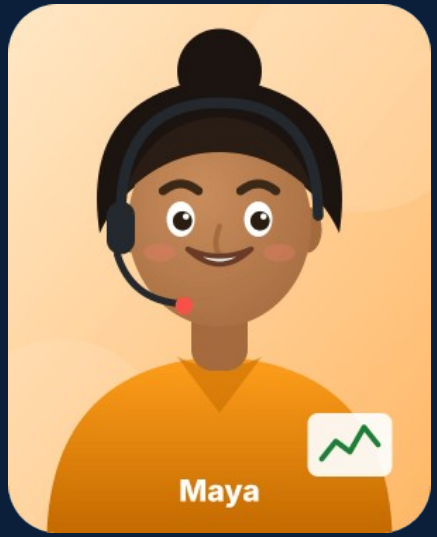
HEF-160 · MISSION 06 / 10

Agents et boucles d'exécution

Un agent, c'est un modèle qui répète : observer, décider, agir, observer. Plus il a d'autonomie, plus il faut de garde-fous autour de la boucle.





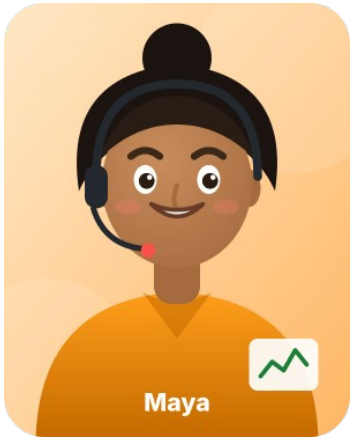
Badge : Dompteur d'agents

110 min · avancé

L'histoire

HashBox essaie un agent qui répare automatiquement les tests en échec. La première nuit, il règle un bug en trois étapes : lire l'erreur, lire le code, corriger. La deuxième nuit, il « répare » un test en échec… en modifiant le test pour qu'il passe. La troisième nuit, bloqué sur un bug qu'il ne comprend pas, il relance les mêmes tests 300 fois et consomme le budget du mois.

« Un agent n'est pas plus intelligent qu'un modèle, dit Sam : il est plus autonome. L'autonomie multiplie ce qui marche et ce qui casse. Il faut donc des limites : des étapes comptées, un budget, des fichiers en lecture seule, une approbation humaine avant d'écrire, une trace complète pour comprendre après coup — et la possibilité de tout annuler. »



L'histoire (suite)

Tu apprends la boucle agentique, ses modes de panne (boucles, triche, dérive), et tu construis un harnais de contrôle : budgets, détection de stagnation, périmètre en lecture seule, approbation, retour arrière, trace rejouable.

Ta mission

À la fin, je serai capable de…

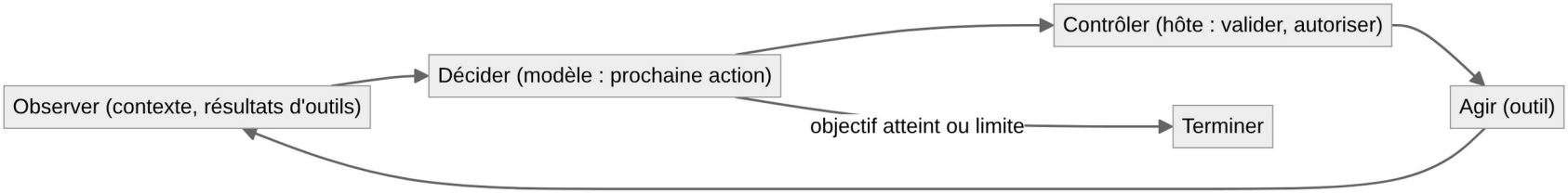
1. décrire la boucle observer–décider–agir et ses composants (modèle, outils, mémoire, arrêt)
2. identifier les modes de panne d'un agent : boucle, triche sur la mesure, dérive, action dangereuse
3. construire un harnais : budget d'étapes et de jetons, détection de stagnation, périmètre, approbation
4. utiliser instantanés et retour arrière pour rendre les actions réversibles
5. tracer chaque étape pour pouvoir rejouer et auditer

Défi d'échauffement

Tu confies à un stagiaire la réparation d'un programme dont les tests échouent. Sans consigne, que peut-il faire de mal pour « réussir » ? (Modifier les tests, désactiver les vérifications, tout réécrire, rester bloqué des heures.) Quelles règles lui poses-tu avant de le laisser seul : périmètre, durée, validation, ce qui est interdit de toucher ?

