

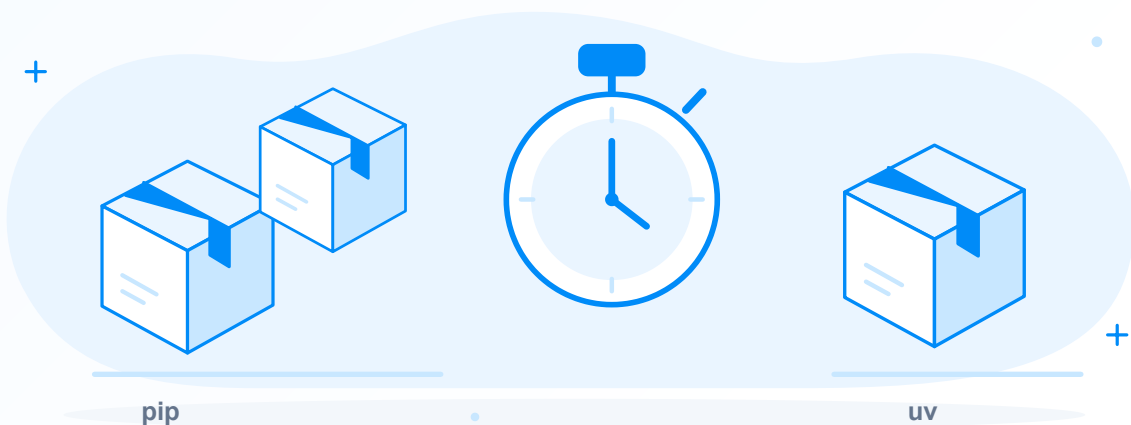
Mini-série

Python · CI/CD · Performance des outils

Pip, uv et alternatives : benchmarker un service Python professionnel

Mesurer pip, uv, pip-tools, Poetry, PDM et Conda/Mamba sur un service API de routage de tickets, puis choisir un workflow adapté à la production.

Auteur **Youness Boutyour**

**DURÉE**

4 épisodes · 45 min /
épisode

NIVEAU

Intermédiaire

FORMAT

Mini-série · 4 épisodes

Python · CI/CD · Performance des outils · publié le 2026-04-23 · local

Objectifs et livrables

Installer les dépendances d'un projet Python paraît simple jusqu'au moment où l'équipe doit reconstruire le même service tous les jours en CI. Les temps d'installation, la gestion des versions et la maintenance des environnements deviennent alors des choix concrets. Cette mini-série vous propose de les explorer à partir d'une petite API de routage de tickets support.

Nous allons construire et tester le service, puis comparer les workflows `pip` / `pip-tools` et `uv` en conservant le même code applicatif. Nous examinerons aussi Poetry, PDM, Pipenv et Conda/Mamba pour comprendre les fonctionnalités qu'ils apportent selon le contexte. Le parcours couvre les fichiers de verrouillage, les dépendances de développement, la CI et la construction d'une image Docker.

Vous apprendrez à distinguer la résolution, l'installation à froid et la synchronisation avec cache. Les mesures seront accompagnées de leurs conditions d'exécution et de répétitions, pour pouvoir expliquer les écarts observés. Nous mesurerons la préparation et la livraison du service ; sa latence HTTP relève d'une autre expérience.

Le résultat sera un rapport de comparaison avec vos mesures, les fonctionnalités examinées et une recommandation adaptée à l'équipe. Les valeurs d'exemple vous aideront à lire les sorties ; les outils ou étapes que vous n'aurez pas mesurés resteront identifiés comme tels.

RÉSULTATS DES ÉTAPES · BENCHMARK

01 · SERVICE

```
GET /health
status: ok

POST /route
queue: incident-response

✓ même payload pour tous
```

02 · WORKFLOWS

```
pip-tools → lock
uv → uv.lock
pytest → PASS

cache cold ≠ warm
mêmes versions et dépôt
```

03 · MESURES

```
benchmarks.csv
cold: 2.41 s
warm: 0.32 s
docker: 4.87 s

exemple — mesurer localement
```

Étapes de mesure, comparaison et décision.

RÉSULTAT DE RÉFÉRENCE · BENCHMARK

Même service · même machine · caches séparés

Workflow	Résolution	Installation froide	Tests	Docker
pip-tools	à mesurer	à mesurer	PASS	à mesurer
uv	à mesurer	à mesurer	PASS	à mesurer

La figure devient quantitative après l'exécution de scripts/plot-benchmarks.py.

Ne pas transformer une mesure locale en classement universel.

Résultats de référence à remplacer par les mesures locales.

