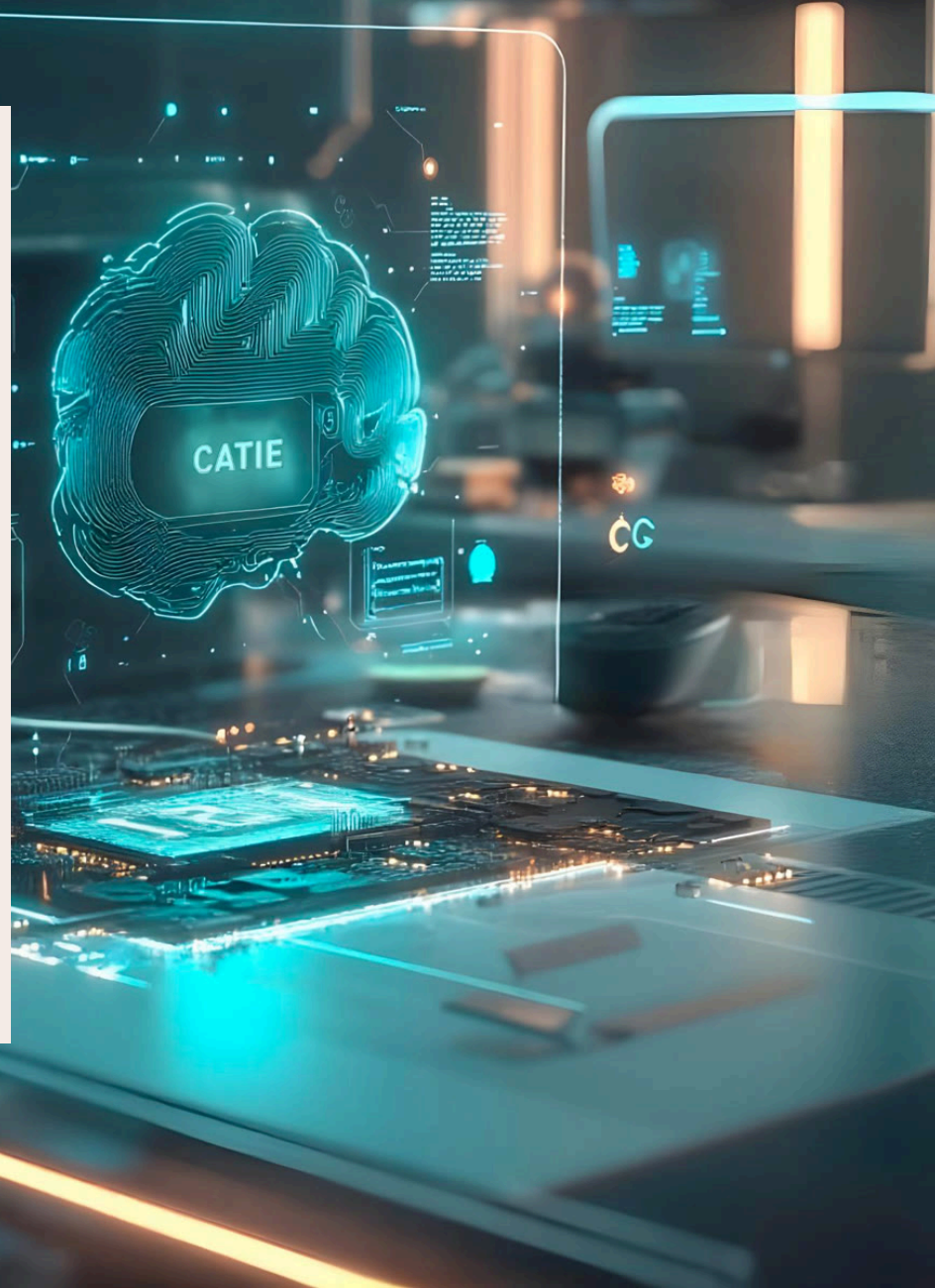




CATIE

Outils d'imagerie pour la
robotique
Conteneurisation avec Docker

28/10/2026



Plan

1. Pourquoi des conteneurs
 2. Utiliser Docker : commandes, données, matériel
 3. Un Dockerfile propre pour un projet uv, compose, publication
 4. TP : fil rouge, étape 4
-

Objectif de la séance

`docker run ghcr.io/<you>/<repo>` traite un dossier d'images sur n'importe quelle machine Linux, sans rien installer d'autre que Docker.