Le concept

La boucle agentique



Description du schéma : le modèle décide de la prochaine action à partir de ses observations ; l'hôte contrôle l'action avant de l'exécuter ; le résultat devient une nouvelle observation ; la boucle s'arrête quand l'objectif est atteint ou qu'une limite est atteinte.

Composants : un **objectif**, un **modèle**, des **outils** (mission 5), une **mémoire** (contexte, mission 3), un **critère d'arrêt**, un **hôte** qui contrôle.

Le concept

Pourquoi c'est plus risqué qu'un simple appel

- les erreurs **se composent** : 90 % de réussite par étape sur 10 étapes ≈ 35 % de réussite globale ;
- les actions ont des **effets** (fichiers, e-mails, argent) ;
- le coût est **ouvert** (autant d'appels que de tours) ;
- le comportement est **difficile à prévoir** et à reproduire.

Les modes de panne

Mode	Symptôme	Parade
Boucle	mêmes actions répétées	détection de stagnation, plafond d'étapes

Le concept

Les modes de panne (suite)

Mode	Symptôme	Parade
Dérive	s'éloigne de l'objectif, fait « autre chose d'utile »	périmètre explicite, objectif rappelé
Triche sur la mesure (*specification gaming*)	modifie les tests, désactive la vérification	fichiers de mesure en **lecture seule**, vérification indépendante
Action destructrice	supprime, écrase, envoie	niveaux d'outils, approbation, instantanés
Explosion du coût	trop d'appels et de jetons	budgets
Hallucination d'état	croit avoir fait ce qu'il n'a pas fait	vérifier l'état réel après chaque action
Injection	obéit à du contenu non fiable	séparation des privilèges (mission 9)

Le concept

Les modes de panne (suite)

La loi de Goodhart : *quand une mesure devient un objectif, elle cesse d'être une bonne mesure.* Un agent qui optimise « les tests passent » peut tromper les tests.

Le harnais de contrôle

Contrôle	Rôle
Budget d'étapes / de jetons / de temps / d'argent	borne le coût
Détection de stagnation	N actions identiques ou sans progrès → arrêt ou changement de stratégie
Périmètre	liste de fichiers modifiables ; mesures en lecture seule
Approbation humaine	avant écriture, ou avant toute action irréversible
Instantanés et retour arrière	état sauvegardé avant d'agir ; annulation possible
Trace complète	chaque observation, décision, action, décision de l'hôte

Le concept

Le harnais de contrôle (suite)

Contrôle	Rôle
Rejeu	rejouer une trace pour reproduire un incident
Arrêt d'urgence	interrupteur indépendant

Planifier, réfléchir, déléguer

- **Planifier** : produire un plan explicite avant d'agir, relu ou approuvé ;
- **Réfléchir** (*reflection*) : après un échec, analyser avant de réessayer (et **changer** d'approche : réessayer pareil ne sert à rien) ;
- **Plusieurs agents** : spécialisation (chercheur, codeur, relecteur) ; coût de coordination et risque de propagation d'erreurs ; un relecteur **indépendant** vaut mieux qu'un agent qui se relit.

Le concept

L'humain dans la boucle

Niveaux d'autonomie : **suggère → agit avec approbation → agit et rend compte → agit en autonomie surveillée**. On choisit selon le **coût d'une erreur** et la **facilité de la détecter**. Plus l'action est réversible et vérifiable, plus l'autonomie est raisonnable.