Profil qualitatif — à confirmer par vos mesures

Échelle indicative : 1 = limité dans ce critère · 5 = fort dans ce critère

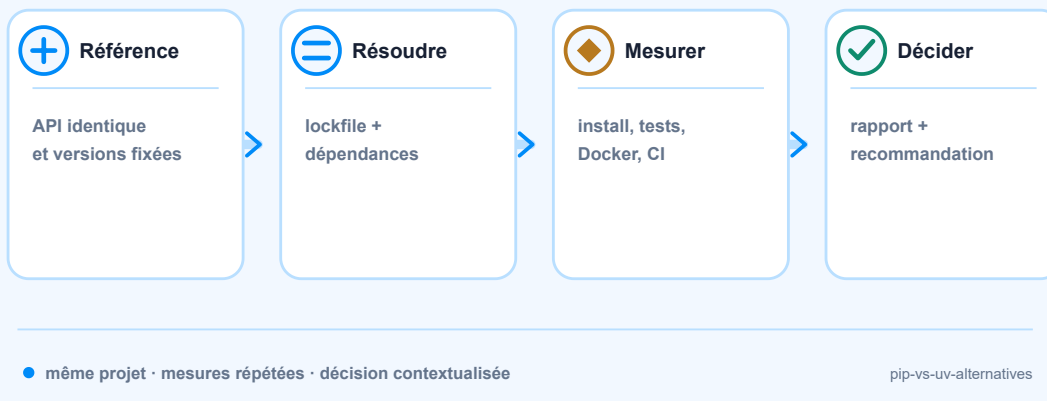
Outil	Simplicité	Lockfile	Projet	CI	Système/GPU
pip + venv	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
pip-tools	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
uv	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
Poetry	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
PDM	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
Conda/Mamba	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>

Ce graphique sert à formuler des hypothèses ; le benchmark du tutoriel doit les confirmer ou les corriger.

Matrice de fonctionnalités à compléter avec le contexte de l'équipe.

Comparer des workflows sur une base commune

Schéma du protocole de benchmark qui garde le même service et compare résolution, installation, tests, Docker et CI.



La comparaison reste interprétable parce que le service et les conditions de mesure restent constants.

Lire le protocole avant de mesurer

Le schéma commence par un projet identique pour éviter de comparer des workflows sur des charges différentes. La résolution répond à la question « quelles versions sont compatibles ? », l'installation mesure le téléchargement et la préparation de l'environnement, et les tests vérifient que le résultat reste fonctionnel. Docker et la CI ajoutent la perspective de livraison.

Un temps plus court ne suffit pas à choisir un outil. Notez la machine, l'état du cache, les versions et le nombre de répétitions, puis regardez aussi la qualité du lockfile, la facilité de mise à jour et les besoins de l'équipe. Le rapport final doit distinguer une mesure observée d'une fonctionnalité simplement déclarée dans la documentation.

1. Préparer l'environnement

Définissez ce qui entre dans chaque chronométrage avant de lancer les outils. Résoudre les versions, télécharger les paquets et installer un environnement sont des opérations différentes, même lorsqu'une seule commande les enchaîne. Gardez une note de la commande complète et de l'état initial pour éviter d'attribuer au solveur un gain qui vient seulement d'un cache déjà rempli.

Fixez aussi les critères de validité : mêmes sources, même version Python, mêmes tests et réponse HTTP conforme. Un workflow plus rapide qui ne démarre pas le service ne constitue pas une amélioration. Les durées et diagrammes d'exemple servent à préparer la lecture du rapport ; seules vos exécutions documentées permettront de conclure pour la machine et le projet utilisés.

Mesurez chaque workflow sur la même machine, avec la même version de Python et le même dépôt :

Métrique	Question professionnelle	Unité
Résolution / lock	Combien de temps pour calculer les versions compatibles ?	secondes
Installation froide	Combien de temps sans environnement ni cache local ?	secondes
Synchronisation chaude	Combien de temps après modification du code, sans changement de dépendances ?	secondes
Test HTTP	Le service reste-t-il fonctionnel après reconstruction ?	PASS / FAIL
Build Docker	Le workflow est-il compatible avec une livraison conteneurisée ?	secondes
Taille du lockfile	Quelle trace de résolution est stockée dans Git ?	Ko
Reproductibilité	Une seconde installation produit-elle le même résultat ?	hash / PASS

Ce graphique est une hypothèse de travail, pas un résultat de benchmark. Les valeurs quantitatives doivent être produites par vos propres mesures et publiées avec l'OS, la version Python, la date et l'état du cache.

