la etiqueta
# image: registry.example.com/app@sha256:9f2e...
```

Reconstruir para promocionar es el anti-patrón que anula todas las pruebas que acabas de pasar: la imagen que va a producción no es la que validaste, aunque el commit sea el mismo, porque las dependencias transitivas y la imagen base pueden haber cambiado entre las dos ejecuciones. Si además el registro soporta etiquetas inmutables —ECR, Artifact Registry y Harbor lo hacen— actívalas para las etiquetas de versión y deja mutables sólo los punteros de conveniencia tipo `staging`. Sobrescribir `v1.4.2` debería ser un error del registro, no una decisión de quien ejecuta el `push`.

La imagen base envejece aunque tu código no cambie

Fijar la base por digest, como vimos, hace el build reproducible. También congela los CVE de esa base: dentro de tres meses seguirás desplegando el mismo `sha256:` con los parches que existían el día que lo fijaste. Reproducibilidad y frescura tiran en direcciones opuestas, y la resolución no es elegir una, sino automatizar el avance del digest para que sea un cambio revisable y no un efecto secundario invisible.

```
{
  "extends": ["config:recommended", "docker:pinDigests"],
  "packageRules": [
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["digest", "pin"],
      "automerge": true,
      "automergeType": "branch"
    },
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["minor", "patch"],
      "minimumReleaseAge": "3 days"
    },
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["major"],
      "dependencyDashboardApproval": true
    }
  ]
}
```

Lo que hace esta configuración: el preset `docker:pinDigests` convierte `FROM python:3.13-slim` en `FROM python:3.13-slim@sha256:...` y mantiene el comentario con la etiqueta legible al lado. Las

actualizaciones de digest —mismo `3.13-slim`, contenido nuevo— se automezclan, porque son precisamente los parches de seguridad de la base y bloquearlas detrás de una revisión humana significa no aplicarlas nunca. Las subidas de versión menor esperan tres días de enfriamiento, y las mayores requieren aprobación explícita, porque un salto de `3.13` a `3.14` no es un parche.

Falta una pieza que ni Renovate ni Dependabot cubren: una imagen ya publicada y desplegada acumula vulnerabilidades sin que nadie toque el repositorio. La respuesta es una reconstrucción periódica del mismo commit con la base actualizada —un *cron* nocturno o semanal que reconstruye las etiquetas activas y las vuelve a escanear— combinada con el reescaneo diario de lo desplegado que ya mencionamos. Si la reconstrucción es rápida y la promoción es por digest, esto cuesta poco; si el build tarda catorce minutos, no lo harás. Es otra forma de decir que la velocidad del build no es una comodidad de quien desarrolla, sino una propiedad de seguridad del sistema.

Arranque en frío: por qué el tamaño no es toda la historia

Reducir la imagen de 1,4 GB a 180 MB mejora el arranque, pero la relación no es lineal y conviene entender por qué antes de seguir recortando megabytes con rendimientos decrecientes. El estudio clásico de Harter y sus colegas midió que la descarga de la imagen supone alrededor del 76% del tiempo de arranque de un contenedor, mientras que sólo una fracción pequeña de los ficheros de la imagen se lee realmente durante ese arranque. Es decir: pasas la mayor parte del tiempo trayendo datos que nunca se van a abrir.

Antes de recurrir a nada exótico, hay tres palancas que casi siempre están sin usar. La primera es la proximidad del registro: extraer desde otra región es un coste fijo por capa y por nodo que se paga en cada escalado, y una caché de extracción local al clúster lo elimina. La segunda es el paralelismo de descarga del kubelet, que históricamente serializaba los pulls; desde Kubernetes 1.27 puedes activar descargas paralelas y acotarlas para no saturar la red del nodo:

```
# KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
serializeImagePulls: false
maxParallelImagePulls: 5
imagePullProgressDeadline: 5m
```

La tercera es medir de verdad en lugar de suponer. El tiempo de descarga aparece en los eventos del pod y, de forma agregada, en las métricas del kubelet:

```
# Cuanto tarda de verdad la descarga en tus nodos, p99
histogram_quantile(0.99,
  sum by (le, node) (
    rate(kubelet_runtime_operations_duration_seconds_bucket{
      operation_type="pull_image"
    }[30m])
  )
)

# Y el detalle de un pod concreto
# kubectl describe pod api-7f9c | grep -A2 Pulled
```

Cuando esas tres palancas ya están tiradas y sigues teniendo un problema —típicamente con imágenes de aprendizaje automático, que arrastran CUDA y pesan varios gigabytes por motivos que no se resuelven con un multi-stage— entra el *lazy pulling*. La idea es dejar de tratar la imagen como un archivo

que hay que descargar entero antes de arrancar y empezar a tratarla como un sistema de ficheros remoto del que se traen bloques a demanda. Hay dos implementaciones maduras y la diferencia entre ellas es práctica, no filosófica.

eStargz, del proyecto stargz-snapshotter de containerd, define un formato de imagen lazily-pullable: hay que convertir la imagen, y el resultado sigue siendo compatible con registros OCI estándar. **SOCI** (Seekable OCI), de AWS, evita la conversión: genera un índice aparte que se sube junto a la imagen y permite extraer ficheros individuales de una capa tar sin descargarla completa. Si controlas el pipeline de build, cualquiera de las dos vale; si consumes imágenes de terceros que no puedes reconstruir, SOCI es la que encaja. La contrapartida es común a ambas: el trabajo de entrada/salida no desaparece, se traslada del arranque al tiempo de ejecución, así que una aplicación que recorre medio sistema de ficheros en su primer minuto puede acabar más lenta en total. En las mediciones publicadas, SOCI tiende a mantener el tiempo de arranque de la aplicación más cerca del original que eStargz.

Los números gestionados dan una idea del orden de magnitud: el Image Streaming de GKE, que es esta misma técnica ofrecida como servicio, redujo el arranque de una imagen de Triton de 5,4 GB de 191 segundos a 30. Es la clase de mejora que no se consigue optimizando el Dockerfile, porque el problema no era el Dockerfile.

La opción aburrida que funciona

Si el lazy pulling te parece mucha maquinaria para el problema que tienes —y muchas veces lo es— existe una alternativa sin componentes nuevos: precargar la imagen en los nodos. Un DaemonSet que hace `pull` de la imagen que va a desplegarse, o directamente incluirla en la AMI o en la imagen de nodo, convierte el arranque en frío en un arranque en caliente. Con `imagePullPolicy: IfNotPresent` y referencias por digest, no hay ambigüedad sobre qué se ejecuta:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: image-prepuller
spec:
  selector:
    matchLabels: { app: image-prepuller }
  template:
    metadata:
      labels: { app: image-prepuller }
    spec:
      initContainers:
        - name: prepull-api
          image: registry.example.com/app@sha256:9f2e...
          command: ["/bin/true"]
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.10
          resources:
            requests: { cpu: 10m, memory: 16Mi }
```

El coste es disco en los nodos y una política de limpieza que hay que operar (el recolector de imágenes del kubelet borra por umbral de uso de disco, y puede llevarse justo la imagen que precargaste). Pero no introduce un snapshotter nuevo en la ruta crítica de arranque de todos tus contenedores, y para un servicio con un puñado de imágenes calientes suele ser la respuesta correcta.

Cuatro errores más, de los que sólo se ven en producción

Además de los ocho anteriores, hay cuatro que aparecen cuando la imagen ya lleva tiempo funcionando.

Nueve. Confiar en `latest` en cualquier sitio. No sólo en el `FROM`: también en el manifiesto de Kubernetes. Una etiqueta móvil en un `Deployment` significa que dos réplicas del mismo despliegue pueden estar corriendo imágenes distintas si un pod se reinicia después de que alguien reetiquete. Los síntomas son un comportamiento no reproducible que desaparece al reiniciar y vuelve tres días después. Usa digests, y si eso complica tu flujo, al menos etiquetas inmutables con el SHA del commit.

Diez. Reconstruir la imagen para cada entorno. Si tu pipeline construye una imagen para staging y otra para producción con distintos argumentos de build, lo que despliegas en producción nunca ha sido probado. La configuración va en el entorno, no en la imagen. El mismo digest debe recorrer todos los entornos; lo único que cambia son las variables y los secretos que se montan.

Once. Escribir en el sistema de ficheros del contenedor. Funciona hasta que activas `readOnlyRootFilesystem`, o hasta que descubres que los logs que tu aplicación escribe en `/app/logs` desaparecen en cada reinicio y llenan el disco del nodo mientras tanto. Todo lo que la aplicación escriba debe ir a un volumen explícito o a la salida estándar. Declara `tmpfs` para lo temporal:

```
        securityContext:
          readOnlyRootFilesystem: true
          allowPrivilegeEscalation: false
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop: ["ALL"]
        volumeMounts:
          - name: tmp
            mountPath: /tmp
      volumes:
        - name: tmp
          emptyDir:
            medium: Memory
            sizeLimit: 64Mi
```

Doce. Un usuario con nombre en lugar de un UID numérico. `USER app` parece más legible que `USER 10001`, pero Kubernetes no puede resolver nombres de usuario: mira el campo `User` de la configuración de la imagen y, si contiene texto en lugar de un número, la comprobación de `runAsNonRoot` falla y el pod no arranca. Es un fallo que sólo aparece cuando alguien endurece el clúster, meses después de que la imagen se escribiera.

Preguntas que aparecen al aplicar todo esto

¿Merece la pena distroless si mi equipo no tiene experiencia con contenedores? Probablemente no como primer paso. El orden que funciona es: `.dockerignore` y orden de copia, multi-etapa, base *slim*, y sólo entonces distroless. Cada escalón añade menos beneficio y más fricción operativa que el anterior. Un equipo que aún no tiene multi-etapa no va a sacar partido de distroless y sí va a sufrir el primer incidente en el que no pueda entrar al contenedor.

¿Y si mi aplicación necesita ejecutar comandos del sistema? Entonces distroless no es para ti, y está bien. Distroless quita el shell y las utilidades a propósito; si tu aplicación llama a `git`, a `ffmpeg` o a

`pg_dump`, necesitas una base con paquetes. La opción intermedia razonable es una imagen tipo Wolfi, que sí tiene gestor de paquetes y a la vez se reconstruye continuamente, así que instalas exactamente lo que necesitas sobre una base con muy pocos CVE.

¿Cuántas etapas son demasiadas? No hay límite técnico relevante, pero hay uno práctico: las etapas que nadie usa no se construyen, así que el coste de una etapa extra es cero salvo que forme parte del camino a la imagen final. El problema no es el número sino la legibilidad. Si el Dockerfile pasa de unas 80 líneas, suele ser señal de que hay lógica que debería estar en un script versionado aparte y llamada desde una sola instrucción `RUN`.