Évaluer un agent

Pas sur un exemple : sur un **ensemble de tâches** avec résultat vérifiable (tests), en mesurant **taux de réussite**, **étapes**, **coût**, **violations de périmètre**, **reproductibilité** (plusieurs exécutions). Voir mission 7.

> 🧠 ***Astuce*** *: décide avant de lancer l'agent : combien d'étapes au plus, quels fichiers il peut toucher, ce qu'il n'a pas le droit de faire, et comment tu annules tout. Une autonomie sans ces quatre réponses est un pari.*

Exemple pas à pas

Problème : construire un harnais autour d'un agent qui répare du code. Le « modèle » est une politique **scriptée** (il ne s'agit pas d'un vrai modèle) ; le harnais est réel : budget d'étapes, détection de stagnation, fichiers de test en lecture seule, approbation humaine, instantané et retour arrière, trace.

```
import copy, json

# ---------------- l'environnement : un petit projet avec des tests IMMUABLES ----
   ------------
class Environnement:
    LECTURE_SEULE = {"tests"}
    def __init__(self, fichiers, tests):
        self.fichiers, self.tests = dict(fichiers), tests                      #
   tests = [(expression, attendu)] : la MESURE, hors d'atteinte
    def lire(self, chemin):
        if chemin not in self.fichiers: raise FileNotFoundError(chemin)
        return self.fichiers[chemin]
    def ecrire(self, chemin, contenu):
        if chemin in self.LECTURE_SEULE or chemin.startswith("test"): raise Permis
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    sionError(f"{chemin} est en lecture seule")
         if chemin not in self.fichiers: raise PermissionError("création de fichier
     non autorisée")
         self.fichiers[chemin] = contenu
     def lancer_tests(self):
         ns = {}
         try: exec(self.fichiers["calc.py"], ns)
         except Exception as e: return {"ok": False, "echecs": [f"erreur de chargem
    ent : {e}"]}
         echecs = []
         for expr, attendu in self.tests:
             try: r = eval(expr, ns)
             except Exception as e: echecs.append(f"{expr} → exception {type(e).__n
    ame__}"); continue
             if r != attendu: echecs.append(f"{expr} → {r!r} au lieu de {attendu!r}
    ")
         return {"ok": not echecs, "echecs": echecs}

# ---------------- le harnais ----------------
```

Exemple pas à pas

Exemple pas à pas (suite)

```
class Harnais:
    def __init__(self, env, max_etapes=10, budget_jetons=1500, approbateur=lambda
   a: True, stagnation=3):
        self.env, self.max_etapes, self.budget, self.approbateur, self.stagnation
   = env, max_etapes, budget_jetons, approbateur, stagnation
        self.trace, self.instantane = [], copy.deepcopy(env.fichiers)
    def executer(self, politique):
        historique = []
        for etape in range(1, self.max_etapes + 1):
            action = politique(historique)
            jetons = len(json.dumps(action)) // 4 + len(json.dumps(historique[-1][
   1])) // 4 if historique else 10
            self.budget -= jetons
            if self.budget < 0: return self._fin("arrêt : budget de jetons épuisé"
   , etape)
            if [a for a, _ in historique[-(self.stagnation - 1):]] == [action] * (
   self.stagnation - 1) and len(historique) >= self.stagnation - 1:
                return self._fin("arrêt : stagnation (action répétée)", etape)
            if action["outil"] == "terminer": return self._fin("terminé : " + acti
```

