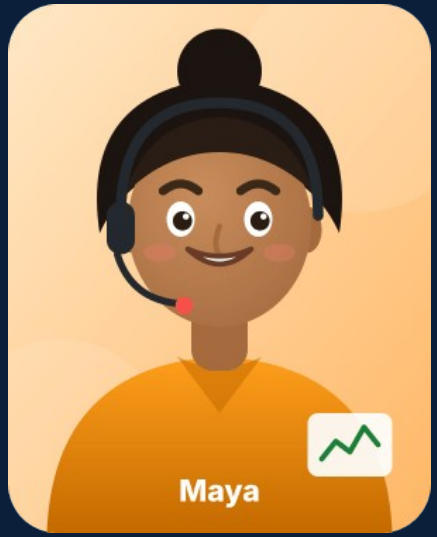
HEF-110 · MISSION 10 / 10

Tests dans CI

Les tests n'ont de valeur que s'ils tournent à chaque changement, vite, et si leur verdict bloque ce qui est cassé.



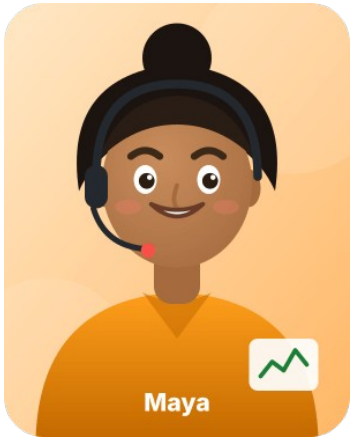


Badge : Gardien du pipeline

110 min · intermédiaire

L'histoire

La suite de HashBox est belle… mais elle ne tourne que sur l'ordinateur de Kodi, quand il y pense. Un jeudi, quelqu'un fusionne un changement sans lancer les tests : le dimanche, l'inscription est cassée. Personne ne s'en rend compte avant lundi.



« Un test qu'on peut oublier de lancer sera oublié, dit Sam. Mets-le dans la chaîne : chaque proposition de modification déclenche automatiquement les vérifications, et rien de rouge ne passe. C'est l'intégration continue. »

Dans cette dernière mission, tu bâtis un pipeline de tests complet : étapes, retour rapide, rapports, seuils de qualité, tests en parallèle, sélection des tests utiles, et gestion des instables.

Ta mission

À la fin, je serai capable de…

1. décrire le rôle de l'intégration continue et des portes de qualité
2. organiser un pipeline en étapes à retour rapide (échouer vite)
3. produire des rapports exploitables (JUnit, couverture) et fixer des seuils
4. répartir les tests en parallèle (sharding) et sélectionner ceux qui sont concernés par un changement
5. maintenir la santé du pipeline (durée, instabilité, propriété)

Défi d'échauffement

Défi d'échauffement

Dans une usine, un contrôle qualité en bout de chaîne trouve un défaut… produit trois jours plus tôt, sur 4 000 pièces. Compare avec un contrôle à chaque poste. Combien de pièces sont perdues dans chaque cas ? Quelle est la logique du « retour rapide » ?

Le concept

L'intégration continue (CI)

La **CI** exécute automatiquement, à **chaque changement** proposé, une série de vérifications : compilation, style, tests, sécurité. Le résultat bloque ou autorise la fusion.

Pratique	Bénéfice
fusionner souvent, en petits changements	conflits et bugs plus petits
vérifications automatiques à chaque changement	pas d'oubli, même niveau d'exigence pour tous
rouge = priorité	le pipeline cassé se répare tout de suite
résultat visible de tous	responsabilité partagée

Le concept

Un pipeline de tests : des étapes ordonnées



Description du schéma : cinq étapes en chaîne, des plus rapides aux plus lentes : style et analyse statique, tests unitaires, tests d'intégration, tests de contrat et d'API, E2E sur les parcours critiques, puis une porte de qualité finale.

Principe : échouer vite. On place en premier ce qui est **rapide et décisif**. Si le style ou les tests unitaires échouent, inutile de lancer les E2E de dix minutes.

