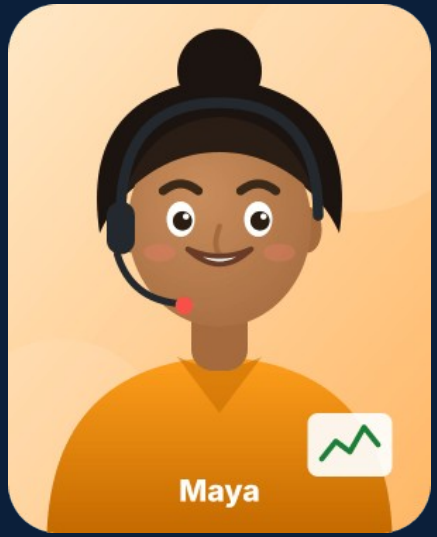
HEF-130 · MISSION 02 / 10

Métriques et budgets

On n'améliore que ce qu'on mesure bien. Mesurer mal est pire que ne pas mesurer : on croit savoir.



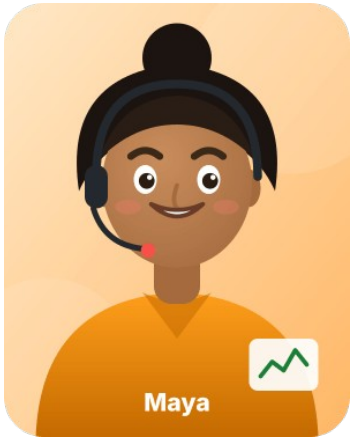


Badge : Mesureur rigoureux

95 min · avancé

L'histoire

Kodi annonce fièrement : « J'ai optimisé la recherche, elle est passée de 120 ms à 95 ms ! » Sam lui demande : « Combien de mesures ? Sur quelle machine ? La première exécution comptait-elle ? Quelle est la dispersion ? » Kodi réalise qu'il a lancé le programme deux fois, de sa machine surchargée, avec un navigateur ouvert. Refaites avec 30 mesures : 100 ms ± 25. La « gain » était du bruit.



> *« Une mesure est une variable aléatoire, dit Sam. Pas un fait. Pour conclure, il faut un protocole : échauffement, répétitions, dispersion, comparaison honnête. Et un budget pour savoir quand on a le droit de s'arrêter. »*

Tu apprends à mesurer proprement, à lire des percentiles, à comparer deux versions avec une incertitude, et à fixer des budgets de performance.

Ta mission

À la fin, je serai capable de…

1. choisir les bonnes statistiques (médiane, percentiles, dispersion) plutôt que la moyenne seule
2. écrire un protocole de mesure : échauffement, répétitions, isolation
3. estimer l'incertitude d'une mesure (intervalle bootstrap) et comparer deux versions honnêtement
4. reconnaître l'omission coordonnée dans les tests de charge
5. définir un budget de performance et un SLO de latence

Défi d'échauffement

Défi d'échauffement

Tu veux savoir si un nouveau trajet pour aller au travail est plus rapide. Tu le fais une fois : 28 min contre 31 min pour l'ancien. Peux-tu conclure ? Combien d'essais voudrais-tu ? Que ferais-tu des jours de pluie, des heures de pointe, du premier trajet où tu t'es trompé de rue ?

Le concept

Une mesure est bruitée

La durée d'un programme varie : cache chaud ou froid, autres processus, fréquence du processeur, ramasse-miettes, réseau. Il faut donc **répéter** et décrire la **distribution**, pas une valeur.

Les statistiques utiles

Statistique	Sens	Quand
Moyenne	somme / nombre	sensible aux valeurs extrêmes ; pour le coût total
Médiane (p50)	la moitié des mesures est en dessous	« valeur typique »
Percentiles p95, p99	95 % / 99 % des mesures sont en dessous	expérience des moins chanceux
Minimum	meilleur cas	pour les micro-mesures : le moins bruité

Le concept

Les statistiques utiles (suite)

Statistique	Sens	Quand
Écart interquartile (IQR)	dispersion du centre (p75 − p25)	robuste
Écart-type	dispersion autour de la moyenne	si distribution sans queue lourde

Les latences réelles ont une **queue lourde** : peu de requêtes très lentes. Un service dont la moyenne est de 100 ms peut avoir un p99 de 2 s. Quand un utilisateur fait 20 requêtes pour afficher une page, la probabilité qu'au moins une soit dans le p99 est de **$1 - 0{,}99^{20} \approx 18\ \%$**.

