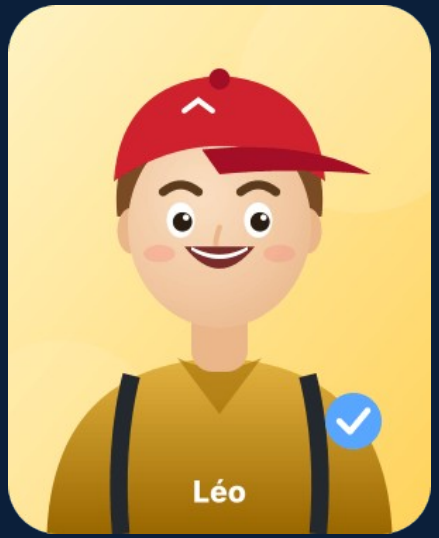
HEF-160 · MISSION 08 / 10

Vérification humaine et tests

Le code généré par une IA est une proposition, jamais une preuve. Entre la proposition et l'intégration, il y a une échelle de vérification — et elle se construit.



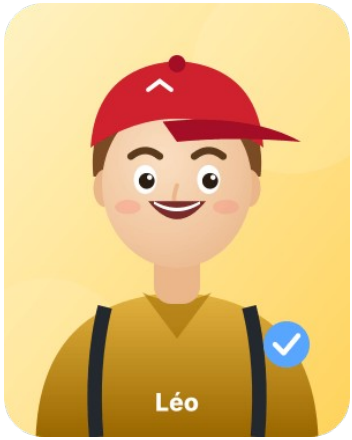


Badge : Vérificateur impitoyable

115 min · avancé

L'histoire

Un développeur de HashBox demande à un assistant une fonction qui fusionne des intervalles de dates (réservations de salles). Le code arrive, élégant, commenté, avec trois exemples qui passent. Il est fusionné. Deux semaines plus tard, une salle est réservée deux fois : la fonction ne fusionne pas les intervalles qui se touchent exactement, modifie la liste d'entrée, et plante sur une liste vide. Personne n'avait lu le code ligne à ligne : « il passait ses tests ». Ses propres tests, écrits par la même IA.



« Un test écrit par l'auteur d'une solution partage ses angles morts, dit Sam. Quand l'auteur est une IA, c'est pareil. La vérification doit être indépendante de la génération : une spécification écrite par toi, des tests de propriétés, une comparaison avec une implémentation de référence lente mais évidente, une analyse statique, des dépendances qui existent vraiment. Et c'est ton nom qui est sur le commit. »

L'histoire (suite)

Tu apprends l'échelle de vérification d'un code généré : spécification, relecture, analyse statique (imports inventés, appels dangereux), tests d'exemples, tests de propriétés, tests différentiels, mutation, exécution isolée — et tu construis un vérificateur automatique qui trie des candidats « produits par une IA ».

Ta mission

À la fin, je serai capable de…

1. énoncer l'échelle de vérification d'une sortie générée (spécification, lecture, analyse statique, tests, propriétés, différentiel, mutation)
2. détecter dépendances et API inventées, appels dangereux et effets de bord
3. écrire des tests de propriétés et un test différentiel contre une implémentation de référence
4. exécuter du code non fiable dans un processus isolé avec délai
5. vérifier des citations et références produites par un modèle

Défi d'échauffement

Défi d'échauffement

Un entrepreneur vous apporte un plan que son assistant a rédigé : tableaux, chiffres, références à des études. Vous devez le valider avant qu'il parte à la banque. Que vérifies-tu en premier ? Les chiffres cités ? L'existence des études ? La cohérence des totaux ? Comment t'y prends-tu si le plan fait 80 pages ? Quelle partie peux-tu vérifier mécaniquement ?

Le concept

Le principe : indépendance

> *Ce qui a été **produit** par une source ne peut pas être **validé** par la même source avec les mêmes angles morts.*

Un modèle qui écrit le code **et** ses tests peut reproduire la même fausse hypothèse dans les deux. La vérification repose sur des éléments que **tu** contrôles : une **spécification** écrite par un humain, un **oracle** (référence, propriété, règle métier), des **outils** déterministes.