¿Debo fijar las versiones de los paquetes del sistema? Es un compromiso incómodo. Fijar `apt-get install nginx=1.26.2-1` hace el build reproducible pero también hace que falle el día que Debian retire esa versión del repositorio, que en la práctica es cuestión de semanas para las actualizaciones de seguridad. La postura razonable para la mayoría de equipos es fijar la imagen base por digest (que ya congela el conjunto de paquetes) y no fijar los paquetes individuales, reconstruyendo con una cadencia conocida para recoger los parches.

¿Cómo mantengo actualizados los digests si lo fijo todo? Con la misma herramienta que ya usas para las dependencias del código. Renovate y Dependabot entienden las líneas `FROM` con digest y abren un pull request cuando la etiqueta apunta a un digest nuevo. La clave es el comentario con la etiqueta legible al final de la línea, que es lo que les permite saber qué están siguiendo:

```
FROM python:3.13-slim@sha256:1e4b6f2a9c... # python:3.13-slim
```

¿Los builds reproducibles son alcanzables de verdad? Bit a bit, en un caso general, no del todo: hay fuentes de no determinismo fuera de tu control (orden del sistema de ficheros, aleatoriedad en algunos compiladores, timestamps de paquetes descargados). Lo que sí es alcanzable y útil es la reproducibilidad práctica: fijar la base por digest, fijar las dependencias por lockfile con hashes, normalizar los timestamps con `SOURCE_DATE_EPOCH` y comprobar que dos builds consecutivos del mismo commit producen el mismo digest. Cuando eso deja de cumplirse, tienes una señal fiable de que algo cambió sin que nadie lo declarara.

¿Y Podman, o los builds sin daemon? Todo lo de esta guía sobre estructura del Dockerfile, capas, caché y elección de base aplica igual. Lo que cambia es la herramienta que ejecuta el build. Buildah y Kaniko construyen sin daemon privilegiado, lo que es relevante si construyes dentro del propio clúster. Kaniko tiene un modelo de caché distinto (por capa, en un registro) y no soporta la sintaxis de cache mounts de BuildKit, así que si dependes mucho de ellos la migración no es transparente.

Glosario

- **Capa**: conjunto de cambios en el sistema de ficheros producido por una instrucción del Dockerfile. Es la unidad de caché, de transferencia y de almacenamiento compartido.
- **Digest**: hash SHA-256 del manifiesto de una imagen. Identifica contenido exacto e inmutable, a diferencia de una etiqueta.
- **Índice (manifest list)**: manifiesto que agrupa varias imágenes, típicamente una por arquitectura, bajo una sola referencia.
- **Cache mount**: directorio que BuildKit conserva entre builds y que no forma parte de ninguna capa. Ideal para cachés de gestores de paquetes.
- **Atestación**: documento firmado y asociado a un digest que afirma algo sobre la imagen. Las dos más comunes son la SBOM (qué contiene) y la procedencia SLSA (cómo se construyó).

- **SBOM**: inventario de componentes de la imagen, con nombre, versión y licencia de cada paquete.
- **Procedencia SLSA**: descripción verificable del proceso de build: qué fuente, qué parámetros, qué constructor.
- **VEX**: declaración firmada sobre si un CVE concreto afecta o no a un producto concreto, con una justificación de un vocabulario cerrado.
- **Distroless**: familia de imágenes base que contienen las bibliotecas de runtime y nada más: sin shell, sin gestor de paquetes, sin utilidades.
- **Firma keyless**: firma basada en un certificado efímero emitido contra una identidad OIDC y registrada en un log de transparencia, sin clave privada persistente.
- **Pull-through cache**: registro que actúa de espejo bajo demanda de un registro externo, cacheando lo que ya se descargó.

Ocho errores que se repiten

1. **Copiar el código antes de instalar dependencias.** El error más caro y el más fácil de arreglar. Invalida la capa de dependencias en cada commit.
2. **Creer que la caché de CI funciona porque el log no da error.** Sin `mode=max` las etapas de build no se cachean. Compara el tiempo real de dos ejecuciones seguidas sin cambios antes de darlo por bueno.
3. **Borrar un secreto con RUN rm.** La capa anterior sigue en la imagen. Usa `--mount=type=secret` o multi-etapa.
4. **Alpine con Python.** Sin wheels manylinux, tu build compila NumPy desde fuente. Mide antes de migrar.
5. **Encadenar apt-get install sin limpiar en el mismo RUN.** `rm -rf /var/lib/apt/lists/*` en un `RUN` posterior no reduce nada; tiene que ir en la misma instrucción.
6. **USER con nombre en vez de UID.** Rompe `runAsNonRoot` en Kubernetes con un error confuso en tiempo de admisión.
7. **CMD en forma shell.** SIGTERM no llega a tu proceso y cada despliegue pierde peticiones en vuelo.
8. **Un solo tag mutable como `:latest`.** Imposible saber qué está corriendo y imposible hacer rollback determinista. Etiqueta por SHA de commit y despliega por digest.

Checklist de producción

- `.dockerignore` presente y verificado con `docker build --progress=plain` (mira el tamaño del contexto transferido).
- Multi-etapa con la etapa final basada en `-slim` o distroless.
- Manifiestos de dependencias copiados antes que el código fuente.
- `--mount=type=cache` en cada instalación de paquetes.
- Secretos por `--mount=type=secret`, nunca por `ARG`.
- Caché de registro con `mode=max` en CI, o builder persistente.
- Imagen base fijada por digest y actualizada automáticamente.
- `USER` numérico, `ENTRYPOINT`/`CMD` en forma exec, sistema de ficheros raíz de solo lectura.
- Procedencia y SBOM adjuntas; escaneo con Trivy o Grype en el pipeline con umbral que rompe el build.
- Tags inmutables por SHA; despliegue por digest.

Preguntas frecuentes

¿Sigue teniendo sentido reducir el número de capas? Muy poco. Con BuildKit y compresión zstd el coste de una capa extra es despreciable, y agrupar comandos que cambian a ritmos distintos empeora la caché. La excepción real es limpiar dentro del mismo `RUN` que ensucia.

¿Distroless me deja sin depuración? No del todo. Los tags `:debug` traen un shell mínimo, y en Kubernetes `kubectl debug --image=busybox --target=api` te da un contenedor efímero que comparte el espacio de nombres de proceso sin tocar la imagen de producción.

¿Squash o exportar la imagen aplanada? Casi nunca merece la pena: destruye la caché y el compartir capas entre imágenes, que es de donde sale el ahorro real en el registro y en el pull.

¿Y las builds multiarquitectura? `docker buildx build --platform linux/amd64,linux/arm64` funciona, pero por emulación QEMU es lentísimo para lenguajes compilados. Para builds serios usa nodos nativos por arquitectura en el mismo builder, o cross-compilación con `TARGETPLATFORM` y `BUILDPLATFORM`.

Por dónde empezar mañana

Si solo puedes hacer una cosa, reordena el Dockerfile para que los manifiestos se copien antes que el código y añade un `.dockerignore`. Es media hora de trabajo y suele recortar el build a la mitad. Con eso funcionando, añade los cache mounts, después la etapa final ligera, y deja procedencia y fijación por digest para cuando el pipeline ya sea rápido. Optimizar en ese orden significa que cada paso se paga solo antes de empezar el siguiente.

ENGLISH VERSION

Docker Images in Production: Multi-stage Builds, BuildKit Caching, Distroless and Reproducible Builds

The problem: the image nobody looks at until it hurts

Nearly every project starts the same way. Someone writes a seven-line Dockerfile, it works locally, and there it sits for two years. The cost shows up slowly and in pieces: the pipeline takes fourteen minutes because every commit reinstalls dependencies from scratch, the vulnerability scanner reports three hundred CVEs that come from the base system rather than your code, autoscaling takes forty seconds to bring up a pod because there is 1.4 GB to pull, and the registry bill grows without anyone quite knowing why.

None of that is fixed by a trick. It is fixed by understanding three things: how layer caching works, what a multi-stage build actually does, and what is inside the base image you picked without thinking. This guide covers all three, with complete Dockerfiles for Python, Node and Go that you can copy today.

Layers and caching: the one rule that matters

A container image is a stack of read-only layers. Every Dockerfile instruction that changes the filesystem produces a layer. During a build, BuildKit computes a cache key per instruction; if the key matches a previous layer, it reuses it and runs nothing.

For `RUN` the key is the literal text of the command plus the key of the previous layer. For `COPY` and `ADD` it is the content of the copied files. Hence the practical consequence: **one invalidated layer invalidates every layer after it**. If you copy your source code before installing dependencies, a one-line change in a Python file throws away the entire package install.

```
# Bad: changing any file reinstalls everything
COPY . .
RUN pip install -r requirements.txt

# Good: the manifest rarely changes, the code always does
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

The general rule: order the Dockerfile from what changes least to what changes most. Dependency manifests first, code afterwards. It is a two-line change that usually cuts more CI time than any other optimization.

.dockerignore is not optional

The build context is shipped in full to the daemon before anything runs. Without a `.dockerignore`, your `.git` directory, your local `node_modules` and your dataset folder travel on every build, and they also poison the cache key of `COPY . .`: a single commit invalidates it even though you did not touch the code.

```
.git
.gitignore
node_modules
__pycache__
*.pyc
.venv
.env
.env.*
dist
build
coverage
.pytest_cache
.mypy_cache
Dockerfile*
docker-compose*.yml
README.md
tests/fixtures/large/
```

Multi-stage: separating what builds from what runs

A multi-stage build uses several `FROM` instructions in the same Dockerfile. Intermediate stages carry compilers, development headers and build tooling; the final stage copies only the finished artifact. Nothing left behind in earlier stages reaches the published image, not even as a deleted layer.

That nuance matters for security: in a single-stage image, a `RUN rm secret.pem` does not remove the file, it merely hides it behind a new layer. Anyone holding the image can extract it from the previous one. In a multi-stage build, a secret that lived in the build stage simply does not exist in the result.

Python with uv, cache mounts and a non-root user

```
# syntax=docker/dockerfile:1.7
FROM python:3.13-slim AS builder

# uv resolves and installs orders of magnitude faster than pip
COPY --from=ghcr.io/astral-sh/uv:0.9.7 /uv /usr/local/bin/uv

WORKDIR /app
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy

# 1) Dependencies only: this layer survives code changes
COPY pyproject.toml uv.lock ./
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-install-project --no-dev

# 2) Now the project itself
COPY src/ ./src/
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev

# ---------- runtime ----------
FROM python:3.13-slim AS runtime

RUN useradd --create-home --uid 10001 app
WORKDIR /app

COPY --from=builder --chown=10001:10001 /app/.venv /app/.venv
COPY --from=builder --chown=10001:10001 /app/src /app/src
```

```
ENV PATH="/app/.venv/bin:$PATH" \
    PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1

USER 10001
EXPOSE 8000
CMD ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Three details do the work here. `--mount=type=cache` mounts a directory that persists between builds and **does not end up inside the image**: the downloaded package index is reused even when the layer is invalidated, and that is the difference between ninety seconds and three. Calling `uv sync` twice (first without the project, then with it) keeps dependencies in a different layer from your code. And the virtualenv copied into the final image carries only what the runtime needs, without the compiler or the uv cache.