Un protocole de mesure honnête

1. **Question précise** : « la fonction X est-elle plus rapide que Y pour des entrées de taille N ? »
2. **Environnement contrôlé** : machine calme, mêmes versions, même entrée.
3. **Échauffement** (*warmup*) : on jette les premières exécutions (caches, compilation paresseuse).

Le concept

Un protocole de mesure honnête (suite)

4. **Répétitions** suffisantes (30 ou plus) ; ordre **entrelacé** pour A et B (ABABAB) afin d'éviter une dérive.
5. **Mesurer le bon périmètre** : ni trop petit (bruit de l'horloge), ni trop large.
6. **Enregistrer les données brutes**, pas seulement un résumé.
7. **Comparer avec incertitude** : intervalle de confiance, pas deux nombres.
8. **Vérifier la correction** : une version plus rapide qui donne un mauvais résultat n'est pas une optimisation.

En Python : `time.perf_counter()` pour l'horloge, `timeit` pour les micro-mesures (désactive le ramasse-miettes pendant la mesure et répète).

Incertitude par rééchantillonnage (*bootstrap*)

Le concept

Incertitude par rééchantillonnage (*bootstrap*) (suite)

Pour une statistique (la médiane par exemple) : tirer **avec remise** B échantillons de même taille parmi les mesures, calculer la statistique de chacun, et prendre l'intervalle central 95 % des résultats. Si les intervalles de A et de B **se chevauchent largement**, la différence n'est pas démontrée.

L'omission coordonnée (*coordinated omission*)

Un outil de test de charge qui envoie la requête suivante **après** la réponse à la précédente sous-estime la latence : quand le serveur se fige 5 secondes, l'outil n'envoie rien pendant ce temps, donc **aucune** mesure ne reflète l'attente des requêtes qui auraient dû partir. Remède : un générateur de charge **en boucle ouverte** — les requêtes partent selon un calendrier fixe, indépendamment des réponses, et la latence est mesurée **depuis l'heure prévue**.

Budgets de performance

Un **budget** est un seuil qu'on s'interdit de dépasser :

Le concept

Budgets de performance (suite)

Domaine	Exemples de budgets
Web (côté utilisateur)	page interactive en moins de 3 s, 200 Ko de JavaScript
API	p95 < 300 ms, p99 < 1 s
Traitement par lots	un lot de 1 M d'enregistrements en moins de 10 min
Ressources	mémoire < 512 Mo, CPU < 70 % en pointe
Coût	< 0,002 € par requête

Un budget a un **propriétaire**, est **mesuré en continu** (en CI et en production) et déclenche une alerte ou un échec de pipeline quand il est dépassé. Il s'exprime idéalement comme un **SLO** (« 99 % des requêtes en moins de 300 ms sur 30 jours », HEF-120).

Le concept

Apdex

L'indice **Apdex** résume la satisfaction avec un seuil T : requêtes **satisfaisantes** (≤ T), **tolérables** (entre T et 4T), **frustrantes** (> 4T) :

```
Apdex = (satisfaisantes + tolérables / 2) / total
```

🧠 ***Astuce*** *: quand quelqu'un te donne un chiffre de performance, demande : combien de mesures, quelle dispersion, quelles conditions, et par rapport à quoi ? Sans réponses, ce n'est pas encore une information.*

Exemple pas à pas

Problème : construire un banc de mesure rigoureux (échauffement, répétitions, entrelacement, bootstrap), l'utiliser pour comparer deux implémentations, puis montrer l'omission coordonnée sur des données simulées.

```
import math, random, statistics, time

# ---------------- banc de mesure ----------------
def mesurer(fonction, repetitions=30, echauffement=3):
    for _ in range(echauffement):
        fonction()
    durees = []
    for _ in range(repetitions):
        debut = time.perf_counter()
        fonction()
        durees.append(time.perf_counter() - debut)
    return durees

def percentile(valeurs, p):
```