Les portes de qualité (*quality gates*)

Des seuils **automatiques** qui bloquent la fusion :

Le concept

Les portes de qualité (*quality gates*) (suite)

Porte	Exemple
Tous les tests passent	aucune exception
Couverture minimale	≥ 80 % de lignes **sur le code modifié**
Pas de nouvelle faille	analyse de sécurité (HEF-100)
Pas de dépendance vulnérable	audit des dépendances
Taille et revue	une relecture humaine obligatoire

Une porte doit être **stable** et **comprise** : une porte qui bloque pour de mauvaises raisons est contournée.

Rapports exploitables

- **JUnit XML** : format standard lu par tous les outils de CI (nom de chaque test, durée, échec, message).

Le concept

Rapports exploitables (suite)

- **Couverture** : par fichier, et surtout **sur le code modifié**.
- **Artefacts** : captures d'écran, journaux, HTML des pages en échec (mission 6).
- **Tendances** : durée de la suite, taux d'instabilité, nombre de tests.

Rester rapide

Objectif courant : un retour **en moins de 10 minutes** sur un changement.

Technique	Principe
Parallélisme	répartir les tests sur plusieurs machines (*sharding*) en équilibrant les durées
Cache	réutiliser dépendances et résultats de build
Sélection des tests	ne lancer que les tests concernés par les fichiers modifiés (analyse d'impact) ; suite complète en parallèle ou la nuit

Le concept

Rester rapide (suite)

Technique	Principe
Pyramide respectée	beaucoup d'unitaires rapides, peu d'E2E
Chasse aux tests lents	traiter les 10 plus lents

Répartition équilibrée (LPT) : trier les tests par durée décroissante, et affecter chacun à la machine la moins chargée. Simple et près de l'optimum.

Analyse d'impact

Un graphe de dépendances indique quel test dépend de quel module. Si `tarifs.py` change, on lance les tests de `tarifs` et de **tout ce qui en dépend** (transitivement). En cas de doute (fichier de configuration global), on lance **tout**. Sécurité d'abord : mieux vaut un test en trop qu'un bug raté.

Le concept

Un exemple de configuration

```yaml
name: tests
on: [pull_request, push]
jobs:
  verification:
    runs-on: ubuntu-latest
    timeout-minutes: 15
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: "3.12", cache: pip }
      - run: python -m compileall -q src            # style / syntaxe
      - run: python -m unittest discover -s tests -v   # unitaires + intégration
      - run: python ci/porte_couverture.py --minimum 80
      - uses: actions/upload-artifact@v4
        if: always()
        with: { name: rapports, path: rapports/ }
```

Le concept

Santé du pipeline

Indicateur	Cible
Durée	< 10 min
Taux d'instabilité	proche de 0 % (mission 9)
Temps de réparation d'un rouge	quelques heures au plus
Propriétaire de chaque test	identifié
Branches protégées	fusion impossible si rouge

🧠 ***Astuce*** *: un pipeline est un* ***produit*** *avec des utilisateurs (l'équipe). Mesure leur frustration : durée, relances, rouges inexpliqués.*

Exemple pas à pas

Problème : construire un mini pipeline de CI en Python : étapes, échec rapide, rapport JUnit XML, porte de couverture, sur un vrai petit projet créé dans un dossier temporaire ; puis simuler une régression et un manque de couverture.

```
import ast, io, sys, tempfile, time, unittest
import xml.etree.ElementTree as ET
from pathlib import Path

sys.dont_write_bytecode = True          # pas de .pyc périmé quand on réécrit un f
   ichier de même taille dans la même seconde

CALC = '''
def somme(a, b):
    return a + b

def moyenne(valeurs):
    if not valeurs:
        raise ValueError("vide")
```