1. Pourquoi des conteneurs

« Ça marche sur ma machine »

Le fil rouge tourne chez vous. Il doit maintenant tourner chez un camarade, sur le serveur de calcul, dans la CI, sur le PC embarqué du robot... avec :

- la même version de Python et des bibliothèques (**uv.lock** s'en charge déjà)
- les mêmes bibliothèques système (libjpeg, libGL, CUDA...) : **pas** dans **uv.lock**
- la même commande de lancement, les mêmes chemins, les mêmes variables d'environnement

Une **image** Docker embarque tout cela : le code, l'environnement Python et le système de fichiers minimal qui va autour. On la construit une fois, on la lance partout où Docker est installé, comme un conteneur maritime standard passe sur n'importe quel bateau, train ou camion.

L'image est l'artefact du « build once, deploy many » de la séance 1 : on livre le logiciel avec son environnement, pas une notice d'installation.

Image, conteneur, registre

Image

Modèle en lecture seule, construit par **docker build** à partir d'un **Dockerfile** : une pile de **couches** (une par instruction **RUN**, **COPY**, **ADD**) plus des métadonnées (commande, variables, utilisateur).

Nom complet : **ghcr.io/me/fil-rouge:v1** = registre / propriétaire / nom : **tag** (étiquette de version, **latest** par défaut).

Les résultats s'écrivent dans un volume, jamais dans le conteneur.

Conteneur

Une instance de l'image en cours d'exécution (**docker run**), avec une couche inscriptible en plus.
Supprimer le conteneur supprime cette couche : rien n'y est durable.

Registre

Serveur qui stocke et distribue les images (**docker pull**, **docker push**) : Docker Hub par défaut, GitHub Container Registry (**ghcr.io**) lié au dépôt.

Conteneur ou machine virtuelle ?

	Machine virtuelle	Conteneur
Isole	Un système complet, avec son propre noyau	Un groupe de processus, noyau de l'hôte partagé
Mécanisme	Hyperviseur, matériel virtualisé	namespaces + cgroups du noyau Linux
Démarrage	Dizaines de secondes	Moins d'une seconde
Taille	Plusieurs Go	Quelques dizaines à centaines de Mo
Isolation	Forte	Plus faible : même noyau

- Un conteneur n'est qu'un **processus** de l'hôte : les **namespaces** restreignent ce qu'il voit (système de fichiers, réseau, PID), les **cgroups** ce qu'il consomme (CPU, mémoire)
- Sous macOS et Windows, Docker Desktop fait tourner... une VM Linux, dans laquelle tournent les conteneurs

Une VM est une maison entière, avec ses propres fondations ; un conteneur est un appartement : chacun chez soi, mais tous sur les fondations (le noyau) de l'immeuble.

Tags et digest

Pour aller plus loin

```
# registry/owner/name:tag
docker pull python:3.12-slim
docker pull ghcr.io/astral-sh/uv:0.11.8

docker tag fil-rouge \
  ghcr.io/me/fil-rouge:v1
docker push ghcr.io/me/fil-rouge:v1

docker pull python@sha256:...
```

- `python` = `docker.io/library/python` : Docker Hub, images officielles
- Un **tag** est une étiquette mobile : `latest` bouge à chaque push, `3.12-slim` à chaque correctif de Debian ou de Python
- Le **digest** (`@sha256:...`) désigne une image exacte, immuable : c'est lui qu'il faut noter pour une reproductibilité stricte
- `docker push` exige d'être authentifié sur le registre (`docker login ghcr.io`)