Exemple pas à pas (suite)

```
    v = sorted(valeurs)
    return v[min(len(v) - 1, math.ceil(p / 100 * len(v)) - 1)]

def intervalle_bootstrap(valeurs, statistique=statistics.median, b=2000, graine=1)
   :
    rng = random.Random(graine)
    tirages = sorted(statistique([rng.choice(valeurs) for _ in valeurs]) for _ in
   range(b))
    return tirages[int(0.025 * b)], tirages[int(0.975 * b)]

def comparer(f_a, f_b, repetitions=30):
    a, b = [], []
    for _ in range(3):
        f_a(); f_b()                                    # échauffement
    for _ in range(repetitions):                        # mesures ENTRELACÉES : p
   as de dérive systématique
        d = time.perf_counter(); f_a(); a.append(time.perf_counter() - d)
        d = time.perf_counter(); f_b(); b.append(time.perf_counter() - d)
    return a, b
```

Exemple pas à pas (suite)

```
# ---- deux façons de tester l'appartenance ----
N = 20000
liste = list(range(N))
ensemble = set(liste)
cibles = [N - 1 - (i * 37) % N for i in range(200)]       # 200 recherches

def avec_liste():
    return sum(1 for c in cibles if c in liste)           # O(n) par recherche

def avec_ensemble():
    return sum(1 for c in cibles if c in ensemble)        # O(1) par recherche

a, b = comparer(avec_liste, avec_ensemble)
for nom, d in (("liste", a), ("ensemble", b)):
    lo, hi = intervalle_bootstrap(d)
    print(f"{nom:9s} médiane {statistics.median(d) * 1000:8.3f} ms  IC95 [{lo * 10
   00:8.3f} ; {hi * 1000:8.3f}]  p95 {percentile(d, 95) * 1000:8.3f} ms")
(a_lo, a_hi), (b_lo, b_hi) = intervalle_bootstrap(a), intervalle_bootstrap(b)
```

Exemple pas à pas (suite)

```
assert avec_liste() == avec_ensemble() == 200           # même résultat : la comp
   araison est valide
assert b_hi < a_lo                                      # intervalles disjoints :
    la différence est démontrée
assert statistics.median(a) > 20 * statistics.median(b)  # et l'écart est très gra
   nd

# ---- deux variantes quasi identiques : l'intervalle ne permet PAS de conclure --
   --
def v1(): return sorted(range(2000), key=lambda x: -x)
def v2(): return sorted(range(2000), key=lambda x: -x)
x, y = comparer(v1, v2)
(x_lo, x_hi), (y_lo, y_hi) = intervalle_bootstrap(x), intervalle_bootstrap(y)
chevauchement = max(x_lo, y_lo) <= min(x_hi, y_hi)
print("deux fonctions identiques : les intervalles se chevauchent ?", chevauchemen
   t)
assert chevauchement                                    # on ne déclarera pas de
   « gain » sur du bruit
```

Exemple pas à pas (suite)

```
# ---------------- omission coordonnée ----------------
def charge(boucle_ouverte, graine=7, n=1000, intervalle=0.010):
    # le serveur répond en 5 ms, sauf un gel de 2 s entre t = 3 s et t = 5 s (il r
   épond alors à la fin du gel)
    rng = random.Random(graine)
    latences, t_serveur_libre = [], 0.0
    horloge_client = 0.0
    for i in range(n):
        prevu = i * intervalle                                  # calendrier thé
   orique
        envoi = prevu if boucle_ouverte else max(prevu, horloge_client)
        debut = max(envoi, t_serveur_libre)
        if 3.0 <= debut < 5.0:
            debut = 5.0                                         # gel
        fin = debut + 0.005
        t_serveur_libre = fin
        latences.append(fin - (prevu if boucle_ouverte else envoi))   # boucle ouv
   erte : mesurée depuis l'heure PRÉVUE
        horloge_client = fin
```

Exemple pas à pas (suite)

```
    return latences

fermee, ouverte = charge(False), charge(True)
print(f"boucle fermée : p99 = {percentile(fermee, 99) * 1000:7.0f} ms | boucle ouv
   erte : p99 = {percentile(ouverte, 99) * 1000:7.0f} ms")
assert percentile(ouverte, 99) > 20 * percentile(fermee, 99)       # la boucle fer
   mée masque le gel

# ---------------- Apdex ----------------
def apdex(latences, T):
    sat = sum(1 for l in latences if l <= T); tol = sum(1 for l in latences if T <
    l <= 4 * T)
    return (sat + tol / 2) / len(latences)
print(f"Apdex (T = 100 ms) : boucle fermée {apdex(fermee, 0.1):.2f} ; boucle ouver
   te {apdex(ouverte, 0.1):.2f}")
assert apdex(ouverte, 0.1) < apdex(fermee, 0.1)
```