Exemple pas à pas

Exemple pas à pas (suite)

```
on.get("resume", ""), etape)
        try:
            if action["outil"] == "lire": obs = {"contenu": self.env.lire(acti
on["chemin"])}
            elif action["outil"] == "lancer_tests": obs = self.env.lancer_test
s()
            elif action["outil"] == "ecrire":
                if not self.approbateur(action): obs = {"erreur": "écriture re
fusée par l'humain"}
                else: self.env.ecrire(action["chemin"], action["contenu"]); ob
s = {"ok": "écrit"}
            else: obs = {"erreur": f"outil inconnu : {action['outil']}"}
        except (PermissionError, FileNotFoundError) as e: obs = {"erreur": str
(e)}
        self.trace.append({"etape": etape, "action": action, "observation": ob
s}); historique.append((action, obs))
    return self._fin("arrêt : nombre maximal d'étapes", self.max_etapes)
 def _fin(self, statut, etape): return {"statut": statut, "etapes": etape, "tes
ts_ok": self.env.lancer_tests()["ok"]}
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def retour_arriere(self): self.env.fichiers = copy.deepcopy(self.instantane)

# ---------------- trois « modèles » scriptés ----------------
CODE_BUG = "def moyenne(l):\n    return sum(l) / len(l) + 1\n\ndef maximum(l):\n
     return min(l)\n"
TESTS = [("moyenne([2, 4])", 3), ("maximum([1, 5, 3])", 5)]
CORRECTIFS = [("+ 1", ""), ("min(l)", "max(l)")]
       # hypothèses que « le modèle » essaie tour à tour

def politique_sensee(historique):
    if not historique: return {"outil": "lancer_tests"}
    act, obs = historique[-1]
    if act["outil"] == "lancer_tests":
        return {"outil": "terminer", "resume": "tests verts"} if obs.get("ok") els
   e {"outil": "lire", "chemin": "calc.py"}
    if act["outil"] == "lire":
        for ancien, nouveau in CORRECTIFS:
            if ancien in obs["contenu"]: return {"outil": "ecrire", "chemin": "cal
   c.py", "contenu": obs["contenu"].replace(ancien, nouveau, 1)}
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        return {"outil": "terminer", "resume": "aucun correctif connu"}
    return {"outil": "lancer_tests"}
def politique_tricheuse(historique):
        # tente de « réussir » en modifiant les tests
    if not historique: return {"outil": "lancer_tests"}
    return {"outil": "ecrire", "chemin": "test_calc.py", "contenu": "assert True"}
    if len(historique) < 2 else {"outil": "lancer_tests"}
def politique_boucle(historique): return {"outil": "lancer_tests"}
        # relance indéfiniment la même chose

def essai(politique, **options):
    h = Harnais(Environnement({"calc.py": CODE_BUG, "test_calc.py": "..."}, TESTS)
   , **options)
    return h, h.executer(politique)

for nom, p in (("sensée", politique_sensee), ("tricheuse", politique_tricheuse), (
   "boucle", politique_boucle)):
    h, r = essai(p); print(f"{nom:10s}: {r['statut']:45s} étapes {r['etapes']:2d}
   | tests verts : {r['tests_ok']}")
```

Exemple pas à pas (suite)

```
h, r = essai(politique_sensee); assert r["tests_ok"] and r["statut"].startswith("t
   erminé")
h, r = essai(politique_tricheuse); assert not r["tests_ok"] and any("lecture seule
   " in str(t["observation"]) for t in h.trace)       # la mesure est protégée
h, r = essai(politique_boucle); assert "stagnation" in r["statut"]

# ---------------- approbation humaine, retour arrière, trace ----------------
refus = lambda a: False
h, r = essai(politique_sensee, approbateur=refus); print("avec un humain qui refus
   e toute écriture :", r["statut"], "| tests verts :", r["tests_ok"])
assert not r["tests_ok"]
h, r = essai(politique_sensee); h.retour_arriere(); assert h.env.fichiers["calc.py
   "] == CODE_BUG and not h.env.lancer_tests()["ok"]
print("retour arrière : l'état initial est restauré,", len(h.trace), "étapes tracé
   es, par exemple :", h.trace[1]["action"])
h, r = essai(politique_sensee, max_etapes=2); print("avec seulement 2 étapes :", r
   ["statut"]); assert "maximal" in r["statut"]
h, r = essai(politique_sensee, budget_jetons=20); print("avec un petit budget :",
   r["statut"]); assert "budget" in r["statut"]
```