Node with pnpm and dependency pruning

```
# syntax=docker/dockerfile:1.7
FROM node:22-slim AS deps
WORKDIR /app
RUN corepack enable
COPY package.json pnpm-lock.yaml ./
RUN --mount=type=cache,target=/pnpm/store \
    pnpm config set store-dir /pnpm/store && pnpm install --frozen-lockfile

FROM node:22-slim AS build
WORKDIR /app
RUN corepack enable
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN pnpm run build
# Keep production dependencies only, ready to copy into the runtime
RUN --mount=type=cache,target=/pnpm/store \
    pnpm prune --prod

FROM node:22-slim AS runtime
ENV NODE_ENV=production
WORKDIR /app
COPY --from=build --chown=node:node /app/node_modules ./node_modules
COPY --from=build --chown=node:node /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

The `pnpm prune --prod` step is the one almost everyone forgets. Without it you ship TypeScript, ESLint, Vitest and their dependency trees to production: hundreds of megabytes and hundreds of packages that never execute but do show up in your vulnerability report.

Build-time secrets without leaking them

Passing a private token through `ARG` or `ENV` writes it into the image metadata forever. `docker history` will show it. BuildKit solves this with secret mounts: the file exists for the duration of that `RUN` and disappears afterwards, producing no layer.

```
# In the Dockerfile
RUN --mount=type=secret,id=npm_token,env=NPM_TOKEN \
    npm config set //registry.npmjs.org/:_authToken=${NPM_TOKEN} && \
    npm ci --omit=dev

# On the command line
docker build --secret id=npm_token,env=NPM_TOKEN -t api:dev .
```

For private dependencies over SSH there is an equivalent `--mount=type=ssh`, which forwards your agent into the build without copying the key. If you ever see an `ARG GITHUB_TOKEN` in a Dockerfile, that token is compromised the moment the image leaves the registry.

The cache that evaporates in CI

CI runners are ephemeral: every job starts with an empty build store, so the local layer cache does not exist. The fix is to export and import the cache against a registry.

```
- name: Build and push
  uses: docker/build-push-action@v6
  with:
    context: .
    push: true
    tags: ghcr.io/acme/api:${{ github.sha }}
    cache-from: type=registry,ref=ghcr.io/acme/api:buildcache
    cache-to: type=registry,ref=ghcr.io/acme/api:buildcache,mode=max
    provenance: mode=max
    sbom: true
```

The detail that ruins half of these setups is `mode=max`. By default the exporter uses `mode=min`, which stores only the layers present in the final image. In a multi-stage build that means the compilation stages — precisely the expensive ones — **are not cached**, and every run recompiles everything while the log cheerfully reports that caching is enabled. With `mode=max` the intermediate stages are exported too.

A cost warning: `mode=max` stores considerably more, so put a retention policy on the cache tag. And if your provider offers one, a persistent remote builder (Docker Build Cloud, a self-hosted builder with a volume, or buildx on Kubernetes) removes the problem at the root, because the BuildKit store stops being ephemeral.

COPY --link so invalidations do not cascade

By default, a `COPY --from=builder` depends on the filesystem state of the source stage: if that stage changes, the copy is redone. With `COPY --link` the layer is created independently and linked into the result, so it stays valid even when the previous stage or the base image changes. It is particularly useful when you bump the runtime base and do not want to rebuild everything you copied on top of it.

```
COPY --link --from=builder /app/.venv /app/.venv
```

The trade-off: `--link` cannot see the destination filesystem, so you cannot use it to write over an existing directory whose contents you expect to keep, and it combines poorly with `--chown` against users created in the same stage. Use it to copy whole trees into fresh paths.

Choosing a base image: four families, four trade-offs

This is where 90% of the size and the attack surface is decided. The reasonable options in 2026:

- **Debian slim** (`python:3.13-slim`, `node:22-slim`). Glibc, a package manager, a shell. Maximum compatibility and comfortable debugging. Between 70 and 150 MB of base and a handful of inherited CVEs that are rarely exploitable but that your scanner will flag anyway. It is the sensible starting point.
- **Alpine**. 5-10 MB, but it uses musl instead of glibc. For statically compiled Go that is perfect. For Python it is a classic trap: many packages publish wheels only for manylinux (glibc), so pip compiles from source and your build goes from two minutes to twenty. There are also real performance differences in musl's memory allocator under heavily threaded workloads.
- **Distroless** (`gcr.io/distroless/*`, from Google, Debian-based). Only the language runtime and your application: no shell, no package manager, no `curl`. It drastically reduces what an attacker can do after an RCE. They ship a `:debug` tag with BusyBox for when you need to get in.
- **Wolfi / Chainguard and other hardened images**. Distroless philosophy but with glibc, rebuilt daily to keep zero known CVEs, signed with Sigstore and shipping SBOM and SLSA provenance attestations. The broad catalog is commercial; there are alternatives in the same space (Docker Hardened Images, UBI Micro, Chiselled Ubuntu). If you have to answer a security questionnaire, this saves a lot of time.

One practical piece of advice: do not jump straight from `:latest` to distroless. Move to `-slim` with multi-stage first, measure, and only then evaluate whether a shell-less runtime is worth it operationally.

Go: the case where the result is 8 MB

```
# syntax=docker/dockerfile:1.7
FROM golang:1.25 AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    CGO_ENABLED=0 go build -trimpath -ldflags="-s -w" -o /out/api ./cmd/api

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/api /api
USER nonroot:nonroot
EXPOSE 8080
ENTRYPOINT ["/api"]
```

`-trimpath` strips absolute paths from the binary (required for reproducible builds), `-ldflags="-s -w"` removes the symbol table and debug information, and `CGO_ENABLED=0` produces a static binary that does not even need libc. The mounted `go-build` cache is what stops the whole tree from recompiling on every change.

Making the container a good citizen

Size aside, four details separate an image that works from one that operates well:

Non-root user with a numeric UID. Kubernetes cannot evaluate `runAsNonRoot` when `USER` is a name,

because the name only resolves inside the container. Use `USER 10001`, not `USER app`.

Exec-form ENTRYPOINT. `CMD python app.py` (shell form) starts a `/bin/sh -c` as PID 1 that does not forward SIGTERM to your process. The result is that Kubernetes waits out the full grace period and then SIGKILLs: dropped requests on every deploy. Always use the list form: `CMD ["python", "app.py"]`.

Read-only root filesystem. If the application writes nothing, declare `readOnlyRootFilesystem: true` and mount an `emptyDir` at `/tmp`. Building the image with that in mind from the start avoids discovering at three in the morning that something was writing into `/app`.

OCI metadata. Label the origin; you will thank yourself when you find an orphaned image in the registry.

```
LABEL org.opencontainers.image.source="https://github.com/acme/api" \
      org.opencontainers.image.revision="${GIT_SHA}" \
      org.opencontainers.image.licenses="Apache-2.0"
```

Provenance, SBOM and digest pinning

An image with no evidence of how it was built is an anonymous binary running with access to your internal network. BuildKit generates provenance and SBOM attestations with no extra work:

```
docker buildx build \
  --provenance=mode=max \
  --sbom=true \
  -t ghcr.io/acme/api:1.4.2 --push .

# Inspect it afterwards
docker buildx imagetools inspect ghcr.io/acme/api:1.4.2 \
  --format '{{ json .Provenance }}'
```

And the other side of the coin: **pin your base images by digest**. A tag is a mutable pointer; `python:3.13-slim` points at something different every week, so two builds of the same commit can produce different images.

```
FROM python:3.13-slim@sha256:1e5f0ba1f52d... AS runtime
```

Yes, that means updating the digest periodically; it is exactly the work Renovate or Dependabot do for you with a `docker` rule. What you gain is a deterministic build and the guarantee that a compromised base does not slip silently into your next deploy.

For full reproducibility there is one more piece: timestamps. BuildKit honours `SOURCE_DATE_EPOCH` and rewrites layer timestamps, so two builds of the same source tree produce the same digest.

```
SOURCE_DATE_EPOCH=$(git log -1 --pretty=%ct) \
docker buildx build --output type=image,rewrite-timestamp=true -t api:repro .
```

Multi-platform builds: why your ARM build takes seven times longer

The day someone on the team buys an ARM laptop, or the day production moves to Graviton instances, the Dockerfile that worked stops being enough. You need an image that serves both `linux/amd64` and

`linux/arm64`, and how you build it decides whether the pipeline goes from three minutes to five or from three minutes to twenty.

There are three paths, and it pays to understand the trade-off in each before choosing.

QEMU emulation. This is the one-line option: you register the binary interpreter and Docker runs the other architecture's instructions by translating them on the fly. It works for almost everything and requires no Dockerfile changes. The problem is the price. Instruction-by-instruction translation is expensive, and for CPU-intensive work (compiling C, building native Python extensions, transpiling TypeScript) the measurements Docker publishes point to builds roughly seven times slower on the emulated architecture. If your emulated stage crosses five minutes, you have left the range where QEMU is a good idea.

```
docker run --privileged --rm tonistiigi/binfmt --install all
docker buildx create --name multi --driver docker-container --use
docker buildx build --platform linux/amd64,linux/arm64 \
  -t registry.example.com/app:1.4.0 --push .
```

Cross-compilation. Instead of emulating the whole build, you run the build stage natively and tell the compiler to produce binaries for the other architecture. BuildKit exposes two sets of automatic variables that make this workable: `BUILDPLATFORM`, `BUILDOS` and `BUILDARCH` describe the machine doing the building; `TARGETPLATFORM`, `TARGETOS` and `TARGETARCH` describe the image being produced. The key is pinning the build stage to `--platform=$BUILDPLATFORM` so nobody emulates it.

```
FROM --platform=$BUILDPLATFORM golang:1.25-alpine AS build
ARG TARGETOS
ARG TARGETARCH
WORKDIR /src
COPY go.mod go.sum ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    CGO_ENABLED=0 GOOS=$TARGETOS GOARCH=$TARGETARCH \
    go build -trimpath -ldflags="-s -w" -o /out/api ./cmd/api

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]
```

With this pattern, adding a second architecture does not double the time: the shared stages (module download, source copy) resolve once and only the final link repeats. Docker's published figures for this case describe an increase of around 70% versus the 600% of pure emulation.

Native runners. The third path is neither emulating nor cross-compiling: have one builder per architecture and let each build its own on real hardware. Buildx supports node clusters, and GitHub Actions now offers managed ARM runners. It is the fastest option and the one that surprises you least with native dependencies, at the cost of extra infrastructure.

