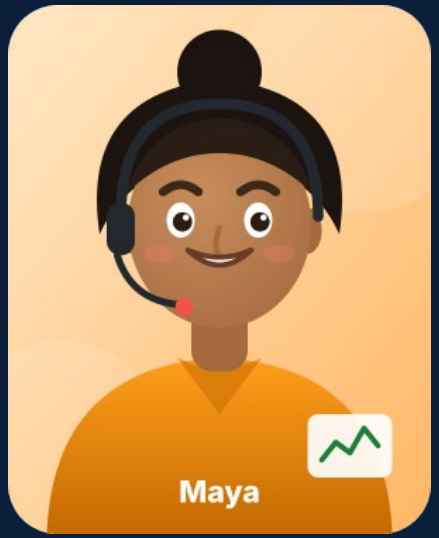
HEF-130 · MISSION 10 / 10

Optimisation et validation

Optimiser, ce n'est pas écrire du code astucieux : c'est suivre une méthode — mesurer, corriger un seul point, vérifier le résultat, protéger le gain.



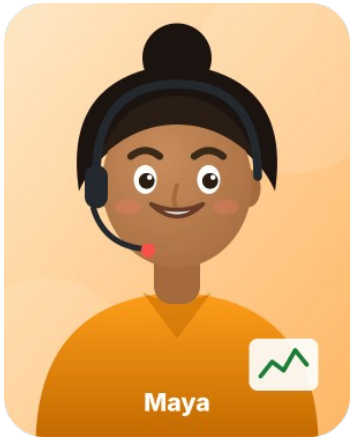


Badge : Optimiseur méthodique

115 min · avancé

L'histoire

Après quatre mois, HashBox a vu défiler une dizaine d'« optimisations ». Certaines ont réellement aidé. D'autres ont rendu le code illisible sans gain mesurable. Une a introduit un bug subtil : un cache qui renvoyait de vieux prix. Une autre a été défaite trois semaines plus tard, quand quelqu'un a modifié le code sans savoir que la performance en dépendait, et que personne n'a rien détecté.



> *« Une optimisation sans mesure est une opinion, dit Sam. Une optimisation sans test de correction est un pari. Une optimisation sans garde-fou est un souvenir. Il faut les trois : mesurer, vérifier, protéger. »*

Dans cette dernière mission, tu appliques toute la méthode du module : baseline, hypothèse, changement unique, mesure, vérification d'équivalence, garde-fou de non-régression et rapport.

Ta mission

À la fin, je serai capable de…

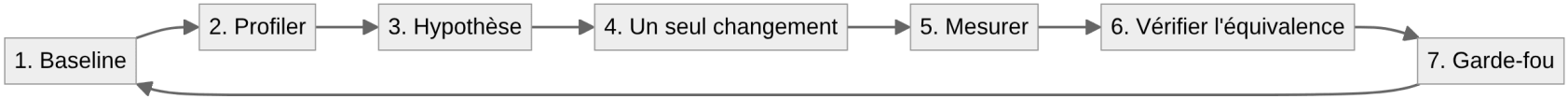
1. suivre le protocole baseline → hypothèse → changement → mesure → vérification
2. décider si une optimisation vaut son coût (gain, complexité, maintenance)
3. garantir l'équivalence fonctionnelle avant et après optimisation
4. écrire des garde-fous déterministes (comptage d'opérations, tests de complexité) et relatifs
5. rédiger un rapport de performance reproductible (protocole, données brutes, résultat)

Défi d'échauffement

Un garagiste dit : « Votre voiture consomme trop, j'ai remplacé le filtre à air, les bougies, le carburateur et réglé les pneus. » Elle consomme maintenant un peu moins. Peut-il dire laquelle de ces actions a servi ? Qu'aurait-il fallu faire pour le savoir, et pour que le gain ne disparaisse pas au prochain entretien ?

Le concept

Le protocole d'optimisation



Description du schéma : sept étapes en boucle : établir une baseline, profiler, formuler une hypothèse, appliquer un seul changement, mesurer, vérifier que le résultat est équivalent, puis poser un garde-fou avant de recommencer sur le point chaud suivant.

Étape	Exigence
Baseline	charge représentative, protocole de la mission 2, données brutes conservées
Profiler	désigner le goulot par des mesures (mission 3)

Le concept

Le protocole d'optimisation (suite)

Étape	Exigence
Hypothèse	« si je fais X, Y devrait baisser de Z % parce que… » — **falsifiable**
Un changement	sinon on ne sait pas ce qui a aidé
Mesurer	mêmes conditions, intervalle d'incertitude
Vérifier	résultats **identiques** (tests, comparaison de sorties, propriétés)
Garde-fou	un test qui empêchera le retour en arrière

Faut-il optimiser ?

