Guide CI/CD – Fintech-test

Ce document explique clairement, pas à pas, ce que fait le pipeline CI/CD de ce repo, comment il fonctionne, comment l'exécuter localement (VS Code/Docker) versus sur GitHub Actions, et ce que signifient chaque étape et chaque test. Objectif: qu'un débutant puisse comprendre et diagnostiquer.

---

1) Vue d'ensemble

- But: vérifier automatiquement que l'application Flask (conteneurisée) se construit, démarre, passe les tests (unitaires et intégration), est scannée côté sécurité (SCA/SAST/scan image/DAST), et résiste à des tests de performance k6: un smoke léger et un test de charge (load) plus réaliste.
- Où:
  - Local (VS Code + Docker): on peut rejouer l'essentiel manuellement pour valider avant de pousser.
  - GitHub Actions: pipeline automatique sur push/pr/workflow_dispatch.
- Sorties (artefacts): rapports JUnit, coverage, rapports sécurité (pip-audit JSON, Bandit JSON, Trivy SARIF), rapports ZAP (HTML/XML), résumé k6 JSON, publiés comme artefacts.

---

2) Déclencheurs

Le workflow `.github/workflows/ci.yml` se lance automatiquement sur:

- `push` (toutes branches)
- `pull_request` (toutes branches)
- `workflow_dispatch` (bouton manuel dans l'onglet Actions)

---

3) Ce que fait le pipeline (résumé en étapes)

1. Checkout du code.
2. Build de l'image Docker de l'API (`bank-api:ci`).
3. Démarrage de l'API avec Docker Compose (expose le port 5000) + healthcheck sur `/health`.
4. Tests unitaires en conteneur (pytest) + génération `junit-unit.xml` et `coverage.xml`.
5. Tests d'intégration en conteneur (pytest) qui appellent `http://localhost:5000/health`.
6. Scans sécurité:
   - SCA (dépendances Python) via pip-audit → `pip-audit.json`.
   - SAST (code Python) via Bandit → `bandit.json`.
   - Scan image Docker via Trivy → `trivy.sarif` (publie aussi vers Code scanning si activé).
   - DAST (OWASP ZAP Baseline) sur `http://localhost:5000` → HTML + XML (non bloquant).
7. Test de performance k6 (smoke) → `k6-summary.json`.
8. Test de performance k6 (load) → `k6-summary-load.json` (scénario réaliste: register/login/dashboard, charge progressive).
9. Upload de tous les rapports comme artefacts.
10. Publication SARIF Trivy vers "Code scanning alerts" (non bloquant si Code scanning désactivé).
11. Teardown (arrêt et suppression des conteneurs/volumes).

---

4) Pré-requis

- Docker installé (local & runners GitHub).
- Pour Code scanning (alertes SARIF dans l'onglet Security): activer "Code security and analysis" > "Code scanning" sur le dépôt GitHub.
- Secrets (optionnels):
  - `SLACK_WEBHOOK_URL` si vous voulez une notification Slack en cas d'échec.

---

5) Détails: chaque étape expliquée

Ci-dessous, les steps clés du job "Build, Test, Scan, DAST, Perf".

5.1 Build image Docker

- Commande: `docker build -t bank-api:ci app_bank`
- Source: `app_bank/Dockerfile`
- But: avoir une image exécutable contenant l'app Flask + dépendances.

5.2 Démarrer l'API (Docker Compose)

- Fichier: `docker/docker-compose.ci.yml`
- Expose `5000:5000`, configure SQLite en volume (`/data`).
- Healthcheck: requête HTTP sur `/health` à l'intérieur du conteneur.
- Le pipeline attend que `/health` réponde 200 avant d'enchaîner.

5.3 Tests unitaires (pytest)

- S'exécutent dans un conteneur éphémère Python.
- Installent pytest + dépendances, puis lancent `tests/unit`.
- Rapports:
  - `reports/junit-unit.xml` (JUnit)
  - `reports/coverage.xml` (cobertura/xml)

5.4 Tests d'intégration (pytest)

- S'exécutent aussi dans un conteneur Python, mais avec `--network host` pour atteindre `http://localhost:5000`.
- Vérifient au minimum le `/health`.
- Rapport: `reports/junit-integration.xml`.