Lecture :

Exemple pas à pas (suite)

1. La **médiane** et un **intervalle bootstrap** décrivent chaque variante. Pour `liste` contre `ensemble`, les intervalles sont disjoints et l'écart est de plusieurs ordres de grandeur : la différence est **démontrée**, et le résultat est identique (200), donc la comparaison est valide.
2. Pour deux fonctions **identiques**, les intervalles se chevauchent : on ne prétend pas avoir trouvé un gain — l'écart observé n'est que du bruit.
3. Avec l'**omission coordonnée**, un outil en boucle fermée voit un p99 anodin malgré un gel de 2 s ; la boucle ouverte, mesurée depuis l'heure prévue, révèle la vraie expérience des clients qui attendaient.
4. L'**Apdex** résume la satisfaction et distingue les deux protocoles de mesure.

À toi, avec aide

Un service a les latences (en ms) suivantes pour 20 requêtes :

```
[42, 38, 45, 40, 41, 39, 44, 43, 37, 40, 41, 39, 42, 46, 40, 38,
41, 900, 1200, 43]
```

Calcule la moyenne, la médiane, p90, p95, l'écart-type, puis dis :

1. laquelle de ces statistiques représente le mieux l'expérience *typique* ? laquelle représente la *queue* ?
2. si le budget est « p95 < 100 ms », est-il respecté ?
3. si une page effectue 10 de ces appels en séquence, quelle est la probabilité qu'au moins un tombe parmi les 10 % les plus lents ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise `statistics.mean`, `median`, `pstdev`, et la fonction `percentile` de l'exemple.

Indice 2

Pour la probabilité : $1 - (0{,}9)^{10}$.

Indice 3

Deux valeurs sur vingt sont de grosses valeurs aberrantes.

À toi, avec aide · réponse

Réponse

```
import math, statistics
L = [42, 38, 45, 40, 41, 39, 44, 43, 37, 40, 41, 39, 42, 46, 40, 38, 41, 900, 1200
   , 43]
def percentile(v, p):
    v = sorted(v); return v[min(len(v) - 1, math.ceil(p / 100 * len(v)) - 1)]
print(f"moyenne {statistics.mean(L):.1f} | médiane {statistics.median(L)} | p90 {p
   ercentile(L, 90)} | p95 {percentile(L, 95)} | écart-type {statistics.pstdev(L):
   .1f}")
print(f"probabilité qu'au moins un appel sur 10 soit dans les 10 % les plus lents
   : {1 - 0.9 ** 10:.1%}")
assert statistics.median(L) == 41 and percentile(L, 95) == 900 and round(statistic
   s.mean(L)) == 142
```

1. La **médiane (41 ms)** décrit l'expérience typique ; **p95 = 900 ms** décrit la queue. La **moyenne (142 ms)** ne représente personne : elle est tirée par deux valeurs extrêmes.
2. Le budget p95 < 100 ms **n'est pas respecté** (900 ms).

À toi, avec aide · réponse

Réponse (suite)

3. **65 %** : sur une page à 10 appels, il est probable que la page soit ralentie par au moins un appel lent. C'est pourquoi on se préoccupe des **percentiles élevés**, pas de la moyenne.

À toi, en solo

Garde-fou de performance en CI. Écris `verifier_budget(mesures_base, mesures_nouvelles, budget)` qui décide si un changement est acceptable :

1. échoue si le **p95 absolu** dépasse le budget ;
2. échoue si la **médiane** a augmenté de plus de 10 % **et** que les intervalles bootstrap ne se chevauchent pas (régression démontrée) ;
3. renvoie « incertain » (avertissement) si la médiane semble pire mais que les intervalles se chevauchent ;
4. teste sur 4 cas simulés à graine fixée : identique (bruit), régression de 30 %, amélioration de 25 %, dépassement du budget absolu.