Le principe de coût : vérifier doit coûter moins que produire

Si vérifier une sortie prend plus de temps que l'écrire soi-même, l'IA n'apporte rien. Les bons cas d'usage : **production coûteuse, vérification bon marché** (un test qui passe, une compilation, une requête qu'on relance, un calcul refaisable). Les mauvais : sorties difficiles à valider sans expertise.

Le concept

L'échelle de vérification

Niveau	Vérification	Détecte
0	**Spécification écrite** avant de demander	ambiguïtés, oublis
1	**Lecture** (comprendre chaque ligne, pouvoir l'expliquer)	logique fausse, code inutile
2	**Analyse statique** : syntaxe, imports qui existent, appels dangereux, types, lint	API inventées, `eval`, secrets
3	**Tests d'exemples** (cas nominaux, limites, erreurs)	bugs évidents
4	**Tests de propriétés** (aléatoires, invariants)	cas imprévus, effets de bord
5	**Test différentiel** contre une référence simple et lente	divergences de comportement
6	**Mutation** : les tests détectent-ils des fautes injectées ? (HEF-110)	tests faibles

Le concept

L'échelle de vérification (suite)

Niveau	Vérification	Détecte
7	**Revue de sécurité** (HEF-100), de performance (HEF-130)	failles, lenteurs
8	**Essai en conditions réelles** progressif (drapeau, canary, HEF-120)	écarts d'environnement

On monte l'échelle selon l'**enjeu**.

Les défauts typiques du code généré

Défaut	Exemple	Détecteur
Dépendance ou fonction inventée	`import intervaltools` (inexistant)	vérifier que les modules/attributs existent
Cas limites oubliés	liste vide, bornes qui se touchent	tests de limites, propriétés

Le concept

Les défauts typiques du code généré (suite)

Défaut	Exemple	Détecteur
Effet de bord	modifie l'argument	propriété « l'entrée n'est pas modifiée »
Version de l'API périmée	fonction supprimée	exécution réelle
Appel dangereux	`eval`, `os.system`, `pickle.loads`	analyse statique (HEF-100)
Sécurité subtile	injection SQL par concaténation	revue + SAST
Complexité cachée	$O(n^2)$ là où $O(n \log n)$ suffit	mesures, compteurs d'opérations
Sur-ingénierie	abstractions inutiles	lecture, simplicité
Commentaires trompeurs	le commentaire ne correspond pas au code	lecture

Le concept

Les tests de propriétés

Au lieu de choisir des exemples, énoncer des **propriétés** vraies pour toute entrée et les vérifier sur des milliers d'entrées aléatoires (graine fixée). Pour une fusion d'intervalles : (1) le résultat est trié et **sans chevauchement** ; (2) il **couvre exactement les mêmes points** que l'entrée ; (3) l'entrée **n'est pas modifiée** ; (4) fusionner deux fois = fusionner une fois (idempotence).

Le test différentiel

Écrire (ou obtenir d'une source fiable) une implémentation **trop simple pour être fausse** (par exemple, lister tous les points entiers couverts), puis comparer sur beaucoup d'entrées aléatoires. Les divergences désignent des bugs.

Le concept

Exécuter du code non fiable

Code généré = code **non fiable** tant qu'il n'est pas vérifié. L'exécuter : dans un **processus séparé**, avec **délai maximal**, **sans accès** réseau/fichiers sensibles (conteneur, utilisateur sans droits), **mémoire limitée** (HEF-100, HEF-120).

Citations et références

Les modèles produisent des références **plausibles mais inexistantes**. Vérifier : l'**existence** (URL, identifiant, DOI), que la source **contient bien** l'affirmation, que les **chiffres** correspondent. Un vérificateur automatique peut contrôler l'existence ; la concordance relève d'une lecture humaine.

La responsabilité

Le concept

La responsabilité (suite)

Celui qui fusionne répond du code. *« C'est l'IA qui l'a écrit » n'est pas un argument de revue. La traçabilité (quelle IA, quel prompt, quelle vérification) aide à comprendre, pas à se décharger.*

Éviter l'atrophie des compétences

Si tu ne fais plus que valider sans comprendre, tu perds la capacité de repérer les erreurs. Parades : **expliquer le code** à voix haute (« explain-back »), écrire d'abord ta propre approche pour les tâches importantes, relire sans l'IA de temps en temps, pratiquer le débogage.