```
docker buildx create --name cluster --node amd --platform linux/amd64 \
  --driver remote tcp://builder-amd64.internal:1234
docker buildx create --append --name cluster --node arm --platform linux/arm64 \
  --driver remote tcp://builder-arm64.internal:1234
docker buildx build --builder cluster --platform linux/amd64,linux/arm64 \
  -t app:1.4.0 --push .
```

A practical rule: start with QEMU because it costs nothing, measure, and move to cross-compilation when the emulated stage passes five minutes. Save native runners for when cross-compilation is not viable, which in practice means dependencies with C extensions that do not cross cleanly: some scientific Python wheels, Node modules that go through `node-gyp`, and system libraries with no cross package available.

The silent multi-platform mistake

When you build for two platforms, the result is not an image: it is an index (formerly called a manifest list) pointing at two images. That has two consequences that surprise a lot of people. First, `--load` does not work with multiple platforms, because the daemon's classic image store cannot hold an index; either you use `--push`, or you enable the `containerd`-backed store. Second, if you tag one platform's image and push it separately, you clobber the index and orphan the other architecture. Always push the complete index in a single operation and check the result:

```
docker buildx imagetools inspect registry.example.com/app:1.4.0
```

Bake: describing the build as configuration, not a 40-line script

There comes a point when the build command no longer fits on one line. You have three images (API, worker, migrations), two platforms, tags that depend on the branch, arguments that differ between staging and production, and a block of `--cache-from` and `--cache-to` nobody dares touch. That command ends up copied into the Makefile, the CI workflow and the README, and the three copies drift.

`docker buildx bake` solves this by moving the build definition into a versioned file. It supports inheritance between targets, matrices, variables and groups, and the same file runs identically on your laptop and in CI.

```
variable "REGISTRY" { default = "registry.example.com" }
variable "TAG"      { default = "dev" }

group "default" {
  targets = ["api", "worker"]
}

target "_common" {
  context    = "."
  platforms  = ["linux/amd64", "linux/arm64"]
  args = {
    PYTHON_VERSION = "3.13"
  }
  cache-from = ["type=registry,ref=${REGISTRY}/cache:buildcache"]
  cache-to   = ["type=registry,ref=${REGISTRY}/cache:buildcache,mode=max"]
  attest = [
    "type=provenance,mode=max",
    "type=sbom"
  ]
}

target "api" {
  inherits = ["_common"]
  target   = "runtime"
  tags     = ["${REGISTRY}/api:${TAG}"]
}
```

```
target "worker" {
  inherits = ["_common"]
  target   = "worker-runtime"
  tags     = ["${REGISTRY}/worker:${TAG}"]
}
```

With that, building everything is `docker buildx bake --push`, and building just the API for a local test is `docker buildx bake api --set api.platforms=linux/arm64 --load`. The `--set` flag overrides any field from the command line without editing the file, which is exactly what you want for a one-off experiment.

Two details that save grief. First: `bake` reads `docker-bake.hcl`, `docker-bake.json` and also `compose.yaml`, so if you already have Compose you can start without writing anything new. Second: targets starting with an underscore are not invalid by convention, they simply are not part of the default group unless you name them explicitly, which makes them the natural home for shared configuration.

Testing inside the build instead of after it

A test stage that produces no artifact is a cheap way to guarantee nobody publishes an image with a red suite. BuildKit does not run stages nobody needs, so the test stage only runs when you ask for it.

```
FROM build AS test
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --group dev
RUN python -m pytest -q --maxfail=1

FROM scratch AS test-report
COPY --from=test /src/reports /
```

```
# In CI: fail the pipeline if tests fail, and extract the report
docker buildx bake test-report --set test-report.output=type=local,dest=./reports
```

The trick with the `scratch` stage and `output=type=local` is that BuildKit writes the stage's contents to your filesystem instead of into an image. It works for coverage reports, compiled binaries, or any artifact you want out of the build without publishing an image.

Measure before optimising: what image size actually means

Almost every Docker guide ends with a megabyte comparison, and almost all of them measure the wrong thing. The number `docker images` returns is the uncompressed size of all that image's layers, counted as if nothing were shared. That is not the number you care about.

Three different things matter:

- **Bytes transferred on a cold pull.** That is the *compressed* layer size, not the uncompressed one. A 300 MB layer of text may travel in 40 MB. It determines how long a new node takes to start serving.
- **Bytes transferred on a normal deploy.** That is only the size of the layers that changed. If your dependencies live in a stable layer and your code in another, a deploy moves kilobytes even if the image weighs 800 MB. This is the metric that actually affects deploy speed, and almost nobody measures it.
- **Attack surface.** Not measured in megabytes but in installed packages. A 900 MB image with twenty packages is safer than a 120 MB one with two hundred.

To see an image's real anatomy, `docker history` gives you the per-layer breakdown with the command that created it:

```
docker history --no-trunc --format 'table {{.Size}}\t{{.CreatedBy}}' app:1.4.0 | head -20
```

And for the actual compressed size, the one that travels over the network, you have to ask the registry:

```
docker buildx imagetools inspect --raw registry.example.com/app:1.4.0 \
  | jq '[.layers[].size] | add / 1024 / 1024 | round'
```

The gap between the two is usually a factor of two to three. If you are justifying an optimisation to your team, use the compressed number: that is the one that translates into startup seconds.

Layers are shared, and that changes the arithmetic

If your fifteen services use the same base image, that base is downloaded and stored once per node. Shrinking the base from 120 MB to 20 MB saves 100 MB once, not fifteen times. Meanwhile, a 300 MB application dependency layer that differs per service is paid fifteen times.

The practical consequence runs against intuition: standardising the base image across the organisation usually saves more disk and more pull time than chasing the smallest possible base in each repository separately. And it has a bonus: when a CVE lands in the base, there is exactly one place to update.

A size budget in CI

Images grow through an accumulation of small decisions, not one big one. A budget that fails the build when the threshold is crossed turns that growth into an explicit conversation.

```
#!/usr/bin/env bash
set -euo pipefail
IMAGE="${1:?usage: check-size.sh image:tag}"
BUDGET_MB="${2:-250}"

SIZE_MB=$(docker image inspect "$IMAGE" --format '{{.Size}}' | awk '{print int($1/1048576)}')
echo "Uncompressed size: ${SIZE_MB} MB (budget: ${BUDGET_MB} MB)"

if [ "$SIZE_MB" -gt "$BUDGET_MB" ]; then
  echo "Image exceeds budget. Largest layers:"
  docker history --format 'table {{.Size}}\t{{.CreatedBy}}' "$IMAGE" | head -8
  exit 1
fi
```

Put it on the main branch pipeline, not on every pull request: you want a signal when something lands, not a constant block while someone experiments.

Scanning for vulnerabilities without drowning in noise

The first scan of a Debian-based image returns between two and three hundred findings. The usual reaction is one of two: ignore them all, or open a ticket for each and burn two weeks. Neither works. Scanning only pays off when it produces a short list of actionable items, and getting there takes three decisions.

First: separate the base from your code. OS CVEs are fixed by updating the base image, and that is scheduled maintenance work. Application dependency CVEs are fixed by updating the lockfile, and that is

the job of the team writing the code. Mixing both in the same report guarantees nobody looks at either.

```
# Only what is on you: application dependencies, not the OS
trivy image --scanners vuln --vuln-type library \
  --severity HIGH,CRITICAL --ignore-unfixed \
  --exit-code 1 registry.example.com/app:1.4.0

# Separate report for the base, for the maintenance cycle
trivy image --scanners vuln --vuln-type os \
  --severity HIGH,CRITICAL --format table \
  registry.example.com/app:1.4.0
```

Second: `--ignore-unfixed` **by default.** A CVE with no available patch is not actionable: you can do nothing except wait or change base. Removing it from the report that blocks the build is not hiding the problem, it is separating what can be fixed today from what cannot. Keep a separate report, without that filter, that someone reviews weekly.

Third: VEX for structural false positives. Many findings are true and simultaneously irrelevant: the vulnerable library is in the image but your code never reaches the affected path. VEX (Vulnerability Exploitability eXchange) is the standard format for declaring that auditably, rather than keeping an exclusion list in a config file nobody can justify six months later.

```
{
  "@context": "https://openvex.dev/ns/v0.2.0",
  "author": "security@example.com",
  "timestamp": "2026-09-02T09:00:00Z",
  "statements": [
    {
      "vulnerability": { "name": "CVE-2026-11841" },
      "products": [
        { "@id": "pkg:oci/app@sha256:9f2c1e...", "identifiers": { "purl": "pkg:oci/app" } }
      ],
      "status": "not_affected",
      "justification": "vulnerable_code_not_in_execute_path",
      "impact_statement": "The affected XML parser is only used by the import script, which
is not part of the runtime image."
    }
  ]
}
```

```
trivy image --vex ./vex/app.openvex.json --severity HIGH,CRITICAL \
  --exit-code 1 registry.example.com/app:1.4.0
```

The difference between an exclusion and a VEX statement is that the second carries an author, a date and a justification from a closed vocabulary. That makes it reviewable, and makes it possible to revoke when the analysis stops being true.

Where to put the gate

Scanning on the pull request and scanning the published image solve different problems. On the pull request you want fast feedback on what the person just changed; there the scan should fail only on application dependencies with an available patch. On the published image you want the full inventory, including what has no fix, because you need to know what you are exposed to even if you cannot act today.

And there is a third moment almost nobody covers: periodic rescanning of already-deployed images. An

image that was clean on Tuesday can have three criticals by Friday because a new CVE was published, and no build will tell you because nobody rebuilt. A daily job that scans the tags running in production and alerts is probably the best effort-to-risk control in this entire section.

```
name: rescan-production
on:
  schedule: [{ cron: "0 6 * * *" }]
  workflow_dispatch:
jobs:
  scan:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        image: [api, worker, scheduler]
    steps:
      - uses: aquasecurity/trivy-action@master
        with:
          image-ref: registry.example.com/${{ matrix.image }}:production
          severity: HIGH,CRITICAL
          ignore-unfixed: true
          exit-code: "1"
          format: sarif
          output: trivy-${{ matrix.image }}.sarif
      - uses: github/codeql-action/upload-sarif@v3
        if: always()
        with:
          sarif_file: trivy-${{ matrix.image }}.sarif
```

Signing the image and verifying the signature where it counts

Generating an SBOM and a provenance attestation answers *what is inside and where it came from*. The other half is missing: *who says so, and how do I know it was not swapped*. That is what the signature is for.

The modern way to sign containers does not use long-lived keys. With Sigstore's *keyless* flow, the CI process identifies itself to Fulcio with an OIDC token, receives a ten-minute ephemeral certificate, signs with it, and the signature is recorded in Rekor, a public append-only transparency log. There is no private key to rotate or leak, and verification does not check a key but an identity: *this image was signed by the release.yml workflow of the example/app repository*.