À toi, en solo · réponse

Pistes de correction

```
import math, random, statistics

def percentile(v, p):
    v = sorted(v); return v[min(len(v) - 1, math.ceil(p / 100 * len(v)) - 1)]

def ic(v, b=2000, graine=1):
    rng = random.Random(graine)
    t = sorted(statistics.median([rng.choice(v) for _ in v]) for _ in v and range(
   b))
    return t[int(0.025 * b)], t[int(0.975 * b)]

def verifier_budget(base, nouvelles, budget_p95):
    if percentile(nouvelles, 95) > budget_p95:
        return "échec", f"p95 {percentile(nouvelles, 95):.1f} > budget {budget_p95
   }"
    m0, m1 = statistics.median(base), statistics.median(nouvelles)
    (lo0, hi0), (lo1, hi1) = ic(base), ic(nouvelles)
    chevauche = max(lo0, lo1) <= min(hi0, hi1)
    if m1 > 1.10 * m0:
```

Pistes de correction (suite)

```
        return ("incertain", "médiane en hausse mais bruit possible") if chevauche
    else ("échec", f"régression démontrée : {m0:.1f} → {m1:.1f}")
    return "ok", f"médiane {m0:.1f} → {m1:.1f}"

def echantillon(moyenne, graine, n=40, bruit=0.08):
    rng = random.Random(graine)
    return [rng.lognormvariate(math.log(moyenne), bruit) for _ in range(n)]

base = echantillon(50, 1)
cas = {
    "identique (bruit)": (echantillon(50, 2), 200),
    "régression de 30 %": (echantillon(65, 3), 200),
    "amélioration de 25 %": (echantillon(37.5, 4), 200),
    "budget dépassé": (echantillon(150, 5), 100),
}
res = {nom: verifier_budget(base, n, budget)[0] for nom, (n, budget) in cas.items(
   )}
for nom, (n, budget) in cas.items():
    print(f"{nom:22s} → {verifier_budget(base, n, budget)}")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert res == {"identique (bruit)": "ok", "régression de 30 %": "échec", "améliora
   tion de 25 %": "ok", "budget dépassé": "échec"}
```

Le garde-fou distingue **trois situations** : régression démontrée (échec), bruit (ok), et un cas **incertain** (à surveiller ou à remesurer). Brancher cette vérification dans la CI évite les régressions silencieuses tout en évitant de bloquer pour du bruit.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Une seule mesure	Du bruit pris pour un résultat	30+ répétitions, médiane et intervalle
Ne regarder que la moyenne	La queue de latence est invisible	Percentiles p95 / p99
Inclure l'échauffement et le premier appel	Mesure faussée par les caches froids	Jeter les premières exécutions
Test de charge en boucle fermée	Omission coordonnée : latences sous-estimées	Boucle ouverte, latence depuis l'heure prévue

Mini-boss

Mini-boss

Banc de mesure de HashBox. Construis `banc.py` :

1. un runner qui mesure une fonction avec échauffement, N répétitions, mesures entrelacées pour comparer A et B ;
2. des statistiques : médiane, p95, p99, IQR, intervalle bootstrap ; Apdex ;
3. un générateur de charge en boucle **ouverte** et en boucle fermée contre un serveur simulé avec gel, pour montrer l'omission coordonnée ;
4. une vérification de **budget** (p95 absolu, régression démontrée, incertain) ;
5. un rapport texte : protocole, données brutes (fichier), résumé, décision ;
6. tests : bruit pur → pas de faux gain sur 100 comparaisons de fonctions identiques (le taux de fausses alertes doit être ≤ 10 %) ; régression de 25 % détectée dans au moins 90 % des cas ; assertions sur le rapport.

Mini-boss · réponse

Pistes de correction

Un bon travail : fonctions `mesurer`, `comparer` entrelacée, `intervalle_bootstrap`, `percentile`; `charge(boucle_ouverte)` comme dans l'exemple; `verifier_budget`; rapport avec tableau et mention du protocole; test statistique de la fausse alerte (100 paires de fonctions identiques simulées par des échantillons lognormaux à graine fixée, taux de faux positifs mesuré). La conclusion attendue : la détection fiable d'un petit gain demande plus de répétitions, et on rapporte toujours l'incertitude.

Quiz éclair

Quiz éclair

1. Pour décrire la latence ressentie par les moins chanceux, on regarde :

A. la moyenne uniquement
B. le nombre de serveurs
C. les percentiles élevés (p95, p99)

2. Avant de mesurer une fonction, on fait un échauffement pour :

A. écarter les premières exécutions à caches froids
B. ralentir la machine
C. économiser de la mémoire

3. Si les intervalles de confiance de deux versions se chevauchent largement :

A. la version B est meilleure

Quiz éclair (suite)

B. la différence n'est pas démontrée
C. la version A est meilleure

4. 🔁 *Révision mission 01* : La loi de Little relie :