2. Utiliser Docker

Aide-mémoire : images et conteneurs

Images

```
docker build -t fil-rouge .  
docker image ls  
docker history fil-rouge # layers  
docker image rm fil-rouge  
docker pull / push <image>
```

Nettoyage

```
docker system df # disk usage  
docker system prune # all unused data
```

Conteneurs

```
docker run --rm -it python:3.12-slim  
docker run -d --name web -p 8080:80 nginx  
docker ps -a  
docker logs -f web  
docker exec -it web bash  
docker stop web && docker rm web
```

- `--rm` : supprimé à la fin
- `-it` : terminal interactif
- `-d` : en arrière-plan
- `-p host:container` : publie un port

Données : bind mounts et volumes

```
docker run --rm \
  -v "$PWD/data:/data/input:ro" \
  -v "$PWD/output:/data/output" \
  --user "$(id -u):$(id -g)" \
  fil-rouge /data/input /data/output
```

```
docker volume create model-cache
docker run -v model-cache:/cache ...
```

- **Bind mount** (host/path:container/path) :
un dossier de l'hôte, visible des deux côtés.
Idéal pour les données d'entrée et de sortie
- **:ro** : entrée en lecture seule, le conteneur ne peut pas l'abîmer
- **Volume nommé** : stockage géré par Docker, qui survit au conteneur. Le bind mount est un dossier partagé ; le volume, un disque dur externe confié à Docker
- **--user** : les fichiers écrits appartiennent à **vous**, pas à **root**

Matériel et configuration

Pour aller plus loin

```
# USB camera (Linux host only)
docker run --device /dev/video0 \
  --group-add video my-image
```

```
# NVIDIA GPU
docker run --gpus all my-image
```

```
# Environment variables
docker run -e LOG_LEVEL=DEBUG \
  --env-file .env my-image
```

```
# Limit resources
docker run --cpus 2 --memory 4g ...
```

- `--device` expose un périphérique ; `--group-add video` donne le droit de le lire à un utilisateur non-root
- `--gpus` demande le NVIDIA Container Toolkit sur l'hôte ; CUDA vient de l'image (roues PyTorch par exemple)
- `.env` contient souvent des secrets : jamais commité, jamais dans l'image
- Piège : sous `--cpus 2`, `os.cpu_count()` renvoie toujours le nombre de cœurs de l'hôte. Fixez `--workers`



3. Un Dockerfile propre pour un projet uv

Premier jet : le Dockerfile naïf

```
# Base image: Debian + Python 3.12
FROM python:3.12
# Command run at build time
RUN pip install uv
# Working directory in the image
WORKDIR /app
# Build context -> image
COPY . /app/
RUN uv sync
# Command started by docker run
CMD ["uv", "run", "fil-rouge", \
    "/data/input", "/data/output"]
```

Il fonctionne. Mais, mesuré sur le fil rouge :

- **1,9 Go** : image complète Debian + compilateurs + `.git`, `.venv` local, `data/...`
- `uv` non épinglé : pas reproductible
- Toute modification du code réinstalle **toutes** les dépendances (le `COPY .` invalide le cache)
- Les dépendances de dev (`pytest`, `ruff`) sont dans l'image
- Le processus tourne en **root**

Couches et cache

Au build, chaque couche est réutilisée si l'instruction et les fichiers qu'elle lit n'ont pas changé. Dès qu'une couche change, **toutes les suivantes** sont reconstruites : comme une pile d'assiettes, on ne remplace pas celle du milieu sans retirer celles du dessus.

Naïf, après modification de `pipeline.py` :

FROM python:3.12	en cache
------------------	----------

RUN pip install uv	en cache
--------------------	----------

COPY . /app/	refaite
--------------	---------

RUN uv sync (toutes les dépendances)	refaite
--------------------------------------	---------

Dépendances d'abord, même modification :

COPY pyproject.toml uv.lock ./	en cache
--------------------------------	----------

RUN uv sync --no-install-project	en cache
----------------------------------	----------

COPY . .	refaite
----------	---------

RUN uv sync (le projet seul)	refaite
------------------------------	---------

On ordonne le Dockerfile du plus stable au plus changeant : système, dépendances, puis code.