Exemple pas à pas (suite)

```
    return sum(valeurs) / len(valeurs)
'''

TESTS = '''
import unittest
from calc import somme, moyenne

class TestCalc(unittest.TestCase):
    def test_somme(self):
        self.assertEqual(somme(2, 3), 5)
    def test_moyenne(self):
        self.assertEqual(moyenne([2, 4]), 3)
    def test_moyenne_vide(self):
        with self.assertRaises(ValueError):
            moyenne([])
'''

# ---------------- étapes du pipeline ----------------
def etape_syntaxe(projet):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    for fichier in Path(projet).glob("*.py"):
        compile(fichier.read_text(), str(fichier), "exec")                # Synta
   xError = échec
    return True, "syntaxe valide"

class Collecteur(unittest.TextTestResult):
    def __init__(self, *a, **k):
        super().__init__(*a, **k)
        self.lignes, self._debut = [], {}
    def startTest(self, test):
        self._debut[test] = time.perf_counter(); super().startTest(test)
    def _noter(self, test, statut, detail=""):
        self.lignes.append((test.id(), time.perf_counter() - self._debut[test], st
   atut, detail))
    def addSuccess(self, test): super().addSuccess(test); self._noter(test, "ok")
    def addFailure(self, test, err): super().addFailure(test, err); self._noter(te
   st, "echec", self._exc_info_to_string(err, test))
    def addError(self, test, err): super().addError(test, err); self._noter(test,
   "erreur", self._exc_info_to_string(err, test))
```

Exemple pas à pas (suite)

```
def oublier_modules():                                                  # c
   haque étape repart d'imports neufs
    for nom in ("calc", "test_calc"):
        sys.modules.pop(nom, None)

def etape_tests(projet, rapport):
    sys.path.insert(0, projet)
    try:
        oublier_modules()
        suite = unittest.defaultTestLoader.discover(projet, pattern="test_*.py")
        resultat = unittest.TextTestRunner(stream=io.StringIO(), resultclass=Colle
   cteur).run(suite)
    finally:
        sys.path.remove(projet)
    racine = ET.Element("testsuite", name="projet", tests=str(resultat.testsRun),
                        failures=str(len(resultat.failures) + len(resultat.errors)
   ))
    for nom, duree, statut, detail in resultat.lignes:
```

Exemple pas à pas (suite)

```
        cas = ET.SubElement(racine, "testcase", name=nom, time=f"{duree:.4f}")
        if statut != "ok":
            ET.SubElement(cas, "failure", message=detail.strip().splitlines()[-1])
   .text = detail
    ET.ElementTree(racine).write(rapport, encoding="utf-8", xml_declaration=True)
    return resultat.wasSuccessful(), f"{resultat.testsRun} tests, {len(resultat.fa
   ilures) + len(resultat.errors)} en échec"

def lignes_executables(source):
    return {n.lineno for n in ast.walk(ast.parse(source)) if isinstance(n, ast.stm
   t)}

def etape_couverture(projet, minimum):
    chemin = str(Path(projet, "calc.py"))
    source = Path(chemin).read_text()
    vues = set()
    def traceur(frame, ev, arg):
        if frame.f_code.co_filename == chemin:
            if ev == "line": vues.add(frame.f_lineno)
```

Exemple pas à pas (suite)

```
            return traceur
    sys.path.insert(0, projet)
    oublier_modules()
    sys.settrace(traceur)
    try:
        suite = unittest.defaultTestLoader.discover(projet, pattern="test_*.py")
        unittest.TextTestRunner(stream=io.StringIO()).run(suite)
    finally:
        sys.settrace(None); sys.path.remove(projet)
    cibles = lignes_executables(source)
    taux = len(vues & cibles) / len(cibles) * 100
    return taux >= minimum, f"couverture {taux:.0f} % (minimum {minimum} %)"

def pipeline(projet, minimum=80):
    rapport = str(Path(projet, "junit.xml"))
    etapes = [("syntaxe", lambda: etape_syntaxe(projet)),
              ("tests", lambda: etape_tests(projet, rapport)),
              ("couverture", lambda: etape_couverture(projet, minimum))]
    journal = []
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    for nom, fonction in etapes:
        debut = time.perf_counter()
        ok, message = fonction()
        journal.append((nom, "✅" if ok else "❌", message, time.perf_counter() - de
   but))
        if not ok:
            journal += [(restante, "⏭ ", "ignorée (échec précédent)", 0.0) for re
   stante, _ in etapes[len(journal):]]
            return 1, journal, rapport                                  # ÉC
   HOUER VITE : code de sortie 1
    return 0, journal, rapport

def afficher(titre, code, journal):
    print(f"--- {titre} → code de sortie {code}")
    for nom, icone, message, duree in journal:
        print(f"  {icone} {nom:10s} {message}")

with tempfile.TemporaryDirectory() as d:
    Path(d, "calc.py").write_text(CALC)
```