Utiliser les comparatifs externes sans les transformer en résultats locaux

Les références en fin de tutoriel servent à construire une hypothèse et un protocole, non à remplir [benchmarks.csv](#) à votre place. Real Python propose par exemple de comparer deux environnements propres après avoir vidé les caches, et rapporte sur son propre scénario JupyterLab **21,409 s** avec pip contre **2,618 s** avec [uv pip](#) ; cet écart est un point de départ, pas une promesse pour votre API ni pour votre runner CI.

L'analyse Medium met en avant la résolution parallèle, le cache global et la compatibilité [uv pip](#) avec les flux [requirements.txt](#). Le guide AppSignal élargit ensuite le périmètre : [uv](#) peut aussi gérer le projet, les environnements, les versions de Python, l'exécution et le verrouillage. Enfin, le comparatif DZone aide à poser les alternatives : [pip](#) reste un socle largement compatible, Poetry conserve un workflow de publication mature, tandis que [uv](#) est à évaluer pour les nouveaux services, la CI et les dépôts multi-projets. Ces quatre lectures sont référencées explicitement à la fin de ce guide.

La règle de décision est donc simple : conservez les chiffres externes comme **repères documentés**, puis publiez vos propres résultats avec le commit testé, le système, les versions des outils, les sources de paquets, la température du cache et le nombre de répétitions.

Sortie obtenue

Sortie / exemple

Même dépôt · même version Python · même payload HTTP · caches documentés

Commande de vérification

Notez l'OS, la version Python, le processeur, les outils testés et la stratégie de cache dans `benchmark-notes.md`. Sans ce contexte, les temps ne sont pas comparables.

2. Construire le service de référence

Le routage est déterministe : les règles de priorité et de mots-clés produisent toujours la même décision pour une même entrée. Cela enlève la variabilité d'un modèle de langue à cette expérience, dont le sujet est la préparation de l'environnement Python. Les modèles Pydantic définissent le contrat HTTP ; les tests vérifient le service, un routage prioritaire et le refus d'une priorité invalide.

Exécutez d'abord les tests, puis les requêtes HTTP dans un second terminal pendant qu'Uvicorn tourne. `PYTHONPATH=src` rend le module du service importable depuis l'arborescence choisie. Conservez cette configuration dans les copies du projet : un échec d'import dû au chemin des sources ne doit pas être interprété comme une défaillance du gestionnaire de dépendances.

Créez le dépôt applicatif :

```
Terminal · Bash

mkdir -p ticket-routing-api/src/ticket_routing ticket-routing-api/tests ticket-routing-api/scripts
cd ticket-routing-api
git init
python3 -m venv .venv-reference
source .venv-reference/bin/activate
python -m pip install --upgrade pip
python -m pip install fastapi 'uvicorn[standard]' httpx pytest
python -m pip freeze > requirements-reference.txt
```

Créez `src/ticket_routing/app.py` :

```
Python

from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI(title="Ticket Routing API", version="1.0.0")
```

```
class Ticket(BaseModel):
    subject: str = Field(min_length=3, max_length=200)
    body: str = Field(min_length=5, max_length=5000)
    priority: str = Field(pattern="^(low|normal|high|critical)$")

class RoutingDecision(BaseModel):
    queue: str
    priority: str
    reason: str

def route_ticket(ticket: Ticket) -> RoutingDecision:
    text = f"{ticket.subject} {ticket.body}".casefold()
    if ticket.priority == "critical" or any(word in text for word in ("outage", "security", "breach")):
        return RoutingDecision(queue="incident-response", priority=ticket.priority, reason="incident keyword")
    if any(word in text for word in ("invoice", "payment", "billing")):
        return RoutingDecision(queue="finance-support", priority=ticket.priority, reason="billing keyword")
    return RoutingDecision(queue="customer-support", priority=ticket.priority, reason="default route")

@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "ok", "service": "ticket-routing"}

@app.post("/route", response_model=RoutingDecision)
def route(ticket: Ticket) -> RoutingDecision:
    return route_ticket(ticket)
```

Créez `tests/test_api.py` :

```
● ● ● Python

from fastapi.testclient import TestClient

from ticket_routing.app import app

client = TestClient(app)

def test_health() -> None:
    response = client.get("/health")
    assert response.status_code == 200
    assert response.json()["status"] == "ok"
```

```
def test_critical_ticket_goes_to_incident_response() → None:
    response = client.post("/route", json={
        "subject": "Production outage",
        "body": "The API is unavailable for all customers.",
        "priority": "critical",
    })
    assert response.status_code == 200
    assert response.json()["queue"] == "incident-response"

def test_invalid_priority_is_rejected() → None:
    response = client.post("/route", json={
        "subject": "Question",
        "body": "I need help.",
        "priority": "urgent",
    })
    assert response.status_code == 422
```