```
      - uses: sigstore/cosign-installer@v3
      - name: Sign the image
        env:
          COSIGN_EXPERIMENTAL: "1"
        run: |
          cosign sign --yes \
            registry.example.com/app@${{ steps.build.outputs.digest }}
```

Notice that it signs the *digest*, not the tag. Signing `app:1.4.0` means nothing: the tag is a mutable pointer and tomorrow it can point elsewhere. The digest is the content.

Verification looks like this:

```
cosign verify \
  --certificate-identity-regexp
'^https://github.com/example/app/.github/workflows/release.yml@refs/heads/main$' \
```

```
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
  registry.example.com/app@sha256:9f2c1e...
```

That command is what separates a decorative signature from a real control. If you only check that a signature exists, anyone with an account can sign anything and pass your verification. What makes the control effective is pinning the specific identity: that repository, that workflow, that branch.

Verify at admission, not in the pipeline

A `cosign verify` inside the same pipeline that just signed the image proves very little. The place where verification has value is the point where the cluster decides whether to start a pod, because that also catches images that arrived through paths nobody planned: a manual deploy, a third-party chart, a tag moved by hand.

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-signature-and-identity
spec:
  validationFailureAction: Enforce
  background: false
  rules:
    - name: verify-signature
      match:
        any:
          - resources:
              kinds: [Pod]
              namespaces: ["production", "staging"]
      verifyImages:
        - imageReferences:
            - "registry.example.com/*"
          mutateDigest: true
          required: true
          attestors:
            - entries:
                - keyless:
                    subject:
"https://github.com/example/*/.github/workflows/release.yml@refs/heads/main"
                    issuer: "https://token.actions.githubusercontent.com"
                    rekor:
                      url: https://rekor.sigstore.dev
```

The `mutateDigest: true` field deserves a comment: it rewrites the pod's image reference to replace the tag with the digest that was just verified. Without it, you verify one digest and the kubelet could pull a different one if the tag changes between admission and start. It is a small window, but it is exactly the gap an attacker with registry access would need.

Roll the policy out in `Audit` mode first for a couple of weeks. You will discover images in production nobody remembered, and that inventory alone justifies the exercise even if you never flip to `Enforce`.

Debugging an image with no shell

The most common objection to distroless is real: when something breaks at three in the morning, you cannot get in. No `sh`, no `ls`, no `curl`. It is worth answering properly, because the answer makes the objection stop applying.

The technique is the same in Docker and in Kubernetes: do not put tools in the production image, attach an ephemeral container that brings them and shares the broken container's namespaces.

```
# Docker: a tools container sharing network and PID with the broken one
docker run --rm -it \
  --pid=container:api \
  --network=container:api \
  --cap-add SYS_PTRACE \
  nicolaka/netshoot

# Inside: the broken container's process is visible and its network is yours
ps aux
ss -tlnp
curl -sv localhost:8080/healthz
cat /proc/1/environ | tr '\0' '\n'
```

```
# Kubernetes: ephemeral container in the already-running pod
kubectl debug -it pod/api-7d9f8b6c5-xk2p9 \
  --image=nicolaka/netshoot \
  --target=api \
  --profile=general

# See the target container's filesystem from the ephemeral one
ls /proc/1/root/app
```

That `/proc/1/root` is the part people do not know and the one that makes the technique complete: from the ephemeral container you can read the shell-less container's entire filesystem, including config files, on-disk logs and binaries. The production image stays minimal and you keep every tool.

For the specific case where the container will not even start, there are two more tricks. One: `kubectl debug pod/api --copy-to=api-debug --set-image=api=busybox --share-processes -it` creates a copy of the pod with a different image and the same configuration, network and volumes, without touching the original. Two: keep a `debug` stage in the Dockerfile that starts from the production one and adds a shell, published under the same tag plus a `-debug` suffix. It costs a minute of build time and removes the temptation to put a shell in the real image.

```
FROM gcr.io/distroless/static-debian12:nonroot AS runtime
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]

FROM gcr.io/distroless/static-debian12:debug-nonroot AS debug
COPY --from=build /out/api /api
USER 65532:65532
ENTRYPOINT ["/api"]
```

CI caching in depth: the three backends and when to use each

The earlier section on the cache that evaporates explained the principle. It is worth going into detail, because the backend choice changes the outcome more than any other pipeline setting.

Inline cache. `--cache-to type=inline` embeds the cache metadata inside the published image itself. It needs no infrastructure and costs nothing, but it only stores the final stage's layers. In a multi-stage build, which is the normal case, that means the expensive stages (compiling, installing dependencies) are not cached at all. It works for single-stage builds and little else.

Registry cache. `--cache-to type=registry,ref=...,mode=max` publishes the cache as a separate artifact in your registry. It stores every intermediate stage, works across ephemeral runners and across repositories, and is the right option for most teams. The `mode=max` is not optional: without it the default is `mode=min`, which stores only the final stage and leaves you exactly where you were.

```
docker buildx build \
  --cache-from type=registry,ref=registry.example.com/cache:main \
  --cache-from type=registry,ref=registry.example.com/cache:pr-${PR_NUMBER} \
  --cache-to   type=registry,ref=registry.example.com/cache:pr-${PR_NUMBER},mode=max \
  -t registry.example.com/app:${SHA} --push .
```

That two-`--cache-from` pattern is what makes pull requests fast: the branch reads its own cache first and, if it does not exist because this is the first build, falls back to the main branch's. And it writes to its own key so it does not pollute the shared cache with intermediate states.

Object storage cache. `type=s3`, `type=azblob` and `type=gha` store the cache outside the registry. The GitHub Actions variant is convenient because it needs no extra credentials, but it is subject to the repository cache quota and its eviction policy: when it fills up, the least recently used entries are dropped, and that can mean your cache vanishes on a Monday with no explanation. For large pipelines, S3 with your own lifecycle policy gives much more predictable behaviour.

```
      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: registry.example.com/app:${{ github.sha }}
          cache-from: type=gha,scope=${{ github.ref_name }}
          cache-to: type=gha,mode=max,scope=${{ github.ref_name }}
          provenance: mode=max
          sbom: true
```

Why the cache misses even though the code did not change

When a build that should be instant takes ten minutes, the cause is almost always one of these four:

1. **A timestamp in the context.** BuildKit computes a `COPY` cache key from file contents and metadata. A file regenerated on every CI run (a `build-info.json` with the date, a full `.git`) invalidates everything after it. The `.dockerignore` fixes this.
2. **An `ARG` that always changes.** If you declare `ARG GIT_SHA` at the top of the Dockerfile, every later instruction depends on a different value on every commit. Declare volatile arguments as low as possible, ideally right before the instruction that uses them.
3. **The builder does not persist.** A fresh `docker buildx create` in every job creates an empty builder. The local cache lives in the builder; throw it away and you throw the cache away.
4. **Different platforms.** The cache is per platform. A `linux/arm64` build reuses nothing from a `linux/amd64` one, and if you build both in separate jobs with the same cache key, they overwrite each other.

When it is not clear which of the four it is, the right tool is BuildKit's plain progress output, which shows the execution graph and marks which steps came from cache:

```
BUILDKIT_PROGRESS=plain docker buildx build --no-cache-filter test . 2>&1 | grep -E
'CACHED|DONE'
```

The `--no-cache-filter` flag deserves its own mention: it invalidates a specific stage's cache and respects the rest. That is what you want when you need to re-run the tests without rebuilding dependencies.

What the cache costs and when to clean it

A registry cache with `mode=max` grows. In a monorepo with six images and active branches, tens of gigabytes in the cache repository after a few months is normal. An explicit policy is worth the effort: seven-day expiry on branch keys, indefinite on the main branch, and a size cap in the backend itself.

```
# Clean the builder's local cache, keeping what was used in the last week
docker buildx prune --filter 'until=168h' --keep-storage 20GB -f

# Persistent builder configuration
cat <<'TOML' > buildkitd.toml
[worker.oci]
  gc = true
  [[worker.oci.gcpolicy]]
    keepBytes = "20GB"
    keepDuration = "168h"
TOML
docker buildx create --name ci --driver docker-container --config buildkitd.toml --use
```

Startup, signals and health: the contract between the image and the orchestrator

A well-built image that misbehaves in Kubernetes usually fails in the same place: the start and stop contract. Four small details with large consequences.

Exec form is not a style preference. `CMD python app.py` (shell form) starts `/bin/sh -c "python app.py"`. PID 1 is the shell, and the shell does not forward SIGTERM to its child. When Kubernetes asks for a clean stop, your process never hears about it, exhausts the grace period, and the kubelet kills it with SIGKILL. In-flight requests are lost. `CMD ["python", "app.py"]` (exec form) makes your process PID 1 and delivers the signal directly.

PID 1 has responsibilities. Process 1 in a namespace adopts orphans and must reap them. If your application spawns subprocesses and never waits on them, you accumulate zombies. On top of that, default signal actions do not apply to PID 1, so a process that does not explicitly handle SIGTERM will ignore it. When your runtime is not prepared to be PID 1, a minimal init solves it.

```
FROM python:3.13-slim
RUN apt-get update && apt-get install -y --no-install-recommends tini \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["python", "-m", "app"]
```

With distroless, the alternative that installs nothing is a Go binary that already manages its goroutines, or `dumb-init` copied from a build stage.

STOPSIGNAL when your process does not speak SIGTERM. Some servers use a different convention: nginx does a fast shutdown on SIGTERM and a graceful one on SIGQUIT. If you do not declare it, the orchestrator sends SIGTERM and you cut connections mid-flight.

```
STOPSIGNAL SIGQUIT
```

Dockerfile HEALTHCHECK versus Kubernetes probes. They are different mechanisms and Kubernetes ignores the first entirely. The `HEALTHCHECK` instruction is only interpreted by the Docker daemon and Compose. If you deploy to Kubernetes, define probes in the manifest; if you also run the image locally with Compose, `HEALTHCHECK` remains useful so that `depends_on: condition: service_healthy` works. Having both is not duplication: they cover different environments.

```
HEALTHCHECK --interval=10s --timeout=2s --start-period=20s --retries=3 \
  CMD ["/app", "healthcheck"]
```

Notice the healthcheck invokes a subcommand of the binary itself rather than `curl`. That avoids installing curl in the final image just to check health, which is one of the most common reasons a distroless image ends up with all of Debian inside it.