Exemple pas à pas (suite)

```
    Path(d, "test_calc.py").write_text(TESTS)

    code, journal, rapport = pipeline(d)
    afficher("1. changement sain", code, journal)
    assert code == 0 and all(e[1] == "✅" for e in journal)

    Path(d, "calc.py").write_text(CALC.replace("a + b", "a - b"))
    # régression introduite
    code, journal, rapport = pipeline(d)
    afficher("2. régression", code, journal)
    assert code == 1 and journal[1][1] == "❌" and journal[2][1] == "⏭ "
    xml = ET.parse(rapport).getroot()
    echecs = [c.get("name") for c in xml.iter("testcase") if c.find("failure") is
   not None]
    print("  tests en échec dans le rapport JUnit :", echecs)
    assert echecs == ["test_calc.TestCalc.test_somme"]

    Path(d, "calc.py").write_text(CALC + "\ndef mediane(valeurs):\n    v = sorted(
   valeurs)\n    m = len(v) // 2\n    return v[m] if len(v) % 2 else (v[m - 1] + v
```

Exemple pas à pas (suite)

```
    [m]) / 2\n")
     code, journal, rapport = pipeline(d)
     afficher("3. nouvelle fonction sans test", code, journal)
     assert code == 1 and journal[2][1] == "❌"
```

Lecture :

1. Les trois étapes s'enchaînent **dans l'ordre du plus rapide au plus lent** ; à la première panne, les suivantes sont **ignorées** (*échouer vite*) et le **code de sortie** vaut 1 : c'est ce code que lit un outil de CI pour bloquer la fusion.
2. La régression (`a + b` devenu `a - b`) est attrapée par `test_somme` et nommée dans le **rapport JUnit** : le format que tous les outils savent afficher.
3. La **porte de couverture** bloque une nouvelle fonction sans test, même si tous les tests existants passent.
4. Tout se passe dans un dossier temporaire : le pipeline est **reproductible** et sans effet de bord.

À toi, avec aide

Écris `repartir(tests, machines)` qui répartit des tests (nom → durée en secondes) sur `machines` machines parallèles en minimisant la durée totale (algorithme LPT : trier par durée décroissante, affecter à la machine la moins chargée). Teste avec ces durées :

```
{"e2e_a": 120, "e2e_b": 90, "integ_a": 45, "integ_b": 40, "integ_c": 35, "unit_a": 5, "unit_b": 4, "unit_c": 3, "unit_d": 2}
```

Compare la durée de la machine la plus chargée avec la durée séquentielle, pour 1, 2, 3 et 4 machines.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise une liste de charges et `min(range(machines), key=...)`.

Indice 2

La durée totale en parallèle = charge maximale.

Indice 3

La somme des durées vaut 344 s.

Réponse

```
def repartir(tests, machines):
    charges = [0] * machines
    affectation = [[] for _ in range(machines)]
    for nom, duree in sorted(tests.items(), key=lambda kv: -kv[1]):
        m = min(range(machines), key=lambda i: charges[i])
        affectation[m].append(nom)
        charges[m] += duree
    return affectation, charges

TESTS = {"e2e_a": 120, "e2e_b": 90, "integ_a": 45, "integ_b": 40, "integ_c": 35, "
   unit_a": 5, "unit_b": 4, "unit_c": 3, "unit_d": 2}
sequentiel = sum(TESTS.values())
for k in (1, 2, 3, 4):
    _, charges = repartir(TESTS, k)
    print(f"{k} machine(s) : {max(charges):3d} s (gain x{sequentiel / max(charges)
   :.2f})")
assert sequentiel == 344
assert max(repartir(TESTS, 1)[1]) == 344 and max(repartir(TESTS, 2)[1]) <= 175 and
    max(repartir(TESTS, 4)[1]) == 120
```

À toi, avec aide · réponse

Réponse (suite)

Avec 4 machines, la durée est plafonnée par le **plus long test** (120 s) : ajouter des machines n'aide plus. Pour aller plus vite, il faudrait **découper** ce test, pas ajouter du matériel.