Lancez le service et les tests :

Terminal · Bash

```
PYTHONPATH=src python -m pytest -q
uvicorn ticket_routing.app:app --app-dir src --host 127.0.0.1 --port 8000
```

Dans un second terminal :

Terminal · Bash

```
curl http://127.0.0.1:8000/health
curl -X POST http://127.0.0.1:8000/route \
-H 'content-type: application/json' \
-d '{"subject":"Invoice rejected","body":"The payment was refused","priority":"high"}
```

Arrêtez ensuite Uvicorn avec `Ctrl+C`. Cette API, son contrat HTTP et ses tests deviennent la référence immuable de toutes les comparaisons.

Sortie obtenue

Sortie / exemple

```
3 passed
HTTP /health → 200
HTTP /route → queue finance-support
```

Commande de vérification

Conservez la sortie des tests et les deux réponses HTTP. Elles constituent le contrat de comparaison pour pip, uv et les alternatives.

3. Préparer l'image Docker commune

Le Dockerfile sert de point fixe : tous les workflows fournissent un fichier de dépendances résolu à la même recette. Comme l'installation dans cette image reste effectuée par pip, son temps de construction ne compare pas directement les installateurs pip et uv à l'intérieur du conteneur. Il mesure la livraison des exports dans un contexte commun.

Libérez le port 8000 avant de démarrer le conteneur de test et vérifiez qu'il répond avant de le supprimer. Notez séparément l'état du cache Docker et celui des caches de paquets sur l'hôte. Une couche Docker réutilisée peut éviter une installation entière : son temps ne doit pas être comparé à celui d'une reconstruction qui télécharge et installe tout.

Créez un [Dockerfile](#) commun qui reçoit le fichier de dépendances déjà résolu. Il permet de comparer les images à code, image de base et commande d'exécution identiques :

```
FROM python:3.12-slim

WORKDIR /app
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1 PYTHONPATH=/app/src

ARG REQUIREMENTS_FILE=requirements-reference.txt
COPY ${REQUIREMENTS_FILE} requirements.txt
RUN python -m pip install --no-cache-dir -r requirements.txt
COPY src ./src

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

Construisez et vérifiez l'image de référence :

```
docker build --tag ticket-routing:pipe-reference .
docker run --rm -d --name ticket-routing-test -p 8000:8000 ticket-routing:pipe-reference
curl --fail http://127.0.0.1:8000/health
docker rm --force ticket-routing-test
```

Le temps du `docker build` doit être mesuré séparément. Le premier build et le build avec cache ne répondent pas à la même question.

La construction d'image mesure ici la portabilité de l'artefact résolu, non la vitesse d'un solveur dans Docker. Mesurez séparément le temps de résolution et de synchronisation sur l'hôte ; ne mélangez pas le cache des couches Docker avec les caches pip ou uv.

Sortie obtenue

■ Sortie / exemple

```
curl /health → HTTP 200  
docker rm --force ticket-routing-test → conteneur supprimé
```

Commande de vérification

Capturez le terminal du build et de la vérification HTTP. Notez le tag `ticket-routing:pip-reference` et le temps du build de référence avant de comparer les workflows.

4. Workflow `pip` et `pip-tools`

`requirements.in` exprime les dépendances que vous choisissez directement ; la compilation ajoute les versions compatibles et les dépendances transitives. Relisez le fichier produit avant la synchronisation : c'est lui qui décrit l'environnement à reproduire. Cette distinction permet de faire évoluer une intention de projet sans éditer manuellement chaque dépendance indirecte.

`pip-sync` rapproche l'environnement du fichier résolu ; travaillez donc dans l'environnement dédié à cet essai. Après synchronisation, `pip check` vérifie la cohérence des dépendances installées, tandis que les tests vérifient le comportement de l'API. Ces contrôles ne sont pas interchangeables : un ensemble de paquets cohérent peut encore être incompatible avec le code de votre application.

Pour `pip`, déclarez les dépendances directes dans `requirements.in` :

■ Sortie / exemple

```
fastapi  
uvicorn[standard]  
httpx  
pytest  
matplotlib
```

Puis produisez un fichier résolu :

 Terminal · Bash