Multi-stage et utilisateur non-root

```
FROM python:3.12-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:0.11.8 /uv /bin/
WORKDIR /app
COPY pyproject.toml uv.lock ./
RUN uv sync --locked --no-install-project --no-dev
COPY . .
RUN uv sync --locked --no-dev --no-editable
```

```
FROM python:3.12-slim
RUN useradd --create-home --uid 1000 app
COPY --from=builder --chown=app:app \
    /app/.venv /app/.venv
ENV PATH="/app/.venv/bin:$PATH"
USER 1000
WORKDIR /data
ENTRYPOINT ["fil-rouge"]
CMD ["/data/input", "/data/output"]
```

- Deux **FROM** : seule la dernière étape est livrée ; uv et les sources restent dans **builder** : l'atelier reste à l'usine, seul le produit part
- uv copié depuis son image officielle, version épinglée ; **--locked** échoue si **uv.lock** n'est pas à jour
- **--no-dev** : ni pytest ni ruff ; **--no-editable** : **.venv** est autonome
- Base **-slim** : Debian minimale, sans compilateurs
- **USER** : une faille dans le code ne donne pas **root**
- **ENTRYPOINT** fixe la commande, **CMD** les arguments par défaut
- Fil rouge : **1,9 Go** en root → **300 Mo** en uid 1000

Le builder du fil rouge : caches de build

Pour aller plus loin

```
# syntax=docker/dockerfile:1
FROM python:3.12-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:0.11.8 \
    /uv /uvx /bin/
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy \
    UV_PYTHON_DOWNLOADS=0
WORKDIR /app

# 1. Dependencies only: stable layer
RUN --mount=type=cache,target=/root/.cache/uv \
    --mount=type=bind,source=uv.lock,target=uv.lock \
    --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
    uv sync --locked --no-install-project \
        --no-dev --no-editable

# 2. The code: frequently changing layer
COPY . /app
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --locked --no-dev --no-editable
```

- `--mount=type=cache` : le cache uv survit d'un build à l'autre, sans finir dans l'image
- `--mount=type=bind` : `pyproject.toml` et `uv.lock` lus sans créer de couche
- `UV_COMPILE_BYTECODE` : `.pyc` précompilés, démarrage plus rapide
- `UV_LINK_MODE=copy` : nécessaire avec un cache monté
- `UV_PYTHON_DOWNLOADS=0` : le Python de l'image, jamais un autre
- `docker run fil-rouge /in /out --workers 4` remplace tout CMD

.dockerignore : le contexte de build

```
.git
.github
.venv
.pytest_cache
.ruff_cache
**/__pycache__
dist
data
output
tests
compose.yaml
Dockerfile
```

`docker build .` envoie tout le dossier `.` au moteur : c'est le **contexte**.

- Sans `.dockerignore`, le `.venv` local, les données et l'historique Git partent dans le contexte... et dans l'image avec **COPY** .
- Un `.venv` de l'hôte copié dans l'image écrase celui construit par uv : binaires et chemins incompatibles
- Moins de fichiers dans le contexte = moins d'invalidations du cache
- Même logique que `.gitignore`, mêmes motifs

docker compose : décrire le lancement

```
# compose.yaml
services:
  pipeline:
    build: .
    image: fil-rouge:latest
    user: "${HOST_UID:-1000}:${HOST_GID:-1000}"
    volumes:
      - ./data:/data/input:ro
      - ./output:/data/output
    environment:
      - PYTHONUNBUFFERED=1
    command: ["/data/input",
              "/data/output",
              "--workers", "4"]
```

```
docker compose build
docker compose run --rm --build pipeline
docker compose up -d # services longs
docker compose logs -f
docker compose down
```