A. CPU, mémoire et disque
B. coût, délai et qualité
C. concurrence, débit et latence moyenne

5. 🔁 *Révision mission 01* : D'après Amdahl, si 90 % d'un programme est parallélisable, l'accélération maximale est :

A. environ ×10
B. infinie

Quiz éclair

Quiz éclair (suite)

C. ×90

Quiz éclair · réponse

Réponses

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

Je réfléchis

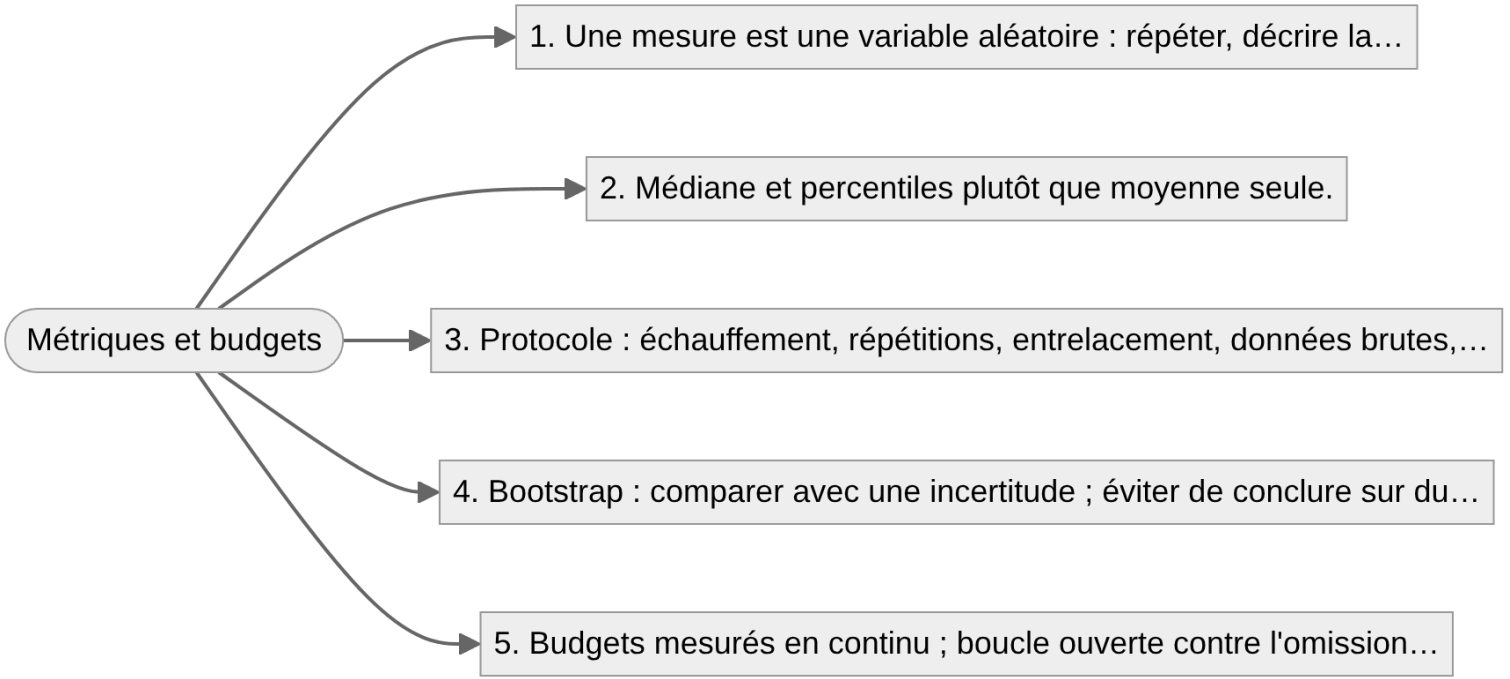
1. Quel chiffre de performance as-tu récemment accepté sans en connaître le protocole de mesure ?
2. Quel budget de performance ton projet devrait-il avoir, et qui en serait responsable ?

À retenir

1. Une mesure est une variable aléatoire : répéter, décrire la distribution.
2. Médiane et percentiles plutôt que moyenne seule.
3. Protocole : échauffement, répétitions, entrelacement, données brutes, correction vérifiée.
4. Bootstrap : comparer avec une incertitude ; éviter de conclure sur du bruit.
5. Budgets mesurés en continu ; boucle ouverte contre l'omission coordonnée.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Mesureur rigoureux

Prochaine mission : Profilage CPU