5.5 SCA – pip-audit (dépendances Python)

- Analyse les vulnérabilités des paquets Python à partir de `app_bank/requirements.txt`.
- Sortie: `reports/pip-audit.json` (uploadé comme artefact).
- Note: le pipeline n'échoue pas automatiquement sur findings; vous pouvez mettre en place des règles si besoin.

5.6 SAST – Bandit (code Python)

- Scanne `app_bank` à la recherche de patterns risqués (ex: utilisations dangereuses, crypto faibles, etc.).
- Sortie: `reports/bandit.json`.
- Non bloquant par défaut (Bandit peut retourner des codes d'erreur quand findings existent).

5.7 Scan image Docker – Trivy

- Scanne l'image `bank-api:ci` pour détecter vulnérabilités OS/paquets.
- Sortie: `reports/trivy.sarif` + publication vers "Code scanning alerts".
- Par défaut, l'upload SARIF est non bloquant si Code scanning n'est pas activé (le build continue).

5.8 DAST (OWASP ZAP Baseline)

- Lance un scan "baseline" passif sur `http://localhost:5000`.
- Sorties: HTML/XML/txt dans `zap-reports/`.
- Non bloquant (`continue-on-error: true`) pour éviter de casser le build au début.
- Config: `docker/zap-baseline.conf` permet d'ignorer ou downgrader certaines règles bruyantes.

5.9 k6 – Test de performance (smoke)

- Script: `k6/perf-smoke.js`.
- Charge très légère (1 VU / 10s) qui frappe `/health`.
- Sortie: `k6/k6-summary.json`.
- On peut ajouter des seuils bloquants (p95, erreurs < x%).

5.9 bis k6 – Test de performance (load)

- Script: `k6/perf-load.js`.
- Objectif: simuler un parcours utilisateur plus réaliste avec une charge progressive, sans être trop lourde pour la CI.
- Parcours couvert par VU (par itérations):
    1. GET `/health`
    2. POST `/register` (uniquement lors de la première itération de chaque VU pour limiter les écritures)
    3. POST `/login`
    4. GET `/dashboard`
- Stages par défaut (CI-friendly): montée progressive à ~40 utilisateurs virtuels (VU), maintien, puis descente.
- Seuils par défaut (adaptés CI): p95 < ~800ms; erreurs HTTP < 10%; erreurs métier < 20%.
- Détails techniques:
    - Les formulaires sont envoyés en form-urlencoded.
    - Les cookies de session sont conservés automatiquement par VU.
    - Avec SQLite (CI), l'appli active WAL + busy_timeout pour réduire les locks sous charge.
- Sortie: `k6/k6-summary-load.json`.

5.10 Artefacts & alertes

- Artefacts uploadés:
    - Tests: `reports/junit-unit.xml`, `reports/junit-integration.xml`, `reports/coverage.xml`.

    - Sécurité: `reports/pip-audit.json`, `reports/bandit.json`, `reports/trivy.sarif`.
    - DAST: `zap-reports/` (HTML/XML/txt).
    - Perf: `k6/k6-summary.json`.
    - Perf (load): `k6/k6-summary-load.json`.
- SARIF Trivy publié vers Code scanning (non bloquant si désactivé).

5.11 Notifications Slack (optionnel)

- Si le job échoue et que `SLACK_WEBHOOK_URL` est défini, envoie un message succinct avec le lien du run.

5.12 Nettoyage

- `docker compose down -v` pour arrêter/retirer les conteneurs et volumes du job.

---

6) Différences: exécution locale vs GitHub Actions

Aspect	Local (VS Code + Docker)	GitHub Actions
Déclencheur	Manuellement, via commandes Docker/pytest	Automatique sur push/PR/dispatch
Environnement	Votre machine (Windows/macOS/Linux)	Runner Ubuntu propre, jetable
Réseau	`--network host` dépend de l'OS (OK Linux, comportement spécifique Windows/Mac)	Runner Ubuntu: `--network host` fonctionne pour joindre `localhost:5000`
Artefacts	Fichiers générés localement dans `reports/`, `k6/`, `zap-reports/`	Uploadés en artefacts du run (onglet Actions > run > Artifacts)
Code scanning (SARIF)	N/A localement	Si activé, SARIF affiché dans Security > Code scanning alerts
Isolation	Vous gérez les conteneurs	Tout est jetable par run

En pratique, on développe/teste localement, puis on push pour une validation reproductible sur Actions.