🧠 ***Astuce*** *: avant de lire le code généré, écris tes* ***trois cas de test les plus vicieux****. Si tu ne sais pas lesquels, tu ne sais pas encore ce que la fonction doit faire.*

Exemple pas à pas

Exemple pas à pas

Problème : construire un vérificateur automatique qui trie cinq candidats « produits par une IA » pour une fonction `fusionner(intervalles)`. Les contrôles : analyse statique (imports inexistants, appels dangereux), exécution isolée avec délai dans un sous-processus, tests d'exemples, **tests de propriétés** et **test différentiel** contre une référence évidente.

```
import ast, importlib.util, json, subprocess, sys, textwrap

# ---------------- les candidats « générés par une IA » ----------------
CANDIDATS = {
"A_correct": '''
def fusionner(intervalles):
    resultat = []
    for debut, fin in sorted(intervalles):
        if resultat and debut <= resultat[-1][1]:
            resultat[-1] = (resultat[-1][0], max(resultat[-1][1], fin))
        else:
            resultat.append((debut, fin))
```

Exemple pas à pas (suite)

```
    return resultat
''',
"B_bornes_qui_se_touchent": '''
def fusionner(intervalles):
    resultat = []
    for debut, fin in sorted(intervalles):
        if resultat and debut < resultat[-1][1]:
            resultat[-1] = (resultat[-1][0], max(resultat[-1][1], fin))
        else:
            resultat.append((debut, fin))
    return resultat
''',
"C_dependance_inventee": '''
from intervaltools import merge_intervals
def fusionner(intervalles):
    return merge_intervals(intervalles)
''',
"D_modifie_l_entree": '''
def fusionner(intervalles):
```

Exemple pas à pas (suite)

```
    intervalles.sort()
    resultat = [intervalles[0]]
    for debut, fin in intervalles[1:]:
        if debut <= resultat[-1][1]:
            resultat[-1] = (resultat[-1][0], max(resultat[-1][1], fin))
        else:
            resultat.append((debut, fin))
    return resultat
''',
"E_appel_dangereux": '''
import os
def fusionner(intervalles):
    os.system("echo exfiltration")
    return sorted(set(intervalles))
''',
}

# ---------------- niveau 2 : analyse statique ----------------
INTERDITS = {"eval", "exec", "compile", "__import__"}
```

Exemple pas à pas

Exemple pas à pas (suite)

```
MODULES_INTERDITS = {"os", "subprocess", "socket", "shutil", "pickle", "ctypes"}
def analyse_statique(code):
    problemes = []
    try: arbre = ast.parse(code)
    except SyntaxError as e: return [f"syntaxe invalide : {e.msg}"]
    for n in ast.walk(arbre):
        modules = []
        if isinstance(n, ast.Import): modules = [a.name for a in n.names]
        elif isinstance(n, ast.ImportFrom) and n.module: modules = [n.module]
        for m in modules:
            racine = m.split(".")[0]
            if racine in MODULES_INTERDITS: problemes.append(f"module interdit : {
racine}")
            elif importlib.util.find_spec(racine) is None: problemes.append(f"modu
le INEXISTANT (hallucination probable) : {racine}")
        if isinstance(n, ast.Call) and isinstance(n.func, ast.Name) and n.func.id
in INTERDITS: problemes.append(f"appel interdit : {n.func.id}")
    return problemes
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------------- niveaux 3 à 5 : le harnais de test exécuté dans un PROCESSUS IS
   OLÉ ----------------
HARNAIS = textwrap.dedent('''
    import copy, json, random, sys
    exec(open(sys.argv[1]).read())
    def points(intervalles): return {x for a, b in intervalles for x in range(a, b
   )}                # segments unitaires [x, x+1]
    def reference(intervalles):                                    # implémentat
   ion évidente : trop simple pour être fausse
        pts = sorted(points(intervalles)); res = []
        for p in pts:
            if res and p == res[-1][1]: res[-1] = (res[-1][0], p + 1)
                        # segments contigus : les bornes qui se touchent fusionn
   ent
            else: res.append((p, p + 1))
        return res
    rapport = {"exemples": True, "proprietes": [], "differentiel": 0, "erreur": No
   ne}
    try:
```