Runtime environment variables almost nobody sets

A container with a 512 MiB memory limit tells your runtime nothing unless you tell it. The JVM, Node and Go read machine memory, not the cgroup, and discover the limit when the kernel kills them.

```
# Go: the garbage collector respects this and collects before the OOM
ENV GOMEMLIMIT=450MiB
ENV GOGC=100

# Node: old heap below the container limit
ENV NODE_OPTIONS="--max-old-space-size=384"

# Python: unbuffered so logs appear immediately
ENV PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1
```

The arithmetic matters: leave 10% to 20% of headroom between the runtime limit and the container limit, because the heap is not the process's whole memory. There are thread stacks, network buffers, allocator memory and native libraries.

The registry is infrastructure too

Images accumulate. A pipeline that publishes a tag per commit generates several thousand images per service per year, each dragging its layers. If nobody defines a policy, the cost grows linearly until someone asks about the bill.

A reasonable policy has three rules: keep semantic version tags and anything deployed indefinitely, keep branch tags for thirty days, and delete pull request tags after seven. The important part is the second half: **deleting the tag does not free space**. The layers stay until the registry's garbage collector runs, and in many managed registries that happens in a window you do not control.

```
{
  "rules": [
    {
      "rulePriority": 1,
      "description": "PRs: seven days",
      "selection": {
        "tagStatus": "tagged",
        "tagPrefixList": ["pr-"],
```

```
        "countType": "sinceImagePushed",
        "countUnit": "days",
        "countNumber": 7
      },
      "action": { "type": "expire" }
    },
    {
      "rulePriority": 2,
      "description": "Untagged: one day",
      "selection": {
        "tagStatus": "untagged",
        "countType": "sinceImagePushed",
        "countUnit": "days",
        "countNumber": 1
      },
      "action": { "type": "expire" }
    }
  ]
}
```

There is a specific trap with expiry policies and multi-platform images: if the rule deletes by age an architecture image that is still part of a live index, the index breaks and pulls for that architecture fail with an unhelpful error. Policies operating on untagged images cause this most often, because the images making up an index carry no tag of their own. Check that your registry understands indexes before enabling an aggressive untagged rule.

Pull-through caching and rate limits

If your nodes pull public images straight from Docker Hub, you are one traffic spike away from rate-limit errors in the middle of a scale-up. A registry with pull-through caching solves both at once: it removes the dependency on a third party's availability and removes the limit.

```
# containerd: redirect docker.io to the internal mirror
version = 2
[plugins."io.containerd.grpc.v1.cri".registry]
  config_path = "/etc/containerd/certs.d"
```

```
# /etc/containerd/certs.d/docker.io/hosts.toml
server = "https://registry-1.docker.io"

[host."https://registry.example.com/v2/dockerhub"]
  capabilities = ["pull", "resolve"]
```

It is half an hour of work and it removes an entire class of incident whose root cause always takes a while to surface, because the error shows up as pods in `ImagePullBackOff` with no apparent relation to whatever was just deployed.

Case study: from 14 minutes and 1.4 GB to 95 seconds and 180 MB

The numbers in this section come from a real service: a Python API on FastAPI with SQLAlchemy, a couple of dependencies with compiled extensions, and an admin frontend built with Vite. The starting point was the usual Dockerfile.

```
FROM python:3.13
WORKDIR /app
COPY . .
RUN apt-get update && apt-get install -y build-essential libpq-dev nodejs npm
RUN pip install -r requirements.txt
RUN cd frontend && npm install && npm run build
EXPOSE 8000
CMD uvicorn app.main:app --host 0.0.0.0 --port 8000
```

Baseline: 14 minutes 20 seconds for a cold CI build, 11 minutes 40 seconds warm (the cache almost never hit), 1.41 GB uncompressed, 512 MB compressed, 287 CVEs of which 9 were critical, and 38 seconds to pull on a fresh node.

Step 1: `.dockerignore` and copy order. The `COPY . .` at the top meant any change to any file invalidated the dependency install. Adding a `.dockerignore` (excluding `.git`, `node_modules`, `.venv`, `__pycache__` and test files) and copying manifests first brought the warm build down to 3 minutes 10 seconds. Half an hour of work, 73% less time. This is always the first optimisation because it has the best effort-to-result ratio.

Step 2: multi-stage. Separating the frontend build, the Python dependency install and the runtime into three stages removed `build-essential`, Node, npm and the development headers from the final image. 1.41 GB became 340 MB. CVEs dropped from 287 to 61, nearly all from the compile toolchain disappearing.

Step 3: cache mounts and uv. Replacing `pip install` with `uv sync` plus a cache mount for uv's cache directory, and `npm install` with `pnpm install --frozen-lockfile` plus its own cache mount, took the cold build from 14 minutes to 4 minutes 5 seconds and the warm build to 95 seconds. The single biggest lever on cold builds, which is what you feel whenever the runner is new.

```
FROM node:22-alpine AS frontend
WORKDIR /fe
RUN corepack enable
COPY frontend/package.json frontend/pnpm-lock.yaml ./
RUN --mount=type=cache,target=/root/.local/share/pnpm/store \
    pnpm install --frozen-lockfile
COPY frontend/ ./
RUN pnpm run build

FROM python:3.13-slim AS deps
COPY --from=ghcr.io/astral-sh/uv:0.9 /uv /usr/local/bin/uv
WORKDIR /app
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy
RUN apt-get update && apt-get install -y --no-install-recommends \
      build-essential libpq-dev && rm -rf /var/lib/apt/lists/*
COPY pyproject.toml uv.lock ./
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev --no-install-project

FROM python:3.13-slim AS runtime
RUN apt-get update && apt-get install -y --no-install-recommends libpq5 \
    && rm -rf /var/lib/apt/lists/* \
    && useradd --uid 10001 --no-create-home --shell /usr/sbin/nologin app
WORKDIR /app
COPY --from=deps /app/.venv /app/.venv
COPY --from=frontend /fe/dist ./static
COPY --chown=10001:10001 app/ ./app/
ENV PATH="/app/.venv/bin:$PATH" PYTHONUNBUFFERED=1 PYTHONDONTWRITEBYTECODE=1
USER 10001:10001
```

```
EXPOSE 8000
ENTRYPOINT ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Step 4: a thinner runtime base. Moving from `python:3.13` (1.02 GB) to `python:3.13-slim` (154 MB) is what took the final image from 340 MB to 180 MB. `python:3.13-alpine` was tried and discarded: the dependencies with compiled extensions had no musl wheels and had to be built from source, which pushed the cold build back over seven minutes to save 30 MB. That is the classic Alpine-with-Python trade-off, and it rarely pays.

Step 5: registry cache in CI. CI runners are ephemeral, so the local cache did not survive between runs. With `--cache-to type=registry,mode=max`, a typical pull request build became a consistent 95 seconds instead of depending on luck.

Step 6: provenance, SBOM and signing. Adding `--provenance=mode=max --sbom=true` and cosign signing added 11 seconds to the pipeline. That is the real cost of the supply-chain half, and it is why there is no reasonable argument for leaving it until later.

Migration summary

- **Cold build**: 14 min 20 s !’ 4 min 05 s
- **Warm build**: 11 min 40 s !’ 95 s
- **Uncompressed size**: 1.41 GB !’ 180 MB
- **Compressed size**: 512 MB !’ 68 MB
- **Pull on a fresh node**: 38 s !’ 6 s
- **High and critical CVEs**: 9 critical and 41 high !’ 0 critical and 3 high
- **Layers changed per deploy**: all of them !’ one, the code layer, at 2.1 MB

The last line is the interesting one. Before, every deploy moved 512 MB per node because `COPY . .` invalidated everything. After, a normal deploy moves 2.1 MB. That appears in no image-size comparison and is by far the most noticeable change in day-to-day deploy speed.

Total cost of the work: about nine hours across three sessions. Step 1 was done in the first half hour and already justified the rest.

Building without a daemon: rootless, Kubernetes and remote builders

Everything so far assumes the build happens on your laptop. In production it almost never does: it happens on a CI runner that very often lives inside Kubernetes, and that changes the subject. Mounting `/var/run/docker.sock` into the pod that builds is the first thing anyone finds in a search engine, and it is also the fastest way to turn a pull request into root access on the node: whoever talks to that socket can start a privileged container with the host filesystem mounted and read every other pod's secrets. If your pipeline builds code nobody has reviewed yet — and that is exactly what a pull request pipeline does — the socket is not a defensible option.

For years the usual answer was Kaniko, which built images without a daemon from inside a container. Google archived the repository in June 2025: it still works, but it no longer receives bug or CVE fixes, and several projects have removed their documentation for it. Third-party forks exist, though building your supply chain on a fork of an abandoned project is a decision worth making on purpose rather than by inertia. The two actively maintained alternatives are BuildKit in rootless mode and Buildah.

BuildKit has one concrete advantage here: it is the same engine you already use locally, so the Dockerfile, the cache mounts, the build secrets and `bake` all behave identically. Buildx knows how to start the builder inside the cluster:

```
# A builder that runs in the cluster, unprivileged
docker buildx create \
  --name k8s-builder \
  --driver kubernetes \
  --driver-opt namespace=ci,replicas=2,rootless=true \
  --driver-opt requests.cpu=2,requests.memory=4Gi,limits.memory=8Gi \
  --use

# The build runs in the pods, not on the runner
docker buildx build \
  --cache-from type=registry,ref=registry.example.com/app:buildcache \
  --cache-to   type=registry,ref=registry.example.com/app:buildcache,mode=max \
  --push -t registry.example.com/app:$GIT_SHA .
```

The `rootless=true` option creates pods without `securityContext.privileged`, relying on user namespaces. Two details you tend to discover late: on some distributions you need to relax the pod's seccomp and AppArmor profiles for the userns to work, and unprivileged overlayfs requires a reasonably modern kernel; when it is unavailable BuildKit falls back to fuse-overlayfs, which works but is noticeably slower writing large layers. It is worth checking the builder logs to see which one is in use before blaming the Dockerfile for a slow build.

The serious problem with an ephemeral builder is the cache. A pod that is born and dies with the build always starts cold, which loses exactly what makes BuildKit fast. There are three ways out, and they are not mutually exclusive. The first is the registry cache with `mode=max` we already covered: it survives the pod because it does not live in the pod. The second is giving the builder a persistent volume and routing builds from the same repository to the same replica so the local cache hits; it works, but it introduces affinity and state you have to operate. The third is to stop having ephemeral builders at all: a long-lived remote builder (the `remote` driver pointing at your own buildkitd, or a managed service such as Docker Build Cloud) keeps the cache warm between builds and, if it also offers native ARM machines, eliminates the QEMU problem from the multi-platform section outright.

Buildah is the alternative when you already live in the Podman ecosystem: it builds daemonless by design, understands Dockerfiles, and additionally lets you write the build as a step-by-step shell script, which is convenient for images generated from logic rather than from a static file. What you do not get is exact parity with BuildKit: cache mounts and the registry cache with `mode=max` are BuildKit territory, and if your Dockerfile depends on them the migration is not just swapping a binary.

When not to write a Dockerfile: buildpacks, ko and jib

There is a question almost nobody asks out loud: do you actually need a Dockerfile? Maintaining a good one per service has a recurring cost — every runtime release, every CVE in the base, every time someone copies a three-year-old Dockerfile into a new service — and in organisations with many repositories that cost multiplies by the number of teams. The three tools that avoid the Dockerfile are not interchangeable; each solves a different case.

Cloud Native Buildpacks inverts the framing: instead of describing how the image is built, you describe nothing and the *builder* detects the language, installs the right runtime, compiles, and produces an image that already has a non-root user, OCI labels and an SBOM. It is convention over configuration, with the operational upside that somebody else maintains the convention:

```
# No Dockerfile: the builder decides how it is built
pack build registry.example.com/app:$GIT_SHA \
  --builder paketobuildpacks/builder-jammy-base \
  --env BP_JVM_VERSION=21 \
  --publish

# Update only the base OS of hundreds of images,
# without recompiling the app: the lower layers are rewritten
pack rebase registry.example.com/app:$GIT_SHA --publish
```

That `pack rebase` is the strong argument and the one almost nobody mentions. When a CVE lands in the base image, a fleet of Dockerfiles forces you to rebuild every application, with all the pipelines, tests and risk of something shifting that implies. With buildpacks, application layers and OS layers are separated by contract, so you can swap the OS underneath and publish a new image in seconds without touching the compiled code. In exchange: builds use more memory and are usually slower than a well-cached Dockerfile, the resulting images are larger than a hand-tuned distroless, and when you need something the convention does not cover — a specific native library, an unusual build step — you end up writing a buildpack, which is more work than writing the Dockerfile you were trying to avoid.

ko is the cleanest case of all, and it only works for Go. It compiles the binary on your machine or on the runner, places it on a minimal static base and pushes the image straight to the registry, with no Docker daemon and no Dockerfile:

```
export KO_DOCKER_REPO=registry.example.com/app
ko build ./cmd/api --bare --sbom=spdx --platform=linux/amd64,linux/arm64

# It returns the reference by digest, ready for the manifest
# registry.example.com/app@sha256:...
```

For a Go service, ko removes half of this article in one move: there is no layer cache to optimise because there are no dependency layers, there is no QEMU because the Go compiler cross-compiles natively, and the output is a digest reference you can inject straight into the manifest. The limit is obvious: if your binary needs cgo, extra certificates, configuration files or a timezone database, you are already fighting ko's flags and you were probably better off with a three-line multi-stage build.

jib does the equivalent for the JVM from Maven or Gradle: it splits dependencies, resources and classes into separate layers — which is exactly the right change-frequency order — uses a distroless base by default, and needs no daemon. In a Java monorepo with dozens of modules, the difference against one Dockerfile per module is hard to overstate.

The practical criterion, condensed: many services in few languages and a desire to centralise base-OS maintenance, buildpacks. Go, ko. JVM, jib. A service with unconventional requirements, or a need for fine control over what lands in the final image for size or audit reasons, and the multi-stage Dockerfile still wins — and now you know how to write it properly.

Private dependencies and builds with no route to the internet

Every tutorial Dockerfile installs public dependencies. Yours, almost certainly, installs at least one package from a private index, and that is where credentials enter the build. We already saw why `ARG` is the worst possible choice: it is written into `docker history` forever and travels with the image. The right way is a mounted secret that exists only while that `RUN` executes and leaves no layer behind:

```
# syntax=docker/dockerfile:1.7
FROM python:3.13-slim AS deps
WORKDIR /app
COPY pyproject.toml uv.lock ./
# The .netrc exists during this RUN and nowhere else
RUN --mount=type=secret,id=netrc,target=/root/.netrc,mode=0400 \
    --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev --no-install-project
```

```
docker buildx build --secret id=netrc,src=$HOME/.netrc -t app:dev .

# In CI, from an environment variable instead of a file
docker buildx build --secret id=netrc,env=NETRC_CONTENT -t app:ci .
```

The pattern is the same across ecosystems, only the target path changes: `/root/.npmrc` for npm and pnpm, `/root/.m2/settings.xml` for Maven, `/root/.config/pip/pip.conf` if you prefer configuring pip by file. One precaution that gets forgotten: if the package manager writes the credentialed URL into the lockfile or into a cache file you later copy into the final stage, the secret leaks through the back door. Grep the resulting image the first time you wire this up, not the tenth.

Go has its own trap. `GOPRIVATE` tells the toolchain that certain modules bypass the public proxy and the checksum database, which is necessary for an internal repository. The tempting shortcut is `GOPRIVATE=*`, and with that you have just disabled cryptographic verification for *all* your dependencies, public ones included. The correct move is to list the real prefixes:

```
# Good: only internal modules skip the proxy and the checksum db
ENV GOPRIVATE=github.com/acme/*,gitlab.acme.internal/*
ENV GOFLAGS=-mod=readonly

# Bad: disables sum.golang.org for the entire dependency tree
# ENV GOPRIVATE=*
```

Proving that a step needs no network

A build that downloads things in steps where it should not is a non-reproducible build waiting for the remote index to have a bad day. BuildKit lets you assert the opposite in a checkable way with `--network=none`, which runs that `RUN` with no network interface. If the step only worked because it was quietly fetching something, it fails the moment you declare it hermetic — which is exactly when you want to find out:

```
FROM deps AS build
COPY . .
# Compiling needs no internet. If it does, I want to know now.
RUN --network=none python -m compileall -q src/
```

The other half of the problem is infrastructure, not Dockerfile. A build that depends on PyPI or the public npm registry depends on a third party being available before you can ship an urgent fix. An internal mirror with pull-through caching — Artifactory, Nexus, devpi, Verdaccio, or your cloud provider's proxy — turns that external dependency into an internal one and, along the way, gives you a single place to enforce version cooling periods and block yanked packages. In genuinely air-gapped environments it is the only path: the internal index syncs from outside under an explicit policy, and builds never have a route to the internet.

Tags, promotion by digest and the base image lifecycle

A Docker tag is a mutable pointer. That sentence, which reads like an implementation detail, is the cause of an entire family of incidents: the pod that restarts and comes up running different code from its siblings, the rollback to `v1.4.2` that brings the wrong binary because somebody overwrote the tag, the image that worked in staging and not in production while being *the same tag*. A digest, by contrast, is the hash of the manifest: if the digest matches, the content matches, and there is no argument to be had.

From that comes a simple operational rule: **build once, tag for humans, deploy by digest**. Tags exist so a person can find the image (`v1.4.2`, `main-a3f9c1`, `staging`); the manifest the orchestrator applies carries the digest. And promotion between environments rebuilds nothing — it moves the reference.

```
# Promote without rebuilding: it is the exact same artifact
DIGEST=$(crane digest registry.example.com/app:main-$GIT_SHA)
crane tag registry.example.com/app@$DIGEST staging

# And later, after the tests, to production
crane tag registry.example.com/app@$DIGEST v1.4.2

# The manifest references the digest, not the tag
# image: registry.example.com/app@sha256:9f2e...
```

Rebuilding in order to promote is the anti-pattern that voids every test you just passed: the image going to production is not the one you validated, even if the commit is identical, because transitive dependencies and the base image may have changed between the two runs. If your registry supports immutable tags — ECR, Artifact Registry and Harbor all do — turn them on for version tags and leave only convenience pointers such as `staging` mutable. Overwriting `v1.4.2` should be a registry error, not a decision made by whoever runs `push`.

The base image ages even when your code does not

Pinning the base by digest, as we saw, makes the build reproducible. It also freezes that base's CVEs: three months from now you will still be shipping the same `sha256:` with the patches that existed the day you pinned it. Reproducibility and freshness pull in opposite directions, and the resolution is not to pick one but to automate digest advancement so it becomes a reviewable change rather than an invisible side effect.

```
{
  "extends": ["config:recommended", "docker:pinDigests"],
  "packageRules": [
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["digest", "pin"],
      "automerge": true,
      "automergeType": "branch"
    },
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["minor", "patch"],
      "minimumReleaseAge": "3 days"
    },
    {
      "matchDatasources": ["docker"],
      "matchUpdateTypes": ["major"],
```

```
      "dependencyDashboardApproval": true
    }
  ]
}
```

What this configuration does: the `docker:pinDigests` preset turns `FROM python:3.13-slim` into `FROM python:3.13-slim@sha256:...` and keeps the readable tag in a comment beside it. Digest updates — same `3.13-slim`, new content — are automerged, because those are precisely the base's security patches and gating them behind human review means never applying them. Minor bumps wait out a three-day cooling period, and majors require explicit approval, because `3.13` to `3.14` is not a patch.

One piece is missing that neither Renovate nor Dependabot covers: an image that is already published and deployed accumulates vulnerabilities without anyone touching the repository. The answer is a periodic rebuild of the same commit against an updated base — a nightly or weekly cron that rebuilds the active tags and rescans them — combined with the daily rescan of what is deployed that we mentioned earlier. If the rebuild is fast and promotion is by digest, this costs very little; if the build takes fourteen minutes, you will not do it. Which is another way of saying that build speed is not a developer convenience but a security property of the system.

Cold start: why size is not the whole story

Cutting the image from 1.4 GB to 180 MB improves startup, but the relationship is not linear and it is worth understanding why before you keep shaving megabytes at diminishing returns. The classic study by Harter and colleagues measured that pulling the image accounts for roughly 76% of container startup time, while only a small fraction of the files in the image are actually read during that startup. In other words: you spend most of the time fetching data that will never be opened.

Before reaching for anything exotic, there are three levers that are almost always unused. The first is registry proximity: pulling from another region is a fixed cost per layer and per node, paid on every scale-up event, and a cluster-local pull-through cache removes it. The second is the kubelet's pull parallelism, which historically serialised pulls; since Kubernetes 1.27 you can enable parallel pulls and bound them so they do not saturate the node's network:

```
# KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
serializeImagePulls: false
maxParallelImagePulls: 5
imagePullProgressDeadline: 5m
```

The third is measuring instead of guessing. Pull time shows up in pod events and, in aggregate, in kubelet metrics:

```
# How long the pull actually takes on your nodes, p99
histogram_quantile(0.99,
  sum by (le, node) (
    rate(kubelet_runtime_operations_duration_seconds_bucket{
      operation_type="pull_image"
    }[30m])
  )
)