---

7) Comment exécuter localement (Windows PowerShell)

Depuis la racine du repo:

```
# Build image
docker build -t bank-api:ci app_bank

# Démarrer l'API (port 5000)
docker compose -f docker\docker-compose.ci.yml up -d

# Vérifier le health
curl http://localhost:5000/health
```

```
# Tests unitaires (JUnit + coverage)
docker run --rm -v "${PWD}:/work" -w /work python:3.11-slim bash -lc "\
  pip install --no-cache-dir -r app_bank/requirements.txt pytest pytest-cov
requests junit-xml && \
  pytest -q tests/unit --junitxml reports/junit-unit.xml --cov=app_bank --cov-
report xml:reports/coverage.xml"

# Tests d'intégration
docker run --rm --network host -v "${PWD}:/work" -w /work python:3.11-slim bash -
lc "\
  pip install --no-cache-dir requests pytest junit-xml && \
  pytest -q tests/integration --junitxml reports/junit-integration.xml"

# k6 smoke
docker run --rm --network host -v "${PWD}\k6:/scripts" grafana/k6:0.51.0 run \
  --vus 1 --duration 10s --summary-export=/scripts/k6-summary.json /scripts/perf-
smoke.js

# (Option) ZAP baseline local (si image accessible)
# docker run --rm --network host -v "${PWD}:/zap/wrk" owasp/zap2docker-stable \
#   zap-baseline.py -t http://localhost:5000 -r zap-baseline-report.html -x zap-
baseline-report.xml -w zap-warn.txt -c zap-baseline.conf -m 5

# Arrêt & nettoyage
docker compose -f docker\docker-compose.ci.yml down -v
```

---

8) Comment lancer sur GitHub Actions

- Sur un push: poussez vos commits sur n'importe quelle branche.
- Sur une Pull Request: créez une PR (le workflow se lance).
- Manuel: allez dans l'onglet Actions > CI > Run workflow.

Résultats à voir:

- Onglet Actions: logs des steps + artefacts (fichiers téléchargeables).
- Onglet Security > Code scanning alerts: les vulnérabilités remontées par SARIF (Trivy), si Code scanning est activé.

---

9) Dépannage (FAQ)

- "Publish SARIF to code scanning failed / Code scanning is not enabled": Activez Code scanning dans Settings > Code security and analysis ou laissez le step non-bloquant (déjà le cas).
- "Impossible d'atteindre http://localhost:5000 en intégration":
  - Vérifiez que le service `bank_api` démarre (logs compose) et que `/health` répond 200.
  - Ports déjà occupés sur la machine locale ? Changez le mapping dans `docker-compose.ci.yml`.
- "ZAP image introuvable": Sur certaines machines, `owasp/zap2docker-stable` nécessite un login/pull. Sur Actions, c'est pris en charge automatiquement par Docker Hub (limites de rate possibles).
- "Trivy ne trouve pas l'image en local": il faut que l'image `bank-api:ci` existe sur la machine, ou montez le socket Docker si vous lancez Trivy en conteneur manuellement.

- “pip-audit en SARIF ?”: on garde JSON ici; pour des alertes Code scanning sur les dépendances, envisagez Trivy FS/SBOM + SARIF ou CodeQL supply chain, selon vos politiques.

---

10) Prochaines évolutions (recommandations)

- Déploiement sur tag (ex: `v*`) vers un registre + staging/prod (environnements GitHub, secrets par env).
- Seuils k6 bloquants (p95 < X ms, erreurs < 1%) et export vers InfluxDB/Grafana.
- Qualité de code: ruff/flake8 + mypy + black, avec checks bloquants.
- Ajout CodeQL (langage Python) pour SAST avancé.
- Cache pip et cache Docker layers pour accélérer les CI.
- SBOM (syft) + policy-as-code (Conftest/OPA) pour gates supply-chain.

---

11) Où sont les fichiers clés ?