Exemple pas à pas (suite)

```
        assert fusionner([(1, 3), (2, 6), (8, 10)]) == [(1, 6), (8, 10)]
        assert fusionner([]) == []
    except Exception as e: rapport["exemples"] = f"{type(e).__name__}: {e}"
    rng = random.Random(1)
    for _ in range(500):
        entree = [(a, a + rng.randint(1, 4)) for a in (rng.randint(0, 15) for _ in
    range(rng.randint(0, 6)))]
        copie = copy.deepcopy(entree)
        try:
            sortie = fusionner(entree)
        except Exception as e:
            rapport["erreur"] = f"{type(e).__name__} sur {copie}"; break
        if entree != copie and "entrée modifiée" not in rapport["proprietes"]: rap
port["proprietes"].append("entrée modifiée")
        if any(sortie[i][1] >= sortie[i + 1][0] for i in range(len(sortie) - 1)) a
nd "chevauchement résiduel" not in rapport["proprietes"]: rapport["proprietes"]
.append("chevauchement résiduel")
        if points(sortie) != points(copie) and "couverture différente" not in rapp
ort["proprietes"]: rapport["proprietes"].append("couverture différente")
```

Exemple pas à pas (suite)

```
        if sortie != reference(copie): rapport["differentiel"] += 1
    print(json.dumps(rapport))
''')

def verifier(nom, code, delai=5):
    verdict = {"candidat": nom, "statique": analyse_statique(code)}
    if verdict["statique"]:
        verdict["decision"] = "REJETÉ (analyse statique)"; return verdict
    import tempfile, os
    with tempfile.TemporaryDirectory() as d:
        open(os.path.join(d, "candidat.py"), "w").write(code); open(os.path.join(d
   , "harnais.py"), "w").write(HARNAIS)
        try:
            r = subprocess.run([sys.executable, "-I", os.path.join(d, "harnais.py"
   ), os.path.join(d, "candidat.py")], capture_output=True, text=True, timeout=del
   ai, cwd=d)
        except subprocess.TimeoutExpired:
            verdict["decision"] = "REJETÉ (délai dépassé)"; return verdict
    if r.returncode != 0:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        verdict["decision"] = "REJETÉ (plantage)"; verdict["detail"] = r.stderr.st
   rip().splitlines()[-1] if r.stderr.strip() else ""; return verdict
    rapport = json.loads(r.stdout.strip().splitlines()[-1]); verdict.update(rappor
   t)
    bons = rapport["exemples"] is True and not rapport["proprietes"] and rapport["
   differentiel"] == 0 and rapport["erreur"] is None
    verdict["decision"] = "ACCEPTÉ pour revue humaine" if bons else "REJETÉ"
    return verdict

print("candidat                    | décision                        | détails")
decisions = {}
for nom, code in CANDIDATS.items():
    v = verifier(nom, code); decisions[nom] = v["decision"]
    details = v["statique"] or [k for k in ("exemples", "proprietes", "differentie
   l", "erreur") if v.get(k) not in (True, [], 0, None)] or ""
    print(f"{nom:27s} | {v['decision']:31s} | {details}")
assert decisions["A_correct"].startswith("ACCEPTÉ") and decisions["B_bornes_qui_se
   _touchent"] == "REJETÉ" and decisions["C_dependance_inventee"].startswith("REJE
   TÉ (analyse")
```

Exemple pas à pas (suite)

```
assert decisions["D_modifie_l_entree"].startswith("REJETÉ") and decisions["E_appel
   _dangereux"].startswith("REJETÉ (analyse")
print("\ncandidats qui passent l'analyse statique :", [n for n, c in CANDIDATS.ite
   ms() if not analyse_statique(c)])
```

Lecture :

1. **A** est accepté **pour revue humaine** — jamais directement intégré : le vérificateur élimine le mauvais, il ne démontre pas le bon.
2. **B** passe l'analyse statique et la plupart des exemples, mais échoue aux **propriétés** et au test différentiel : il ne fusionne pas les intervalles qui se touchent à la borne (l'erreur de HashBox).
3. **C** importe une bibliothèque **qui n'existe pas** : l'analyse statique détecte la **hallucination** avant toute exécution.
4. **D** passe sur certaines entrées mais **modifie la liste d'entrée** et plante sur la liste vide : détecté par les propriétés et l'exécution.