Donald Knuth : *« Nous devrions oublier les petites optimisations, disons 97 % du temps : l'optimisation prématurée est la racine de tous les maux. Pourtant, nous ne devrions pas laisser passer nos opportunités dans ces 3 % critiques. »*

Le concept

Faut-il optimiser ? (suite)

Calcule la **valeur** d'une optimisation :

```
gain = (temps économisé par exécution) × (nombre d'exécutions) × (valeur du temps/
   coût)
coût = temps de développement + complexité ajoutée × maintenance + risque de bug
```

Optimise si le gain l'emporte **nettement**, et si la performance est **un besoin réel** (budget dépassé, coût d'infrastructure, plaintes d'utilisateurs). Sinon, préfère la clarté.

Ordre des remèdes (du plus rentable au moins rentable)

1. **Ne pas faire** le travail (supprimer, éviter les appels redondants) ;
2. **Faire moins** (filtrer tôt, paginer, ne lire que les colonnes utiles) ;
3. **Mieux organiser** : algorithme et structure de données (HEF-020, HEF-030) ;
4. **Faire une fois** : cache, précalcul, regroupement (missions 6 à 8) ;

Le concept

Ordre des remèdes (du plus rentable au moins rentable) (suite)

5. **Faire en parallèle** (mission 9) ;
6. **Micro-optimiser** (en dernier recours, sur le seul point chaud).

Vérifier l'équivalence

Une version plus rapide qui renvoie un résultat différent n'est pas une optimisation.
Méthodes :

- **tests existants** (HEF-110) ;
- **comparaison de sorties** de l'ancienne et de la nouvelle version sur des milliers d'entrées aléatoires (tests *différentiels*, graine fixée) ;
- **propriétés** (le résultat est trié, la somme est conservée) ;
- conserver l'ancienne version comme **référence** dans les tests.

Le concept

Garde-fous durables

Les tests de temps « il doit prendre moins de 50 ms » sont **fragiles** (dépendent de la machine). Mieux :

Garde-fou	Principe	Fiabilité
Comptage d'opérations	compter requêtes SQL, allocations, comparaisons, appels réseau	**déterministe**
Test de complexité	vérifier que le nombre d'opérations croît linéairement quand N double	déterministe
Rapport relatif	la nouvelle version est ≥ 3 fois plus rapide que la référence (même machine, même exécution)	robuste
Budget absolu en CI	seuils larges sur machine dédiée	utile mais bruité
Surveillance en production	percentiles, alertes (HEF-120)	indispensable

Le concept

Le rapport de performance

Section	Contenu
Question et contexte	quel problème, quel budget
Protocole	machine, version, données, nombre de répétitions, échauffement
Données brutes	fichier joint
Analyse	médiane, percentiles, intervalles
Goulot identifié	profil, preuve
Changement	description, diff, justification
Résultat avant/après	avec incertitude
Vérification	équivalence prouvée comment

Le concept

Le rapport de performance (suite)

Section	Contenu
Risques et coûts	complexité, maintenance, limites
Garde-fou	test ajouté
Suite	prochain goulot

Un lecteur doit pouvoir **refaire** la mesure et obtenir la même conclusion.

🧠 ***Astuce*** *: la meilleure optimisation est souvent un changement d'**échelle** : au lieu de faire plus vite 1 000 petites requêtes, en faire une. Cherche les gains de **×10**, ignore les gains de 5 %.*

Exemple pas à pas

Problème : optimiser une fonction de recommandation en suivant le protocole : baseline chronométrée et profilée, trois changements successifs, équivalence vérifiée par comparaison différentielle, garde-fous déterministes (comptage d'opérations, test de complexité), et rapport.

```
import cProfile, io, math, pstats, random, statistics, time

# ---------------- le code d'origine ----------------
COMPTEURS = {"comparaisons": 0}

def recommander_v0(membres, historiques, cours_populaires, k=3):
    # pour chaque membre, recommande les k cours les plus suivis par des membres «
    voisins » qu'il n'a pas déjà suivis
    resultat = {}
    for m in membres:
        deja = historiques[m]
        scores = {}
        for autre in membres:
            if autre == m:
```