À toi, en solo

Sélection des tests par analyse d'impact. Écris un outil `tests_a_lancer(changes, dependances, tests_par_module)` :

1. `dependances` : module → modules qu'il importe (ex. `service` importe `depot` et `tarifs`) ;
2. calcule les modules **impactés** (les modules modifiés et tous ceux qui en dépendent, transitivement) ;
3. renvoie les tests associés aux modules impactés ;
4. si un fichier « global » est modifié (`config`, `requirements`, `setup`), renvoie **tous** les tests ;
5. gère les cycles de dépendances sans boucle infinie ;
6. teste avec un graphe de 8 modules et 5 scénarios de changement, et calcule le gain moyen (tests évités).

À toi, en solo · réponse

Pistes de correction

```
GLOBAUX = {"config", "requirements", "setup"}

def modules_impactes(modifies, dependances):
    inverse = {}
    for module, imports in dependances.items():
        for i in imports:
            inverse.setdefault(i, set()).add(module)
    vus, pile = set(), list(modifies)
    while pile:
        m = pile.pop()
        if m in vus:
            continue
        vus.add(m)
        pile.extend(inverse.get(m, ()))                    # tous ceux qui impo
   rtent m
    return vus

def tests_a_lancer(changes, dependances, tests_par_module):
    if set(changes) & GLOBAUX:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        return sorted(t for ts in tests_par_module.values() for t in ts)
    impactes = modules_impactes(changes, dependances)
    return sorted(t for m in impactes for t in tests_par_module.get(m, ()))

DEPENDANCES = {"depot": [], "tarifs": [], "mail": [], "service": ["depot", "tarifs
   "], "rappel": ["depot", "mail"],
              "api": ["service", "rappel"], "cli": ["service"], "ciclo_a": ["cicl
   o_b"], "ciclo_b": ["ciclo_a"]}
TESTS = {m: [f"test_{m}_{i}" for i in range(1, 4)] for m in ("depot", "tarifs", "m
   ail", "service", "rappel", "api", "cli", "ciclo_a", "ciclo_b")}
total = sum(len(t) for t in TESTS.values())

scenarios = [["tarifs"], ["mail"], ["depot"], ["api"], ["ciclo_a"], ["config"]]
evites = []
for s in scenarios:
    choisis = tests_a_lancer(s, DEPENDANCES, TESTS)
    evites.append(total - len(choisis))
    print(f"{s!s:14s} → {len(choisis):2d}/{total} tests")
assert len(tests_a_lancer(["tarifs"], DEPENDANCES, TESTS)) == 3 * 4          # tar
```

Pistes de correction (suite)

```
   ifs, service, api, cli
assert len(tests_a_lancer(["config"], DEPENDANCES, TESTS)) == total
assert {t.split("_")[1] for t in tests_a_lancer(["ciclo_a"], DEPENDANCES, TESTS)}
   == {"ciclo"}   # le cycle ne boucle pas
print(f"gain moyen : {sum(evites) / len(evites):.1f} tests évités sur {total}")
```

Le parcours en profondeur avec un ensemble de modules **déjà vus** évite la boucle infinie sur les cycles. En cas de doute (fichier global), on lance tout : la **sécurité prime** sur la vitesse.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tests qu'on peut oublier de lancer	Régressions en production	Exécution automatique à chaque changement, fusion bloquée si rouge
Pipeline lent (30 min+)	On contourne ou on empile les changements	Échouer vite, paralléliser, sélectionner, viser < 10 min
Porte de qualité arbitraire ou instable	Contournée, ignorée	Seuils compris, stables, sur le code modifié
« Rouge » toléré	Les vrais bugs passent inaperçus	Rouge = priorité ; instabilités en quarantaine tracée