Exemple pas à pas (suite)

5. **E** contient un appel système : refusé avant exécution (jamais lancé).

Chaque contrôle est **indépendant** du générateur : exemples et référence écrits par un humain, outils déterministes, exécution isolée avec délai.

À toi, avec aide

À toi, avec aide

Voici une fonction « produite par une IA » :

```
def mediane(valeurs):
    valeurs.sort()
    n = len(valeurs)
    return valeurs[n // 2]
```

1. Liste au moins quatre défauts ou risques (cas limites, effet de bord, définition de la médiane…).
2. Écris les propriétés que tu vérifierais (au moins quatre).
3. Écris une implémentation de référence simple (par exemple via `statistics.median`) et un test différentiel sur 1 000 entrées aléatoires.
4. Dis quels défauts le test différentiel détecte et lesquels il faut chercher autrement.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour une liste de taille paire, la médiane est la moyenne des deux valeurs centrales.

Indice 2

`valeurs.sort()` modifie la liste de l'appelant ; liste vide ?

Indice 3

Compare à `statistics.median` sur des listes d'entiers de tailles variées.

À toi, avec aide · réponse

Réponse

```
import copy, random, statistics

def mediane(valeurs):
    valeurs.sort()
    n = len(valeurs)
    return valeurs[n // 2]

rng = random.Random(1)
echecs = {"liste vide": 0, "entrée modifiée": 0, "différentiel": 0, "borne": 0}
for _ in range(1000):
    l = [rng.randint(-5, 5) for _ in range(rng.randint(1, 7))]; copie = copy.deepc
   opy(l)
    r = mediane(l)
    if l != copie: echecs["entrée modifiée"] += 1
    if r != statistics.median(copie): echecs["différentiel"] += 1
    if not min(copie) <= r <= max(copie): echecs["borne"] += 1
try: mediane([]); except_vide = False
except IndexError: except_vide = True
print(echecs, "| liste vide lève IndexError :", except_vide)
```

Réponse (suite)

```
assert echecs["entrée modifiée"] > 500 and echecs["différentiel"] > 100 and echecs
   ["borne"] == 0
```

1. Défauts : **modifie l'entrée** (`sort` en place) ; **liste paire** (renvoie la valeur haute au lieu de la moyenne des deux centrales) ; **liste vide** (IndexError sans message clair) ; types non numériques non gérés ; liste d'objets non comparables.
2. Propriétés : la médiane est comprise entre le minimum et le maximum ; elle est invariante par permutation de l'entrée ; l'entrée n'est pas modifiée ; ajouter un élément très grand ne baisse jamais la médiane ; médiane d'une liste constante = la constante.
3. Le test différentiel contre `statistics.median` révèle plus de 100 divergences sur 1 000 entrées (toutes les listes paires dont les deux valeurs centrales diffèrent).

Réponse (suite)

4. **Détectés** par le différentiel : le défaut des listes paires. **À chercher autrement** : l'effet de bord (propriété « entrée non modifiée »), la gestion de la liste vide (test d'erreur et décision de spécification : lever une erreur claire ou renvoyer `None` ?) — ce sont des **choix de spécification** qu'aucune référence ne tranche.

À toi, en solo

Vérificateur de références. Un assistant a produit un rapport de 8 références (titre, auteur, année, identifiant) et 6 affirmations chiffrées citant ces références. Tu disposes d'une « base documentaire » locale (dictionnaire) de vraies références avec leur contenu.