Exemple pas à pas (suite)

```
                continue
            communs = 0
            for c in historiques[autre]:                        # intersection par
   boucle imbriquée
                for d in deja:
                    COMPTEURS["comparaisons"] += 1
                    if c == d: communs += 1
            if communs >= 2:
                for c in historiques[autre]:
                    if c not in deja:
                        scores[c] = scores.get(c, 0) + communs
        classes = sorted(scores.items(), key=lambda kv: (-kv[1], kv[0]))
        resultat[m] = [c for c, _ in classes[:k]]
    return resultat

def donnees(n_membres, graine=1):
    rng = random.Random(graine)
    cours = [f"C{i:03d}" for i in range(max(40, n_membres // 3))]       # le catal
   ogue grandit avec les membres
```

Exemple pas à pas (suite)

```
    membres = [f"m{i}" for i in range(n_membres)]
    historiques = {m: rng.sample(cours, rng.randint(3, 8)) for m in membres}
    return membres, historiques, cours

M, H, C = donnees(120)

def chrono(f, *args, n=5):
    mesures = []
    for _ in range(n):
        d = time.perf_counter(); f(*args); mesures.append(time.perf_counter() - d)
    return statistics.median(mesures)

# ---------------- 1. baseline et profil ----------------
base = recommander_v0(M, H, C)
t0 = chrono(recommander_v0, M, H, C)
pr = cProfile.Profile(); pr.enable(); recommander_v0(M, H, C); pr.disable()
print(f"baseline : {t0 * 1000:.0f} ms pour {len(M)} membres")
assert t0 > 0.02
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------------- 2. changement n°1 (hypothèse : l'intersection en boucle imbriqu
   ée domine) ----------------
def recommander_v1(membres, historiques, cours_populaires, k=3):
    ens = {m: set(h) for m, h in historiques.items()}              # intersection
   d'ensembles : O(min(|A|,|B|)) au lieu de O(|A|·|B|)
    resultat = {}
    for m in membres:
        deja, scores = ens[m], {}
        for autre in membres:
            if autre == m: continue
            communs = len(ens[autre] & deja)
            if communs >= 2:
                for c in historiques[autre]:
                    if c not in deja:
                        scores[c] = scores.get(c, 0) + communs
        classes = sorted(scores.items(), key=lambda kv: (-kv[1], kv[0]))
        resultat[m] = [c for c, _ in classes[:k]]
    return resultat
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------------- 3. changement n°2 (hypothèse : on parcourt tous les membres alo
   rs que seuls ceux partageant un cours comptent) ----------------
def recommander_v2(membres, historiques, cours_populaires, k=3):
    ens = {m: set(h) for m, h in historiques.items()}
    par_cours = {}
    for m in membres:
        for c in ens[m]: par_cours.setdefault(c, []).append(m)   # index inversé :
    cours → membres
    resultat = {}
    for m in membres:
        deja, communs = ens[m], {}
        for c in deja:
            for autre in par_cours[c]:
                if autre != m: communs[autre] = communs.get(autre, 0) + 1
        scores = {}
        for autre, n in communs.items():
            if n >= 2:
                for c in historiques[autre]:
                    if c not in deja: scores[c] = scores.get(c, 0) + n
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        classes = sorted(scores.items(), key=lambda kv: (-kv[1], kv[0]))
        resultat[m] = [c for c, _ in classes[:k]]
    return resultat

for nom, f in (("v1 (ensembles)", recommander_v1), ("v2 (index inversé)", recomman
   der_v2)):
    t = chrono(f, M, H, C)
    assert f(M, H, C) == base                                    # ÉQUIVALENCE
   sur l'entrée de référence
    print(f"{nom:20s} {t * 1000:7.1f} ms (x{t0 / t:.1f})")

# ---------------- 4. vérification différentielle : 300 jeux de données aléatoires
    ----------------
for graine in range(300):
    rng = random.Random(graine)
    m, h, c = donnees(rng.randint(2, 25), graine)
    ref = recommander_v0(m, h, c)
    assert recommander_v1(m, h, c) == ref and recommander_v2(m, h, c) == ref, grai
   ne
```