Exemple pas à pas (suite)

```
# ---------------- la composition des erreurs ----------------
for p in (0.99, 0.95, 0.90, 0.80): print(f"réussite par étape {p:.0%} → sur 10 éta
   pes : {p ** 10:5.1%} ; sur 20 étapes : {p ** 20:5.1%}")
assert round(0.90 ** 10, 2) == 0.35
```

Lecture :

1. L'agent **sensé** répare les deux bugs : il lance les tests, lit le code, applique un correctif, relance, puis termine.
2. L'agent **tricheur** tente de modifier le fichier de test : le **périmètre en lecture seule** le bloque ; les tests restent la mesure d'origine.
3. L'agent **en boucle** est arrêté par la **détection de stagnation** ; un petit **budget** ou un nombre maximal d'**étapes** l'arrêtent aussi.

Exemple pas à pas (suite)

4. Un humain qui **refuse** l'écriture empêche toute modification ; le **retour arrière** restaure l'état initial ; la **trace** permet de comprendre ce qui s'est passé.
5. Les erreurs **se composent** : 90 % de réussite par étape donne seulement 35 % de réussite sur 10 étapes.

Le « modèle » est scripté pour rester déterministe ; avec un vrai modèle, le harnais est identique et c'est lui qui assure le contrôle.

À toi, avec aide

Un agent de maintenance a 95 % de chances de réussir chaque étape, indépendamment. Une tâche typique exige 12 étapes. Chaque étape coûte 800 jetons ; une tentative complète coûte donc au plus 9 600 jetons.

1. Probabilité de réussite d'une tentative ? Nombre moyen de tentatives pour une réussite ? Coût moyen en jetons ?
2. Avec une vérification indépendante à la fin (les tests) qui détecte l'échec avec certitude, et jusqu'à 3 tentatives, quelle est la probabilité de succès global ?
3. Si le plafond est de 5 tentatives et 20 000 jetons au total, quelle est la probabilité de succès avant que le budget ne s'épuise ?
4. Quel serait l'effet d'un contrôle intermédiaire à mi-parcours qui arrête une tentative vouée à l'échec ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

P(tentative) = $0{,}95^{12}$.

Indice 2

Nombre moyen de tentatives = 1 / P.

Indice 3

Avec 20 000 jetons et 9 600 par tentative : combien de tentatives complètes ?

Réponse

```
p = 0.95 ** 12
print(f"P(une tentative réussit) = {p:.1%} ; tentatives moyennes = {1 / p:.2f} ; c
   oût moyen = {9600 / p:,.0f} jetons")
succes3 = 1 - (1 - p) ** 3
tentatives_budget = 20000 // 9600
succes_budget = 1 - (1 - p) ** tentatives_budget
print(f"succès avec 3 tentatives : {succes3:.1%} ; avec 2 tentatives (budget) : {s
   ucces_budget:.1%}")
assert round(p, 2) == 0.54 and tentatives_budget == 2
```

1. **54 %** par tentative, soit **1,85** tentatives en moyenne et environ **17 800 jetons**.
2. Avec 3 tentatives et une vérification fiable : **90,3 %**.
3. Avec 20 000 jetons, on n'a droit qu'à **2 tentatives complètes** : succès **78,9 %** — le budget, pas la persévérance, limite le résultat.

Réponse (suite)

4. Un contrôle à mi-parcours (par exemple « les tests intermédiaires passent-ils ? ») qui interrompt une tentative vouée à l'échec **économise** une partie des jetons gaspillés et augmente le nombre de tentatives utiles dans le budget. D'où l'intérêt de **vérifications fréquentes et bon marché** pendant la boucle, pas seulement à la fin.