1. écris `verifier_references(rapport, base)` : pour chaque référence, **existence** (identifiant dans la base), **cohérence des métadonnées** (titre/auteur/année), pour chaque affirmation chiffrée : le **chiffre cité** apparaît-il dans le contenu de la source citée ?
2. classe chaque élément : `vérifié`, `référence inexistante`, `métadonnées incohérentes`, `chiffre non retrouvé`, `source ne soutient pas l'affirmation` ;
3. construis un rapport simulé avec des erreurs injectées (2 références inventées, 1 année fausse, 2 chiffres déformés, 1 affirmation attribuée à la mauvaise source) ;
4. calcule le rappel de ton vérificateur sur les erreurs injectées et le nombre de faux avertissements ;

À toi, en solo (suite)

5. explique ce qu'un vérificateur automatique ne peut pas faire (juger si la source soutient *réellement* l'affirmation nuancée).

Pistes de correction

```
import re

BASE = {
 "R1": dict(titre="Taux de complétion des cours en ligne", auteur="Martin", annee=
   2022, contenu="Le taux de complétion moyen est de 12 % pour les cours en ligne
   ouverts à tous."),
 "R2": dict(titre="Effet des rappels sur la rétention", auteur="Durand", annee=202
   3, contenu="Un rappel par e-mail augmente la reprise de 3 points, de 12 % à 15
   %."),
 "R3": dict(titre="Pédagogie par projets", auteur="Leroy", annee=2021, contenu="Le
   s apprenants en projet retiennent 40 % de contenu en plus après un mois."),
 "R4": dict(titre="Évaluation automatique du code", auteur="Petit", annee=2020, co
   ntenu="Les tests automatiques détectent 85 % des erreurs de logique simples."),
}
REFERENCES = [("R1", "Taux de complétion des cours en ligne", "Martin", 2022), ("R
   2", "Effet des rappels sur la rétention", "Durand", 2021), ("R9", "Les cours du
    futur", "Inconnu", 2024),
              ("R3", "Pédagogie par projets", "Leroy", 2021), ("R4", "Évaluation a
   utomatique du code", "Petit", 2020), ("R12", "Intelligence collective", "Nom",
```

À toi, en solo · réponse

Pistes de correction (suite)

```
   2023)]
AFFIRMATIONS = [("Le taux de complétion moyen est de 12 %.", "R1"), ("Un rappel au
   gmente la reprise de 3 points.", "R2"), ("Un rappel augmente la reprise de 8 po
   ints.", "R2"),
                ("Les apprenants en projet retiennent 40 % de contenu en plus.", "
   R3"), ("Les tests automatiques détectent 95 % des erreurs.", "R4"), ("La pédago
   gie par projets double la rétention.", "R4")]
def nombres(t): return set(re.findall(r"\d+(?:[,.]\d+)?", t))
def verifier_references(refs, affirmations, base):
    constats = []
    for id_, titre, auteur, annee in refs:
        if id_ not in base: constats.append((id_, "référence inexistante")); conti
   nue
        b = base[id_]
        constats.append((id_, "vérifié" if (b["titre"], b["auteur"], b["annee"]) =
   = (titre, auteur, annee) else "métadonnées incohérentes"))
    for texte, source in affirmations:
        if source not in base: constats.append((texte, "référence inexistante"));
   continue
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        manquants = nombres(texte) - nombres(base[source]["contenu"])
        constats.append((texte, "vérifié" if not manquants else f"chiffre non retr
   ouvé : {sorted(manquants)}"))
        if not manquants and not (set(re.findall(r"\w{5,}", texte.lower())) & set(
   re.findall(r"\w{5,}", base[source]["contenu"].lower()))): constats[-1] = (texte
   , "source ne soutient pas l'affirmation")
    return constats
for c in verifier_references(REFERENCES, AFFIRMATIONS, BASE): print(f"  {c[1]:38s}
    ← {str(c[0])[:60]}")
res = dict(verifier_references(REFERENCES, AFFIRMATIONS, BASE))
injectees = {"R9": "référence inexistante", "R12": "référence inexistante", "R2":
   "métadonnées incohérentes", "Un rappel augmente la reprise de 8 points.": "chif
   fre non retrouvé", "Les tests automatiques détectent 95 % des erreurs.": "chiff
   re non retrouvé", "La pédagogie par projets double la rétention.": "source ne s
   outient pas l'affirmation"}
trouvees = sum(1 for k, v in injectees.items() if res.get(k, "").startswith(v))
print(f"erreurs injectées détectées : {trouvees}/{len(injectees)}")
assert trouvees == len(injectees)
```