Exemple pas à pas (suite)

```
print("équivalence confirmée sur 300 jeux aléatoires")
assert chrono(recommander_v2, M, H, C) < t0 / 3

# ---------------- 5. garde-fou DÉTERMINISTE : compter les opérations et tester la
    complexité ----------------
def operations_v2(n):
    m, h, c = donnees(n, graine=5)
    ens = {x: set(v) for x, v in h.items()}; par_cours = {}
    for x in m:
        for cc in ens[x]: par_cours.setdefault(cc, []).append(x)
    ops = 0
    for x in m:
        for cc in ens[x]: ops += len(par_cours[cc])            # voisinages parco
   urus
    return ops

def operations_v0(n):
    COMPTEURS["comparaisons"] = 0
    m, h, c = donnees(n, graine=5); recommander_v0(m, h, c); return COMPTEURS["com
```

Exemple pas à pas (suite)

```
   paraisons"]

a0, b0 = operations_v0(60), operations_v0(120)
a2, b2 = operations_v2(300), operations_v2(600)
exposant0 = math.log2(b0 / a0); exposant2 = math.log2(b2 / a2)
print(f"v0 : doubler N multiplie les opérations par {b0 / a0:.1f} (exposant {expos
   ant0:.2f}) | v2 : par {b2 / a2:.1f} (exposant {exposant2:.2f})")
assert exposant0 > 1.8 and exposant2 < 1.3                    # quadratique contr
   e quasi linéaire

def test_garde_fou_complexite():
    assert 1.5 < operations_v2(1200) / operations_v2(600) < 2.6, "la recommandatio
   n n'est plus quasi linéaire"
test_garde_fou_complexite()
print("garde-fou de complexité : OK (indépendant de la machine)")

# ---------------- 6. le rapport ----------------
rapport = {"question": "recommandations trop lentes (budget : 1 s pour 1000 membre
   s)", "baseline_ms": round(t0 * 1000),
```

Exemple pas à pas (suite)

```
            "goulot": "intersection par boucles imbriquées puis parcours de tous le
    s membres", "changements": ["v1 : ensembles", "v2 : index inversé"],
            "equivalence": "300 jeux aléatoires + jeu de référence", "garde_fou": "
    test de complexité par comptage d'opérations"}
assert all(rapport.values())
```

Lecture :

1. La **baseline** est chronométrée (médiane de 5) et profilée. L'hypothèse n°1 vise le goulot identifié (l'intersection en boucle imbriquée) ; **un seul changement** à la fois.
2. Après chaque changement, la **sortie est identique** à la référence — et pas seulement sur un exemple : 300 jeux aléatoires comparent v0, v1 et v2.
3. Le **garde-fou** ne chronomètre pas (fragile) : il **compte les opérations** et vérifie que doubler les données ne fait que doubler (environ) le travail pour v2, alors que v0 le multipliait par ~4. Ce test est **déterministe** et indépendant de la machine.

Exemple pas à pas (suite)

4. Le **rapport** reprend le protocole : question, baseline, goulot, changements, équivalence, garde-fou.

À toi, avec aide

Décide, chiffres à l'appui, si ces trois optimisations valent la peine :

1. Une fonction exécutée **10 millions de fois par jour** passe de 3 ms à 1 ms. Le travail coûte 1 jour de développement (400 €) ; un cœur de serveur coûte 0,04 €/heure. Complexité ajoutée : modérée.
2. Une tâche manuelle exécutée **1 fois par mois** passe de 20 min à 2 min. Le travail coûte 3 jours (1 200 €).
3. Un traitement exécuté **50 fois par jour** par des **utilisateurs en attente** : passe de 8 s à 2 s. Un utilisateur attendant coûte en moyenne 0,5 € de productivité par minute de plus. Le travail coûte 2 jours (800 €).

Pour chacun : gain annuel, coût, retour, décision.

Indices progressifs

Indice 1

Gain annuel = temps économisé × fréquence × 365 × coût du temps.

Indice 2

Pour (1), convertis les cœurs-secondes en cœurs-heures puis en €.

Indice 3

Pour (3), le temps économisé est 6 s par exécution.

À toi, avec aide · réponse

Réponse

```
# 1. serveur : 2 ms économisées × 10 M/jour × 365 = 7,3 M s = 2 028 h → × 0,04 €
g1 = 0.002 * 10_000_000 * 365 / 3600 * 0.04
# 2. tâche mensuelle : 18 min × 12 = 216 min = 3,6 h ; au coût d'un salarié (≈ 40
   €/h)
g2 = 18 * 12 / 60 * 40
# 3. utilisateurs : 6 s × 50 × 365 = 109 500 s = 1 825 min × 0,5 €
g3 = 6 * 50 * 365 / 60 * 0.5
for n, g, c in ((1, g1, 400), (2, g2, 1200), (3, g3, 800)):
    print(f"cas {n} : gain annuel {g:8.0f} € | coût {c:5d} € | retour {g / c:5.2f}
    fois")
assert round(g1) == 81 and round(g2) == 144 and abs(g3 - 912.5) < 1
```

1. **81 €/an** pour un coût de 400 € : **non rentable** sur le seul coût de calcul (on le justifierait seulement si ces 3 ms étaient aussi de la latence ressentie par l'utilisateur).
2. **144 €/an** contre 1 200 € : **non** — à moins que la tâche soit pénible ou source d'erreurs.
3. **913 €/an** pour 800 € : **oui**, rentable dès la première année, et la valeur est surtout l'expérience utilisateur.