- Les options de `docker run` deviennent un fichier versionné
- Plusieurs services (base de données, API, pipeline) dans un même fichier, sur un réseau commun : chacun joint les autres par leur nom de service
- Commande `docker compose` (v2) ; clé `version`: obsolète

Publier sur GHCR avec GitHub Actions

```
# .github/workflows/docker.yml
on:
  push:
    branches: [main]
jobs:
  image:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v5
      - uses: hadolint/hadolint-action@v3.1.0
      - uses: docker/login-action@v4
    with:
      registry: ghcr.io
      username: ${github.actor}
      password: ${secrets.GITHUB_TOKEN}
```

```
- uses: docker/setup-buildx-action@v4
- id: meta
  uses: docker/metadata-action@v6
  with:
    images: ghcr.io/${github.repository}
    tags: |
      type=sha
      type=raw,value=latest
- uses: docker/build-push-action@v7
  with:
    push: true
    tags: ${steps.meta.outputs.tags}
    cache-from: type=gha
    cache-to: type=gha,mode=max
```

- Les **steps** sont désindentés ici pour tenir sur la slide
- **hadolint** : le ruff des Dockerfile, le build échoue sur un avertissement
- **GITHUB_TOKEN** suffit : aucun secret à créer
- Tag **sha-...** par commit, plus **latest** ; cache **gha** entre deux CI

hadolint en local

Pour aller plus loin

```
docker run --rm -i hadolint/hadolint < Dockerfile
```

Sur le Dockerfile naïf :

```
-:4 DL3013 warning: Pin versions in pip. Instead of `pip install <package>`  
    use `pip install <package>==<version>` or `pip install --requirement ...`  
-:4 DL3042 warning: Avoid use of cache directory with pip.  
    Use `pip install --no-cache-dir <package>`
```

Avec **USER app** au lieu de **USER 1000** :

```
-:36 DL3066 info: Non-numeric user-id may not be resolvable by host system
```

- Chaque règle (DL...) est documentée et peut être ignorée explicitement
- Intégrable en pre-commit, comme ruff

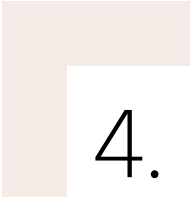
À retenir : utiliser Docker

Besoin	Réflexe	En une ligne
Construire une image	<code>docker build</code> avec un tag	<code>docker build -t fil-rouge .</code>
Lancer un conteneur jetable	<code>run --rm</code>	<code>docker run --rm -it python:3.12-slim</code>
Donner accès à des fichiers	Bind mount	<code>-v "\$PWD/data:/data/input:ro"</code>
Écrire des fichiers à son nom	<code>--user</code>	<code>--user "\$(id -u):\$(id -g)"</code>
Passer une configuration	Variable d'environnement	<code>-e LOG_LEVEL=DEBUG</code>
Comprendre ce qui se passe	<code>logs</code> , <code>exec</code>	<code>docker exec -it web bash</code>