Pistes de correction (suite)

Le vérificateur attrape l'**existence**, la **cohérence des métadonnées** et les **chiffres déformés**. Il ne sait pas dire si une source *nuancée* soutient réellement une affirmation (« augmente la reprise **dans certains cas** » contre « augmente la reprise »). Cette relecture-là reste **humaine** — c'est la partie qui coûte, donc celle à réserver aux affirmations importantes.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Laisser l'IA écrire le code ET ses tests	Les mêmes angles morts dans les deux	Spécification et oracle écrits par un humain, tests indépendants
Faire confiance aux exemples fournis	Trois cas qui passent ne prouvent rien	Propriétés aléatoires et test différentiel
Ne pas vérifier que les dépendances existent	Imports inventés, API périmées	Analyse statique et exécution réelle
Exécuter le code généré sans isolation	Effets de bord, exfiltration	Processus séparé, délai, droits minimaux

Mini-boss

Mini-boss

Protocole de vérification d'une contribution générée par IA. Définis et implémente un protocole complet pour accepter ou rejeter un correctif proposé par un assistant dans HashBox :

1. une **spécification** écrite (3 comportements, 3 cas limites, 2 erreurs) avant toute génération ;
2. un **vérificateur** automatique : analyse statique (imports existants, appels interdits, secrets, complexité cyclomatique ≤ seuil), exécution isolée avec délai et mémoire limitée, tests d'exemples, propriétés (5), test différentiel contre une référence, **mutation** de la suite (au moins 15 mutants, score ≥ 85 %) ;
3. 8 candidats simulés (2 corrects, 6 défectueux de façons différentes dont une dépendance inventée, un effet de bord, une faille d'injection, une fuite de secret, une lenteur quadratique, un cas limite) ;
4. un **rapport** par candidat avec la raison précise de chaque rejet ;
5. une **liste de contrôle de revue humaine** (10 questions, dont « puis-je expliquer chaque ligne ? ») ;

Mini-boss

Mini-boss (suite)

6. la **fiche de traçabilité** (prompt, modèle, version, date, vérifications passées, relecteur) ;
7. les **limites** du protocole : ce qu'il ne détecte pas, et à quel coût humain.

Mini-boss · réponse

Pistes de correction

Un bon protocole : spécification d'abord, vérificateur multi-niveaux qui rejette les 6 défectueux pour de bonnes raisons et ne laisse que les 2 corrects en revue humaine, score de mutation de la suite, liste de revue, fiche de traçabilité, et section de limites honnête (par exemple « ne détecte pas une erreur de spécification, ni une faille logique dépendant du contexte métier »).

Quiz éclair

Quiz éclair

1. Pourquoi les tests écrits par l'IA pour son propre code sont-ils insuffisants ?

A. ils sont toujours trop nombreux
B. ils ne s'exécutent pas
C. ils partagent ses angles morts : même fausse hypothèse dans le code et le test

2. Un test de propriétés :

A. vérifie un invariant sur de nombreuses entrées aléatoires
B. vérifie un seul exemple choisi
C. remplace la spécification

3. Une dépendance qui n'existe pas dans le code généré est détectée par :

A. la température du modèle

Quiz éclair

Quiz éclair (suite)

B. l'analyse statique : vérifier que les modules importés existent
C. la longueur du code

4. 🔁 *Révision mission 07* : Pour comparer deux systèmes, on évalue de préférence :

A. chacun sur des cas différents
B. un seul exemple chacun
C. les deux sur les mêmes cas, en regardant la différence et son intervalle

5. 🔁 *Révision mission 07* : Le kappa de Cohen mesure :