Mini-boss

Mini-boss

Ton pipeline de tests complet. Construis `mini_ci.py` qui prend un projet Python (dossier) et une configuration (dictionnaire) et exécute :

1. étapes configurables (syntaxe, unitaires, intégration, couverture) avec **échec rapide** et temps par étape ;
2. **rapport JUnit XML** et **résumé Markdown** (tests, durées, les 5 plus lents, couverture par fichier) ;
3. **porte de couverture** sur le code *modifié* (en utilisant la liste de fichiers/lignes changés fournie) ;
4. **sélection des tests** par analyse d'impact, avec repli sur la suite complète pour les fichiers globaux ;
5. **répartition** des tests sur N machines simulées (LPT) à partir des durées de la dernière exécution ;

Mini-boss

Mini-boss (suite)

6. **quarantaine** : un fichier `quarantaine.json` liste des tests instables, exécutés mais non bloquants ;
7. 10 tests de `mini_ci.py` lui-même, y compris régression, couverture insuffisante, test en quarantaine qui échoue sans bloquer.

Conclus par 10 lignes : parmi les mesures du pipeline, laquelle surveillerais-tu chaque semaine, et pourquoi ?

Mini-boss · réponse

Pistes de correction

Un bon projet : une fonction par étape et un orchestrateur qui retourne un code de sortie ; un `Collecteur` de résultats (voir exemple) pour le JUnit et les durées ; la couverture « sur le code modifié » en croisant lignes tracées et lignes changées ; l'analyse d'impact de la mission ; LPT pour la répartition ; un test en quarantaine détecté par son identifiant et compté séparément (`non bloquant`). L'indicateur hebdomadaire attendu : plutôt la **durée de retour** (l'indicateur d'expérience développeur) ou le **taux d'instabilité** (indicateur de confiance) — avec une justification liant la métrique à un comportement d'équipe (contournement, relance, abandon).

Quiz éclair

Quiz éclair

1. Dans un pipeline, on place en premier :

A. les étapes rapides et décisives
B. les étapes les plus longues
C. les tests E2E

2. Une porte de qualité est :

A. une règle facultative
B. un seuil automatique qui bloque la fusion s'il n'est pas respecté
C. un outil de design

3. Pour accélérer une suite lente, on peut :

A. supprimer les assertions

Quiz éclair

Quiz éclair (suite)

B. désactiver les tests qui échouent
C. paralléliser, mettre en cache et ne lancer que les tests concernés

4. 🔁 *Révision mission 09* : Un test instable est un test qui :

A. passe ou échoue sur le même code sans modification
B. est très lent
C. n'a pas d'assertion

5. 🔁 *Révision mission 09* : Un mutant survivant indique :

A. que le code est parfait
B. que la suite n'a pas détecté une altération du code
C. que le mutant est mort

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

Je réfléchis

1. Combien de temps s'écoule aujourd'hui entre ton changement et le verdict des tests ? Que ferais-tu si cela doublait ?
2. Quelle règle d'équipe garantirait que le rouge soit réparé tout de suite ?

À retenir

1. CI : vérifications automatiques à chaque changement ; rouge = priorité.
2. Étapes ordonnées du plus rapide au plus lent : échouer vite.
3. Rapports (JUnit, couverture) et portes de qualité stables.
4. Rapidité : parallélisme (LPT), cache, sélection par impact.
5. Santé : durée, instabilité, quarantaine tracée, propriétaires.

Félicitations

Tu as terminé **HEF-110 — Tests et qualité**. Tu sais spécifier un comportement par des exemples, écrire des tests unitaires, d'intégration, d'API et de bout en bout, utiliser des doubles avec discernement, avancer en TDD, traquer l'instabilité, mesurer la qualité d'une suite par la mutation et la faire tourner dans un pipeline.
→ Prochain module : **HEF-120 — DevOps et livraison**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Gardien du pipeline

Fin du module HEF-110. Bravo !