```
python3 -m venv .venv-pip
source .venv-pip/bin/activate
python -m pip install pip-tools
pip-compile requirements.in --output-file requirements-pip.txt
pip-sync requirements-pip.txt
PYTHONPATH=src python -m pytest -q
python -m pip check
deactivate
```

`requirements-pip.txt` est l'artefact à examiner : il contient les versions résolues et leurs dépendances. `pip-tools` ajoute une étape de compilation mais conserve l'installation avec pip. Cette séparation entre dépendances directes (`requirements.in`) et graphe résolu (`requirements-pip.txt`) évite d'utiliser un simple `pip freeze` comme source éditable du projet.

Reconstruisez le conteneur avec le même code source et les mêmes tests :

 Terminal · Bash

```
docker build \
  --build-arg REQUIREMENTS_FILE=requirements-pip.txt \
  --tag ticket-routing:pip-tools .
```

Sortie obtenue

 Sortie / exemple

```
requirements-pip.txt → fichier résolu créé
pip-sync → environnement synchronisé
pytest → 3 passed
pip check → aucune dépendance cassée
```

Commande de vérification

Conservez `requirements.in`, `requirements-pip.txt`, la sortie de `pip check` et l'image `ticket-routing:pip-tools`. Le fichier résolu doit être associé au commit testé.

5. Workflow

Faites la copie depuis le dossier parent, puis vérifiez votre emplacement avant de nettoyer les environnements de cette copie. Le dépôt de référence doit rester intact pour les comparaisons. Le workflow projet crée ses déclarations dans `pyproject.toml` et sa résolution dans `uv.lock` ; les dépendances de développement restent distinguées de celles nécessaires au service.

Le mode `uv pip` constitue un autre chemin : il consomme un fichier de dépendances existant sans démontrer les mêmes fonctions de gestion de projet. Identifiez clairement le mode dans les mesures. Avec `-locked`, l'objectif est de vérifier le fichier de verrouillage existant, pas de le recalculer discrètement à chaque reconstruction ; gardez sa mise à jour comme une opération explicite et relue.

Créez une copie du service sans modifier le dépôt de référence :

```
Terminal · Bash

cd ..
cp -R ticket-routing-api ticket-routing-api-uv
cd ticket-routing-api-uv
rm -rf .venv-reference .venv-pip .git
rm -f pyproject.toml uv.lock requirements-*.txt
rm -f requirements.in
uv init --bare
uv add fastapi 'uvicorn[standard]' httpx
uv add --dev pytest matplotlib
PYTHONPATH=src uv run pytest -q
uv lock --check
```

Le fichier `uv.lock` doit être versionné. Pour reproduire l'environnement dans la CI :

```
Terminal · Bash

uv sync --locked
PYTHONPATH=src uv run --locked pytest -q
```

Pour un dépôt existant qui doit continuer à utiliser `requirements.txt`, le mode de compatibilité est utile :

```
Terminal · Bash

uv venv .venv-uv-pip
uv pip install --python .venv-uv-pip/bin/python -r ../ticket-routing-api/requirements-reference.txt
uv pip freeze --python .venv-uv-pip/bin/python
```

Comparez les deux formes de workflow : `uv add` / `uv lock` / `uv run` d'un côté, `uv pip` / `requirements.txt` de l'autre. Elles ne répondent pas exactement au même besoin opérationnel.

Pour garder le conteneur comparable au workflow pip-tools, exportez le graphe uv vers un fichier d'exécution puis construisez la même image :

 Terminal · Bash

```
uv export --locked --format requirements-txt --no-emit-project -o requirements-uv.txt
docker build \
  --build-arg REQUIREMENTS_FILE=requirements-uv.txt \
  --tag ticket-routing:uv-export .
```

Cette stratégie garde `uv.lock` comme trace de résolution et utilise le fichier exporté uniquement comme interface de livraison pour cette image Docker. Conservez les deux fichiers dans le rapport : ils répondent à deux usages distincts.

Sortie obtenue

 Sortie / exemple