A. l'accord entre annotateurs au-delà du hasard
B. la vitesse du modèle
C. la taille du jeu de test

Quiz éclair · réponse

Réponses

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

Je réfléchis

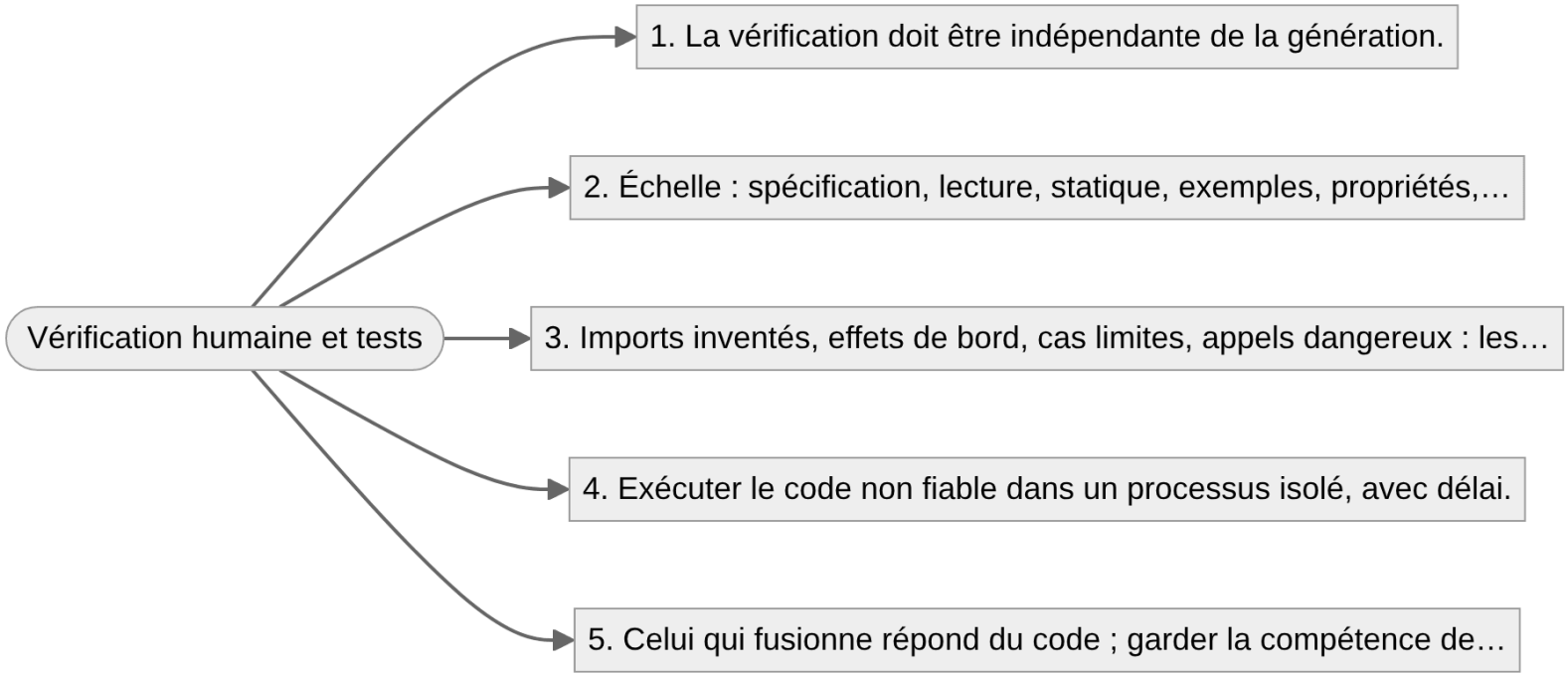
1. Quel code généré as-tu déjà intégré sans pouvoir l'expliquer ligne à ligne ?
2. Pour quelle partie de ton travail la vérification coûte-t-elle plus que la production ?

À retenir

1. La vérification doit être indépendante de la génération.
2. Échelle : spécification, lecture, statique, exemples, propriétés, différentiel, mutation, revue.
3. Imports inventés, effets de bord, cas limites, appels dangereux : les défauts typiques.
4. Exécuter le code non fiable dans un processus isolé, avec délai.
5. Celui qui fusionne répond du code ; garder la compétence de comprendre.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Vérificateur impitoyable

Prochaine mission : Sécurité, confidentialité et gouvernance