- Workflow CI: `.github/workflows/ci.yml`
- Compose CI: `docker/docker-compose.ci.yml`
- Tests unitaires: `tests/unit/`
- Tests d’intégration: `tests/integration/`
- k6: `k6/perf-smoke.js`
- ZAP config: `docker/zap-baseline.conf`
- Application Flask: `app_bank/`

---

12) Couverture: débloquer `.coverage` et générer un rapport HTML (Windows)

Contexte: le fichier `.coverage` produit par pytest-cov en CI/containers peut contenir des chemins de fichiers différents de votre machine locale (ex: `C:\\work\\app_bank` ou `/work/app_bank`). Sans mappage, `coverage` ne retrouve pas les sources et affiche des avertissements du type “Couldn't parse ... No source for code” ou “No data to report”.

12.1 Correction mise en place

Ajout d’un fichier `.coveragerc` à la racine du repo pour résoudre les chemins et fixer la destination HTML:

- Fichier: `.coveragerc`
- Contenu clé:
  - `[run]` → `source = app_bank`
  - `[paths]` → `source = app_bank`, `*/work/app_bank`, `*/workspace/app_bank`, `C:\\work\\app_bank`
  - `[report]` → `show_missing = True`, `ignore_errors = True`
  - `[html]` → `directory = reports/htmlcov`

Objectif: dire à `coverage` que les fichiers référencés sous ces différents préfixes correspondent à votre dossier local `app_bank/`.

12.2 Générer des rapports lisibles depuis `.coverage`

Si `coverage` n'est pas installé pour le lanceur Python Windows:

```
py -m ensurepip --upgrade
py -m pip install coverage
```

Sur Windows PowerShell, si `coverage.exe` n'est pas dans le PATH, utilisez le chemin complet (remplacez si besoin par votre chemin local):

```
# Rapport HTML
& 'C:\\Users\\harry\\Downloads\\Fintech-test-main\\Fintech-test-
main\\Scripts\\coverage.exe' html -d reports\\htmlcov -i

# Rapport texte lisible
& 'C:\\Users\\harry\\Downloads\\Fintech-test-main\\Fintech-test-
main\\Scripts\\coverage.exe' report -m | Out-File -Encoding utf8
reports\\coverage.txt

# Rapport JSON
& 'C:\\Users\\harry\\Downloads\\Fintech-test-main\\Fintech-test-
main\\Scripts\\coverage.exe' json -o reports\\coverage.json
```

Ensuite, ouvrez le fichier `reports/htmlcov/index.html` dans votre navigateur (clic droit > Reveal in File Explorer > double-clic).

12.3 Notes utiles

- Si "No data to report" persiste, vérifiez que `.coveragerc` est bien à la racine du repo (même dossier que `.coverage`) et que la section `[paths]` couvre les préfixes vus dans les avertissements (`C:\\work\\app_bank`, `/work/app_bank`, etc.).
- Vous pouvez aussi régénérer `.coverage` en relançant localement `pytest --cov=app_bank --cov-report xml:reports/coverage.xml`, ce qui créera un `.coverage` adapté à votre machine.
- Les patterns `*/work/app_bank` (Linux runners) et `C:\\work\\app_bank` (Windows) couvrent la plupart des cas CI/containers.

---

13) Comparaison rapide: k6 smoke vs k6 load

- Portée:
  - Smoke: vérifie simplement que `/health` répond (disponibilité basique).
  - Load: rejoue un mini-parcours (register/login/dashboard) avec sessions et écritures.
- Intensité:
  - Smoke: 1 VU sur ~10s.
  - Load: montée jusqu'à ~40 VU avec paliers (CI-friendly), durée plus longue.
- Risques/effets de bord:
  - Smoke: aucun effet d'écriture, très sûr.

    - Load: écritures en base (création d'utilisateurs). Le script limite ces écritures (inscription une seule fois par VU) pour ménager SQLite.
- Seuils/échecs:
    - Smoke: seuils généreux, focus sur erreurs réseau/HTTP.
    - Load: seuils plus exigeants (p95, erreurs HTTP et métier) mais adaptés à l'environnement CI.
- Sorties:
    - Smoke: `k6-summary.json`.
    - Load: `k6-summary-load.json`.