```
uv lock --check → vérification réussie
uv run --locked pytest -q → tests réussis
export requirements → fichier de livraison créé
```

Commande de vérification

Conservez `uv.lock`, le fichier exporté et la sortie de `uv run`. Le même endpoint HTTP doit répondre après reconstruction.

6. Fonctionnalités des autres alternatives

Préparez chaque alternative dans sa propre copie avec les fichiers de configuration qu'elle attend. Les commandes ci-dessous vérifient un projet déjà déclaré pour l'outil concerné ; elles ne convertissent pas automatiquement tous les formats des étapes précédentes. Gardez le même code applicatif, les mêmes dépendances fonctionnelles et les mêmes conditions de test.

La matrice sert à consigner des capacités vérifiées, pas à attribuer un score universel. Une fonction disponible peut dépendre d'un plugin, d'un format d'export ou d'un contexte précis. Pour chaque case importante pour votre équipe, conservez la configuration et le contrôle exécuté ; si vous n'avez pas préparé ou testé cette alternative, indiquez-le dans le rapport plutôt que de lui attribuer une durée.

Cette étape mesure les capacités qui ne sont pas résumées par le temps d'installation. Utilisez la même copie du service et conservez les fichiers produits.

Fonctionnalité	pip-tools	uv	Poetry	PDM	Pipenv	Conda/Mamba
Lockfile natif	Oui, via compilation	Oui	Oui	Oui	Oui	Partiel selon les canaux

Fonctionnalité	pip-tools	uv	Poetry	PDM	Pipenv	Conda/Mamba
Groupes dev	fichiers séparés	Oui	Oui	Oui	Oui	environnements / specs
Gestion Python	externe	intégrée	externe	externe	externe	intégrée
Publication de librairie	séparée	intégrée	intégrée	intégrée	secondaire	non central
Paquets non-Python	non	non	non	non	non	Oui
Export vers requirements	natif	Oui	Oui	Oui	Oui	possible
Intégration CI	simple	simple et verrouillée	simple	simple	existante	plus lourde

Les mentions **Oui** indiquent une capacité à vérifier dans la documentation et dans votre dépôt, pas une promesse de comportement identique. Testez notamment les groupes de développement, les dépendances privées, les versions Python et le mode hors ligne.

Sortie obtenue

Sortie / exemple

Matrice complétée : lockfile · groupes dev · version Python · export · CI
Chaque capacité est reliée à une commande ou à une référence.

Commande de vérification

Ajoutez au rapport les fichiers de configuration produits par chaque outil et la commande de validation exécutée. Une fonctionnalité non testée reste « non vérifiée ».

Poetry, PDM et Pipenv

Dans les copies correspondantes du dépôt, vérifiez les mêmes tests :

Terminal · Bash

poetry install
poetry run pytest -q
poetry check

pdm install --check

```
pdm run pytest -q
pdm lock --check

pipenv install --dev
pipenv run pytest -q
pipenv verify
```

Poetry et PDM sont à comparer si le dépôt est une librairie publiable ou si l'équipe veut un projet centré sur `pyproject.toml`. Pipenv est surtout pertinent lorsqu'un dépôt existant l'utilise déjà.

Conda / Mamba

Pour un service qui dépend de bibliothèques système, créez une spécification explicite :

```
name: ticket-routing
channels:
  - conda-forge
dependencies:
  - python=3.12
  - pip
  - pip:
    - fastapi
    - uvicorn[standard]
    - httpx
    - pytest
```

Puis exécutez :

```
conda env create --file environment.yml
conda run --name ticket-routing python -m pytest -q
conda env export --from-history > environment-history.yml
```

Conda/Mamba doit être évalué lorsque le service utilise CUDA, BLAS ou des bibliothèques natives. Pour une API Python pure, mesurez aussi le coût de distribution de l'environnement et de l'image Docker.

7. Automatiser les mesures et les graphiques

Consignez chaque répétition avant de produire une synthèse. Une ligne de mesure doit rester reliée à sa commande, à son outil, à son état de cache et à son statut fonctionnel. Les marqueurs `à_mesurer` évitent de confondre une case encore vide avec une durée nulle ; remplacez-les par des nombres uniquement

après une exécution réelle.

Le script de graphique trace les lignes numériques retenues, mais ne calcule pas lui-même les médianes. Pour afficher une barre de médiane par outil, fournissez-lui un tableau de synthèse calculé à partir des répétitions et conservez les données brutes séparément. Une ligne **FAIL** reste utile au diagnostic même si elle est exclue du graphique : un temps rapide ne doit pas masquer un service non fonctionnel.

Sortie obtenue

Sortie / exemple

benchmarks.csv → une ligne par outil et répétition
benchmarks.svg → médianes froides, chaudes et Docker

Commande de vérification

Le graphique doit être produit à partir des mesures locales et accompagné de la machine, des versions, du cache et du commit testé. Les valeurs à mesurer ne constituent pas un résultat.

Créez un tableau **benchmarks.csv** avec une ligne par outil et une exécution documentée :

Aperçu du fichier de mesures

Cet extrait présente la structure attendue avant de remplacer **à_mesurer** par des valeurs réellement observées. Il ne constitue pas un benchmark exécuté.

Outil	Python	Résolution (s)	Froid (s)	Chaud (s)	Tests (s)	Docker (s)	Lockfile (Ko)	Résultat
pip	3.12	à mesurer	à mesurer	à mesurer	à mesurer	à mesurer	à mesurer	PASS
uv	3.12	à mesurer	à mesurer	à mesurer	à mesurer	à mesurer	à mesurer	PASS

CSV