# And the detail for one specific pod
```

```
# kubectl describe pod api-7f9c | grep -A2 Pulled
```

When those three levers are already pulled and you still have a problem — typically with machine-learning images, which drag CUDA along and weigh several gigabytes for reasons no multi-stage build will fix — *lazy pulling* enters. The idea is to stop treating the image as an archive that must be downloaded whole before starting and start treating it as a remote filesystem whose blocks are fetched on demand. There are two mature implementations and the difference between them is practical, not philosophical.

eStargz, from containerd's stargz-snapshotter project, defines a lazily-pullable image format: the image must be converted, and the result stays compatible with standard OCI registries. **SOCI** (Seekable OCI), from AWS, avoids the conversion: it generates a separate index pushed alongside the image that allows individual files to be extracted from a tar layer without downloading it in full. If you control the build pipeline, either works; if you consume third-party images you cannot rebuild, SOCI is the one that fits. The trade-off is common to both: the I/O work does not disappear, it moves from startup into runtime, so an application that walks half the filesystem in its first minute can end up slower overall. In published measurements, SOCI tends to keep application startup closer to the original than eStargz does.

The managed numbers give a sense of the order of magnitude: GKE Image Streaming, which is this same technique offered as a service, took a 5.4 GB Triton image from 191 seconds to 30. That is the class of improvement you do not get by optimising the Dockerfile, because the Dockerfile was not the problem.

The boring option that works

If lazy pulling feels like a lot of machinery for the problem you actually have — and often it is — there is an alternative with no new components: pre-pull the image onto the nodes. A DaemonSet that pulls the image about to be deployed, or baking it into the AMI or node image, turns a cold start into a warm one. With `imagePullPolicy: IfNotPresent` and digest references there is no ambiguity about what runs:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: image-prepuller
spec:
  selector:
    matchLabels: { app: image-prepuller }
  template:
    metadata:
      labels: { app: image-prepuller }
    spec:
      initContainers:
        - name: prepull-api
          image: registry.example.com/app@sha256:9f2e...
          command: ["/bin/true"]
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.10
          resources:
            requests: { cpu: 10m, memory: 16Mi }
```

The cost is disk on the nodes and a cleanup policy you have to operate (the kubelet's image garbage collector evicts on a disk-usage threshold and may remove exactly the image you pre-pulled). But it does not put a new snapshotter in the critical startup path of every container you run, and for a service with a

handful of hot images it is usually the right answer.

Four more mistakes, the kind you only see in production

Beyond the eight above, four show up once the image has been running for a while.

Nine. Trusting `latest` anywhere. Not just in the `FROM`: in the Kubernetes manifest too. A moving tag in a `Deployment` means two replicas of the same deployment can be running different images if a pod restarts after someone retags. The symptom is non-reproducible behaviour that disappears on restart and returns three days later. Use digests, and if that complicates your flow, at least use immutable tags with the commit SHA.

Ten. Rebuilding the image per environment. If your pipeline builds one image for staging and another for production with different build arguments, what you deploy to production has never been tested. Configuration belongs in the environment, not in the image. The same digest should travel through every environment; only mounted variables and secrets change.

Eleven. Writing to the container filesystem. It works until you enable `readOnlyRootFilesystem`, or until you discover that the logs your application writes to `/app/logs` vanish on every restart and fill the node's disk in the meantime. Anything the application writes belongs in an explicit volume or on standard output. Declare `tmpfs` for scratch space:

```
        securityContext:
          readOnlyRootFilesystem: true
          allowPrivilegeEscalation: false
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop: ["ALL"]
        volumeMounts:
          - name: tmp
            mountPath: /tmp
      volumes:
        - name: tmp
          emptyDir:
            medium: Memory
            sizeLimit: 64Mi
```

Twelve. A named user instead of a numeric UID. `USER app` reads better than `USER 10001`, but Kubernetes cannot resolve usernames: it reads the image config's `User` field and, if it contains text rather than a number, the `runAsNonRoot` check fails and the pod does not start. It is a failure that only appears when someone hardens the cluster, months after the image was written.

Questions that come up while applying all this

Is distroless worth it if my team has little container experience? Probably not as a first step. The order that works is: `.dockerignore` and copy order, multi-stage, a *slim* base, and only then distroless. Each rung adds less benefit and more operational friction than the last. A team without multi-stage yet will get nothing from distroless and will definitely suffer the first incident where they cannot get into the container.

What if my application needs to run system commands? Then distroless is not for you, and that is fine. Distroless removes the shell and utilities on purpose; if your application calls `git`, `ffmpeg` or

`pg_dump`, you need a base with packages. The sensible middle ground is a Wolfi-style image, which does have a package manager and is also rebuilt continuously, so you install exactly what you need on top of a base with very few CVEs.

How many stages are too many? There is no meaningful technical limit, but there is a practical one: stages nobody uses are not built, so an extra stage costs nothing unless it is on the path to the final image. The problem is not the count but readability. If the Dockerfile grows past roughly 80 lines, that usually signals logic that belongs in a versioned script called from a single `RUN`.

Should I pin system package versions? It is an uncomfortable trade-off. Pinning `apt-get install nginx=1.26.2-1` makes the build reproducible but also makes it fail the day Debian retires that version from the repository, which for security updates is a matter of weeks. The reasonable stance for most teams is to pin the base image by digest (which already freezes the package set) and not pin individual packages, rebuilding on a known cadence to pick up patches.

How do I keep digests current if I pin everything? With the same tooling you already use for code dependencies. Renovate and Dependabot understand `FROM` lines with digests and open a pull request when the tag points at a new one. The key is the readable tag comment at the end of the line, which is what lets them know what they are tracking:

```
FROM python:3.13-slim@sha256:1e4b6f2a9c... # python:3.13-slim
```

Are reproducible builds actually achievable? Bit for bit, in the general case, not entirely: there are sources of non-determinism outside your control (filesystem ordering, randomness in some compilers, timestamps on downloaded packages). What is achievable and useful is practical reproducibility: pin the base by digest, pin dependencies by lockfile with hashes, normalise timestamps with `SOURCE_DATE_EPOCH`, and check that two consecutive builds of the same commit produce the same digest. When that stops holding, you have a reliable signal that something changed without anyone declaring it.

What about Podman, or daemonless builds? Everything in this guide about Dockerfile structure, layers, caching and base image choice applies unchanged. What differs is the tool running the build. Buildah and Kaniko build without a privileged daemon, which matters if you build inside the cluster itself. Kaniko has a different caching model (per layer, in a registry) and does not support BuildKit's cache mount syntax, so if you lean on those heavily the migration is not transparent.

Glossary

- **Layer**: the set of filesystem changes produced by one Dockerfile instruction. It is the unit of caching, transfer and shared storage.
- **Digest**: the SHA-256 hash of an image manifest. It identifies exact, immutable content, unlike a tag.
- **Index (manifest list)**: a manifest grouping several images, typically one per architecture, under a single reference.
- **Cache mount**: a directory BuildKit keeps between builds that is not part of any layer. Ideal for package manager caches.
- **Attestation**: a signed document attached to a digest asserting something about the image. The two most common are the SBOM (what it contains) and SLSA provenance (how it was built).
- **SBOM**: an inventory of the image's components, with each package's name, version and licence.
- **SLSA provenance**: a verifiable description of the build process: which source, which parameters, which builder.

- **VEX**: a signed statement about whether a specific CVE affects a specific product, with a justification from a closed vocabulary.
- **Distroless**: a family of base images containing runtime libraries and nothing else: no shell, no package manager, no utilities.
- **Keyless signing**: signing based on an ephemeral certificate issued against an OIDC identity and recorded in a transparency log, with no persistent private key.
- **Pull-through cache**: a registry acting as an on-demand mirror of an external registry, caching what has already been pulled.

Eight recurring mistakes

1. **Copying the code before installing dependencies.** The most expensive mistake and the easiest to fix. It invalidates the dependency layer on every commit.
2. **Assuming CI caching works because the log shows no error.** Without `mode=max` the build stages are not cached. Compare the real time of two consecutive no-change runs before believing it.
3. **Deleting a secret with RUN rm.** The previous layer is still in the image. Use `--mount=type=secret` or multi-stage.
4. **Alpine with Python.** Without manylinux wheels, your build compiles NumPy from source. Measure before migrating.
5. **Chaining apt-get install without cleaning in the same RUN.** An `rm -rf /var/lib/apt/lists/*` in a later `RUN` saves nothing; it has to be in the same instruction.
6. **USER with a name instead of a UID.** It breaks `runAsNonRoot` in Kubernetes with a confusing admission-time error.
7. **Shell-form CMD.** SIGTERM never reaches your process and every deploy drops in-flight requests.
8. **A single mutable tag such as `:latest`.** Impossible to know what is running and impossible to roll back deterministically. Tag by commit SHA and deploy by digest.

Production checklist

- `.dockerignore` present and verified with `docker build --progress=plain` (watch the transferred context size).
- Multi-stage build with a final stage based on `-slim` or distroless.
- Dependency manifests copied before the source code.
- `--mount=type=cache` on every package installation.
- Secrets via `--mount=type=secret`, never via `ARG`.
- Registry cache with `mode=max` in CI, or a persistent builder.
- Base image pinned by digest and updated automatically.
- Numeric `USER`, exec-form `ENTRYPOINT`/`CMD`, read-only root filesystem.
- Provenance and SBOM attached; scanning with Trivy or Grype in the pipeline, with a threshold that fails the build.
- Immutable tags by SHA; deploy by digest.

Frequently asked questions

Does reducing the number of layers still matter? Very little. With BuildKit and zstd compression the

cost of an extra layer is negligible, and grouping commands that change at different rates makes caching worse. The real exception is cleaning up inside the same `RUN` that made the mess.

Does distroless leave me without debugging? Not entirely. The `:debug` tags include a minimal shell, and in Kubernetes `kubectl debug --image=busybox --target=api` gives you an ephemeral container sharing the process namespace without touching the production image.

Should I squash or export a flattened image? Almost never worth it: it destroys caching and layer sharing between images, which is where the real savings in registry storage and pull time come from.

What about multi-architecture builds? `docker buildx build --platform linux/amd64,linux/arm64` works, but QEMU emulation is painfully slow for compiled languages. For serious builds use native nodes per architecture in the same builder, or cross-compile with `TARGETPLATFORM` and `BUILDPLATFORM`.

Where to start tomorrow

If you can only do one thing, reorder the Dockerfile so manifests are copied before the code, and add a `.dockerignore`. That is half an hour of work and usually halves the build. Once that is in place, add the cache mounts, then the lightweight final stage, and leave provenance and digest pinning for when the pipeline is already fast. Optimizing in that order means each step pays for itself before you start the next one.