Réponse (suite)

On optimise ce qui **se répète** et ce que des **humains attendent**, pas ce qui est lent mais rare.

À toi, en solo

Test d'exposant de complexité. Écris un outil `exposant(f, tailles, mesurer)` qui estime l'ordre de croissance d'une fonction :

1. pour chaque taille N, produit la mesure (nombre d'opérations compté, ou temps médian) ;
2. ajuste une droite sur `log(mesure)` contre `log(N)` (régression par moindres carrés) et renvoie la **pente** (l'exposant) et le coefficient de détermination R^2 ;
3. teste-le sur quatre fonctions de comportement connu (linéaire, N log N, quadratique, cubique) en comptant les opérations ;
4. teste le **garde-fou** : une fonction qui devrait être linéaire mais devient quadratique est détectée ;
5. applique-le à ta propre fonction de recommandation (v0, v1, v2) et compare les exposants.

À toi, en solo · réponse

Pistes de correction

```
import math, random

def exposant(tailles, mesures):
    xs, ys = [math.log(n) for n in tailles], [math.log(m) for m in mesures]
    mx, my = sum(xs) / len(xs), sum(ys) / len(ys)
    pente = sum((x - mx) * (y - my) for x, y in zip(xs, ys)) / sum((x - mx) ** 2 f
   or x in xs)
    residus = sum((y - (my + pente * (x - mx))) ** 2 for x, y in zip(xs, ys))
    total = sum((y - my) ** 2 for y in ys)
    return pente, 1 - residus / total if total else 1.0

def ops_lineaire(n):
    c = 0
    for _ in range(n): c += 1
    return c
def ops_nlogn(n):
    c = 0
    for _ in range(n): c += max(1, int(math.log2(n)))
    return c
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def ops_quadratique(n):
    c = 0
    for i in range(n):
        for j in range(n): c += 1
    return c
def ops_cubique(n):
    c = 0
    for i in range(n):
        for j in range(n):
            for k in range(n // 10 or 1): c += 1
    return c

TAILLES = [20, 40, 80, 160, 320]
for nom, f in (("linéaire", ops_lineaire), ("N log N", ops_nlogn), ("quadratique",
    ops_quadratique), ("cubique", ops_cubique)):
    p, r2 = exposant(TAILLES, [f(n) for n in TAILLES])
    print(f"{nom:12s} exposant {p:4.2f}  R² = {r2:.3f}")
assert 0.95 < exposant(TAILLES, [ops_lineaire(n) for n in TAILLES])[0] < 1.05
assert 1.1 < exposant(TAILLES, [ops_nlogn(n) for n in TAILLES])[0] < 1.3
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert 1.95 < exposant(TAILLES, [ops_quadratique(n) for n in TAILLES])[0] < 2.05
assert 2.8 < exposant(TAILLES, [ops_cubique(n) for n in TAILLES])[0] < 3.2

def garde_fou(f, tailles, exposant_max):
    p, _ = exposant(tailles, [f(n) for n in tailles])
    assert p <= exposant_max, f"régression de complexité : exposant {p:.2f} > {exp
   osant_max}"
garde_fou(ops_lineaire, TAILLES, 1.2)
try:
    garde_fou(ops_quadratique, TAILLES, 1.2); raise SystemExit("aurait dû échouer"
   )
except AssertionError as e:
    print("garde-fou :", e)
```

Un **exposant** est une signature robuste du comportement algorithmique : ni la vitesse de la machine, ni le bruit n'y changent grand-chose, tant qu'on mesure des **opérations** plutôt que des secondes. C'est le garde-fou idéal contre le retour d'un $O(n^2)$ glissé par inadvertance dans un correctif.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Optimiser sans mesure ni profil	On améliore ce qui est froid	Baseline, profil, hypothèse
Plusieurs changements à la fois	On ne sait pas ce qui a aidé	Un changement, une mesure
Ne pas vérifier l'équivalence	Une version rapide mais fausse	Tests différentiels, version de référence
Aucun garde-fou	Le gain disparaît au prochain changement	Comptage d'opérations, test de complexité, budget en CI

Mini-boss

Dossier de performance complet de HashBox (projet de fin de module). Choisis le traitement de ton choix (ou prends « import des inscriptions + calcul des statistiques mensuelles + export ») avec un budget : 1 million d'enregistrements en moins de 10 secondes et moins de 200 Mo de mémoire.