```
tool,python,lock_seconds,cold_seconds,warm_seconds,test_seconds,docker_seconds,lockfile_kb,result  
pip,3.12,à_mesurer,à_mesurer,à_mesurer,à_mesurer,à_mesurer,à_mesurer,PASS  
uv,3.12,à_mesurer,à_mesurer,à_mesurer,à_mesurer,à_mesurer,à_mesurer,PASS
```

Mesurez avec **/usr/bin/time** ou un outil équivalent :

Terminal · Bash

```
/usr/bin/time -f '%e' pip-compile requirements.in -o requirements-pip.txt  
/usr/bin/time -f '%e' pip-sync requirements-pip.txt  
/usr/bin/time -f '%e' uv sync --locked
```

```
/usr/bin/time -f '%e' python -m pytest -q
/usr/bin/time -f '%e' docker build --tag ticket-routing:benchmark .
```

Réalisez au moins trois répétitions par mesure froide et chaude, puis publiez la médiane et les valeurs brutes. Une mesure froide commence après suppression de l'environnement et nettoyage du cache de l'outil concerné ; une mesure chaude conserve le cache mais recrée l'environnement. Ne mélangez pas les caches pip, uv, Docker et Conda.

Créez `scripts/plot-benchmarks.py` :

```
Python

from __future__ import annotations

import csv
import sys
from pathlib import Path

import matplotlib.pyplot as plt

rows = list(csv.DictReader(Path(sys.argv[1]).open(encoding="utf-8")))
rows = [row for row in rows if row["result"] == "PASS" and row["cold_seconds"] != "à_mesurer"]
if not rows:
    raise SystemExit("Remplissez benchmarks.csv avec des mesures PASS.")

labels = [row["tool"] for row in rows]
cold = [float(row["cold_seconds"]) for row in rows]
warm = [float(row["warm_seconds"]) for row in rows]
docker = [float(row["docker_seconds"]) for row in rows]
x = range(len(labels))
width = 0.25

fig, ax = plt.subplots(figsize=(9, 4.8))
ax.bar([i - width for i in x], cold, width, label="Installation froide")
ax.bar(list(x), warm, width, label="Synchronisation chaude")
ax.bar([i + width for i in x], docker, width, label="Build Docker")
ax.set_ylabel("Secondes - mesure locale")
ax.set_title("Coût de la chaîne de livraison du service")
ax.set_xticks(list(x), labels)
ax.legend()
ax.grid(axis="y", alpha=.25)
fig.tight_layout()
fig.savefig("benchmarks.svg", format="svg")
```

Le graphique est valide uniquement si la légende mentionne l'environnement, le matériel, les versions des outils et la date. Les valeurs obtenues ne doivent pas être copiées dans le site comme si elles étaient universelles.

8. Vérifier la CI et la reproductibilité

Reproduisez dans la CI les chemins d'import et les fichiers utilisés localement. L'exemple de service suit une arborescence `src/` : si le paquet n'est pas installé, les tests doivent retrouver ce chemin, comme dans les commandes locales. Le YAML montre l'organisation des jobs ; adaptez-le à l'assemblage effectif de votre dépôt avant d'interpréter un échec de test.

Vérifiez aussi la présence de tous les fichiers attendus par chaque branche de la matrice. Une CI verte atteste les commandes exécutées sur ce commit et ce runner ; elle ne prouve ni une reproductibilité binaire ni la compatibilité avec tous les systèmes. Testez le démarrage du conteneur dans un contrôle distinct plutôt que de déduire sa validité des seuls tests Python.

Ajoutez un contrôle minimal dans GitHub Actions :

```
name: dependency-workflow

on:
  pull_request:
  push:

jobs:
  test:
    strategy:
      matrix:
        workflow: [pip, uv]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Install with pip
        if: matrix.workflow == 'pip'
        run: |
          python -m pip install -r requirements-pip.txt
          python -m pytest -q
      - name: Install with uv
        if: matrix.workflow == 'uv'
        uses: astral-sh/setup-uv@v6
      - name: Test with uv
        if: matrix.workflow == 'uv'
        run: uv run --locked pytest -q
```

Le contrôle échoue si le lockfile est absent, obsolète ou incompatible avec le code. Ajoutez ensuite un job Docker séparé : l'image produite doit répondre à `/health` avant d'être envoyée à un registre.

Sortie obtenue

Sortie / exemple

Pour le service X, dans l'environnement Y, nous retenons l'outil Z parce qu'il respecte les contraintes A et B.

Commande de vérification

La recommandation doit citer les mesures, les contraintes, le compromis accepté et la commande de re-construction. Une préférence non reliée à une mesure vérifiable n'est pas une conclusion de benchmark.

Vérification finale — CI et reproductibilité

9. Décider pour un contexte professionnel

Commencez par les contraintes non négociables : paquets natifs, registre privé, version Python, règles de sécurité et maintenance par l'équipe. Examinez ensuite les mesures des workflows qui respectent ces contraintes. Cette démarche évite de retenir un outil uniquement parce qu'une installation isolée est plus rapide sur la machine du benchmark.

Reliez chaque motif à un fichier ou à une mesure, puis décrivez le coût du changement : migration des déclarations, adaptation de la CI et entretien des exports. Si un workflow n'a pas été testé ou si les mesures sont incomplètes, formulez une hypothèse à vérifier. La recommandation doit permettre à un collègue de reconstruire le service et de comprendre dans quelles conditions le choix pourrait être revu.

Utilisez les mesures et les contraintes du produit :

Contexte	Décision à tester en premier
API Python conteneurisée avec CI fréquente	uv avec pyproject.toml et uv.lock
Dépôt existant déjà standardisé sur pip	pip-tools sans migration précipitée
Librairie Python publiée sur PyPI	Poetry, PDM ou uv selon les standards retenus
Service avec CUDA ou bibliothèques natives	Conda/Mamba, éventuellement combiné avec pip
Migration d'un système ancien	Reproduire l'existant, mesurer, puis migrer par étape

La recommandation finale doit prendre cette forme : « Pour le service X, dans l'environnement Y, nous retenons l'outil Z parce qu'il respecte les contraintes A et B. Nous acceptons le compromis C. La commande de reconstruction est D et le contrôle obligatoire est E. »

Sortie obtenue

Sortie / exemple

```
Décision : outil Z retenu pour le service X
Motifs : contraintes A et B vérifiées
Compromis : C accepté
Reconstruction : commande D
```

Commande de vérification

La recommandation doit renvoyer aux mesures et aux fichiers du benchmark. Si les données contiennent encore « à mesurer », la conclusion reste provisoire.

Exemples de sorties

Le fichier `benchmarks.csv` doit contenir une ligne par outil et par protocole. Le graphique final doit être généré avec :

Terminal · Bash

```
python scripts/plot-benchmarks.py benchmarks.csv
```

La sortie de benchmark est un triptyque :

Sortie / exemple

<code>benchmarks.csv</code>	→ mesures brutes, statut PASS/FAIL
<code>benchmarks.svg</code>	→ médianes froides, chaudes et Docker
<code>benchmark-notes.md</code>	→ machine, Python, versions, caches et commit

Une mesure « à mesurer » n'est pas un résultat exploitable. Tant qu'elle est présente, la recommandation doit rester conditionnelle.

Dépannage et diagnostics

Symptôme	Cause probable	Vérification
temps identiques à chaque outil	cache partagé ou commande non exécutée	isoler les caches et conserver la commande complète
<code>uv run --locked</code> échoue	lockfile absent ou obsolète	exécuter <code>uv lock --check</code> , puis documenter toute régénération
graphique vide	aucune ligne <code>PASS</code> avec des nombres	compléter les colonnes numériques et relancer le script
build Docker non comparable	image ou contexte différents	utiliser le même <code>Dockerfile</code> , le même tag de commit et le même contexte

Limites et prolongements

Un benchmark local ne prouve pas la performance sur tous les runners CI. Répétez l'expérience sur le système de build réel, avec le registre de paquets réellement utilisé et les mêmes règles de cache.

Pour compléter l'étude, mesurez le mode hors ligne, un dépôt multi-packages, des dépendances privées, une construction Docker multi-étapes, l'analyse de vulnérabilités et le temps nécessaire à une mise à jour contrôlée. Le choix d'un gestionnaire n'est pas une préférence esthétique : c'est une décision d'exploitation qui doit rester documentée et réversible.

Références

- [Python Packaging User Guide — recommandations d'outils](#)
- [uv — documentation officielle](#)
- [pip — documentation officielle](#)
- [Surendhan A — UV vs. Pip: A Professional Comparison of Python Package Managers](#)
- [AppSignal — uv vs. Pip in Python: What It Is and How to Switch](#)
- [Real Python — uv vs pip: Managing Python Packages and Dependencies](#)
- [DZone — Python in 2026: uv vs Poetry vs pip](#)