À retenir : écrire un Dockerfile

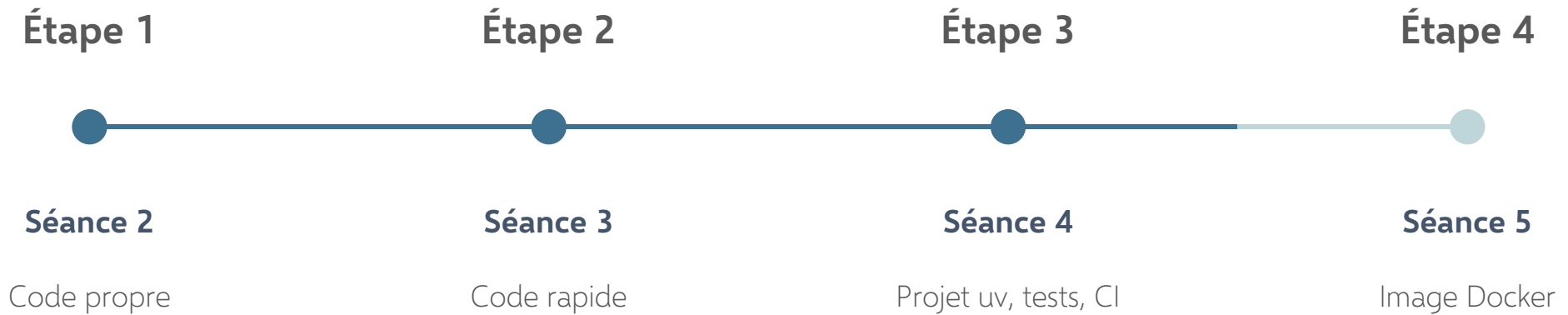
Besoin	Réflexe	En une ligne
Profiter du cache	Dépendances avant le code	<code>COPY pyproject.toml uv.lock ./</code>
Livrer une image légère	Multi-stage, base <code>-slim</code>	<code>FROM python:3.12-slim AS builder</code>
Ne pas tourner en root	Utilisateur dédié	<code>USER 1000</code>
Alléger le contexte de build	<code>.dockerignore</code>	<code>.git, .venv, **/__pycache__</code>
Décrire le lancement	<code>compose.yaml</code>	<code>docker compose run --rm pipeline</code>
Partager l'image	GHCR, publiée par la CI	<code>ghcr.io/<account>/fil-rouge</code>

Une image construite une fois tourne à l'identique partout.



4. TP : fil rouge, étape 4

TP : de l'étape 3 à l'étape 4



■ Fait ■ Aujourd'hui

Point de départ : votre **etape_3/** (ou le corrigé de la séance 3, sur le site), sur votre dépôt GitHub, avec un dossier **data/** d'images. Pour vérifier l'installation : **docker run hello-world**.

TP : construire, mesurer, optimiser

1. Écrivez le Dockerfile **naïf** vu en cours. Construisez, lancez sur **data/** avec deux bind mounts, **sans** `--user` (uv ne pourrait pas créer son cache). Notez la taille (`docker image ls`) et l'utilisateur (`docker run --rm --entrypoint id <image>`)
2. Modifiez une ligne de `pipeline.py`, reconstruisez : quelles étapes sont **CACHED** ? (`docker build --progress=plain`)
3. Ajoutez `.dockerignore` ; séparez dépendances et code ; épinglez uv en le copiant depuis `ghcr.io/astral-sh/uv`. Refaites la mesure 2
4. Passez en multi-stage, `--no-dev --no-editable`, utilisateur non-root.
Checkpoint : image sous 350 Mo, `id -u` ne renvoie plus 0
5. Lancez avec `--user "$(id -u):$(id -g)"` : à qui appartiennent les fichiers de `output/` ? Et ceux écrits par l'image naïve à l'étape 1 ?

TP : compose, hadolint, publication

6. Écrivez `compose.yaml` ; `docker compose run --rm --build pipeline` doit produire la même archive que `uv run fil-rouge`
7. `hadolint` : zéro avertissement
8. Ajoutez `.github/workflows/docker.yaml` ; poussez sur `main` ; vérifiez l'image dans l'onglet **Packages** du dépôt
9. **Checkpoint final** : un camarade lance **votre** image avec `docker run ghcr.io/<you>/<repo>` sur ses propres données, sans cloner le dépôt

Bonus : comparez les temps `--executor processes` dans et hors du conteneur, avec et sans `--cpus 2` ; ajoutez les caches de build du fil rouge (`--mount=type=cache`) et mesurez le gain ; ajoutez un service à `compose.yaml` qui lance les tests avec les dépendances de dev ; construisez l'image pour `linux/arm64` avec `docker buildx`.

À retenir

Image = environnement figé

Code, Python, dépendances et système, construits une fois, lancés partout

Image légère et sûre

Multi-stage, base `-slim`, `.dockerignore`, utilisateur non-root

Ordre des couches

Du plus stable au plus changeant : dépendances, puis code

Données dehors

Entrées et sorties en bind mounts, secrets en variables d'environnement

QUESTIONS ?