1. écris la version initiale et sa suite de tests de correction ;
2. mesure la baseline avec ton banc (mission 2), temps et mémoire ;
3. profile (CPU et mémoire), désigne les goulots avec preuves ;
4. applique au moins **quatre** optimisations successives issues du module (algorithme/structure, E/S, base de données ou cache, concurrence), chacune avec hypothèse, mesure avec incertitude et vérification d'équivalence différentielle ;
5. pose des garde-fous déterministes (opérations comptées, exposant, requêtes, allocations) et un garde-fou relatif ;
6. décide à quel moment tu t'arrêtes (budget atteint, rendement décroissant) en chiffrant le coût/bénéfice ;

Mini-boss

Mini-boss (suite)

7. rédige le **rapport** complet (protocole, données brutes, analyse, avant/après, risques, suite) et fournis un script qui **reproduit** toutes les mesures ;
8. conclus par une rétrospective de 10 lignes : quelle optimisation a eu le meilleur rapport gain/effort, laquelle n'en valait pas la peine, et ce que tu changerais dans ta méthode.

Mini-boss · réponse

Pistes de correction

Un bon dossier : tableau d'étapes (changement, hypothèse, temps médian ± intervalle, mémoire, équivalence : 500 jeux aléatoires), gains typiques dans l'ordre (index/structure de données ×10-50, E/S regroupées ×5-20, cache ×2-5, parallélisme ×1,5-3 si applicable), rendement décroissant évident, garde-fous (test d'exposant, compteur de requêtes SQL, budget relatif ≥ 3× la référence), analyse coût/bénéfice justifiant l'arrêt, rapport reproductible (script unique). La rétrospective honnête mentionne au moins une idée qui n'a rien apporté ou qui a été abandonnée.

Quiz éclair

1. Avant de modifier du code lent, il faut :

A. mesurer et profiler pour identifier le goulot
B. réécrire le code en C
C. ajouter du cache partout

2. Après une optimisation, il faut vérifier que :

A. le code est plus court
B. le résultat est strictement équivalent à la version précédente
C. le fichier est plus petit

3. Un garde-fou de performance robuste compte de préférence :

A. les lignes de commentaires

Quiz éclair

Quiz éclair (suite)

B. les fichiers du dépôt
C. les opérations (requêtes, comparaisons) plutôt que les secondes

4. 🔁 *Révision mission 09* : Les threads Python accélèrent surtout :

A. les tâches limitées par les entrées-sorties (attentes)
B. le calcul numérique pur
C. la compilation

5. 🔁 *Révision mission 09* : Face à une surcharge, la contre-pression consiste à :

A. augmenter la file sans limite
B. refuser ou ralentir le travail entrant plutôt que d'accumuler une file infinie
C. redémarrer le serveur

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quelle optimisation de ton projet a été faite sans mesure, et en connais-tu le gain réel ?
2. Quel garde-fou protège aujourd'hui la performance de ton traitement le plus critique ?

À retenir

À retenir

1. Protocole : baseline, profil, hypothèse, un changement, mesure, équivalence, garde-fou.
2. Ordre des remèdes : ne pas faire, faire moins, mieux organiser, faire une fois, en parallèle, micro-optimiser.
3. Optimiser seulement si le gain l'emporte sur le coût et le risque.
4. Garde-fous déterministes : opérations comptées, exposant de complexité.
5. Rapport reproductible : protocole, données brutes, résultat avec incertitude.

Félicitations

Félicitations

Tu as terminé **HEF-130 — Performance**. Tu sais modéliser la capacité d'un système, mesurer proprement avec une incertitude, profiler CPU et mémoire, optimiser entrées-sorties, réseau, requêtes et caches, raisonner sur concurrence et saturation, puis conduire et valider une optimisation avec des garde-fous durables.
→ Prochain module : **HEF-140 — Systèmes distribués**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Optimiseur méthodique

Fin du module HEF-130. Bravo !