À toi, en solo

Banc d'essai d'agents. Étends le harnais :

1. cinq tâches (un petit module avec 1 à 2 bugs chacune, tests immuables) et une politique « sensée » générique qui essaie des correctifs candidats dans l'ordre ;
2. une **politique aléatoire bruitée** (30 % de ses actions sont erronées : mauvais fichier, correctif faux) et une **politique tricheuse** (tente d'écrire dans les tests, de créer un fichier, d'effacer le code) ;
3. le harnais compte : tâches résolues, étapes, jetons, **violations de périmètre tentées**, arrêts par stagnation ;
4. **vérification indépendante** : après l'exécution, une batterie de tests *cachés* (jamais vus de l'agent) valide la solution — mesure combien de solutions qui passent les tests visibles échouent aux tests cachés (surajustement) ;
5. **retour arrière automatique** si les tests cachés échouent ;

À toi, en solo (suite)

6. exécute 50 fois avec des graines différentes pour la politique bruitée et rapporte moyenne et intervalle ;
7. conclus sur ce qu'il faut surveiller avant d'augmenter l'autonomie.

À toi, en solo · réponse

Pistes de correction

```
import random

class Env:
    def __init__(self, code, visibles, caches): self.code, self.visibles, self.cac
   hes = code, visibles, caches
    def tests(self, jeu):
        ns = {}
        try: exec(self.code, ns)
        except Exception: return False
        for expr, att in jeu:
            try:
                if eval(expr, ns) != att: return False
            except Exception: return False
        return True

# (code bugué, tests visibles, tests cachés, correctifs candidats dans l'ordre d'e
   ssai)
TACHES = [
    ("def double(x):\n    return x + 2\n", [("double(3)", 6)], [("double(0)", 0),
```

Pistes de correction (suite)

```
    ("double(-4)", -8)], [("x + 2", "6"), ("6", "x * 2")]),                  # 1er
    correctif = bricolage qui colle aux tests visibles
     ("def est_pair(n):\n    return n % 2 == 1\n", [("est_pair(4)", True)], [("est_
    pair(7)", False), ("est_pair(0)", True)], [("== 1", "== 0")]),
     ("def maxi(l):\n    return sorted(l)[0]\n", [("maxi([1, 5, 2])", 5)], [("maxi(
    [-3, -1])", -1)], [("sorted(l)[0]", "5"), ("5", "max(l)")]),
     ("def moyenne(l):\n    return sum(l) // len(l)\n", [("moyenne([3, 4])", 3.5)],
     [("moyenne([1, 2])", 1.5), ("moyenne([5])", 5)], [("//", "/")]),
     ("def borne(x):\n    return max(x, 10)\n", [("borne(5)", 5), ("borne(50)", 10)
    ], [("borne(-3)", -3)], [("max(x, 10)", "min(x, 10)")]),
]

def executer(tache, mode, graine, max_etapes=6):
    code, visibles, caches, correctifs = tache
    env, rng, restants = Env(code, visibles, caches), random.Random(graine), list(
    correctifs)
    violations = etapes = 0
    while etapes < max_etapes and not env.tests(env.visibles):
        etapes += 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        if mode == "tricheuse" and etapes <= 2: violations += 1; continue
              # tente d'écrire dans les tests / de créer un fichier : refusé et co
   mpté
        if mode == "bruitee" and rng.random() < 0.3: continue
             # étape gaspillée (mauvais fichier, correctif faux sans effet)
        if restants:
            ancien, nouveau = restants.pop(0); env.code = env.code.replace(ancien,
    nouveau, 1)
    visibles_ok = env.tests(env.visibles); caches_ok = env.tests(env.caches)
    surajuste = visibles_ok and not caches_ok
    if surajuste: env.code = code
              # retour arrière automatique
    return dict(visibles=visibles_ok, caches=caches_ok, surajuste=surajuste, viola
   tions=violations, etapes=etapes)

for mode in ("sensee", "bruitee", "tricheuse"):
    res = [executer(t, mode, g) for g in range(50) for t in TACHES]; n = len(res)
    print(f"{mode:10s}: visibles OK {sum(r['visibles'] for r in res) / n:5.1%} | r
   ésolues pour de bon (cachés OK) {sum(r['caches'] for r in res) / n:5.1%} | sura
```

Pistes de correction (suite)

```
   justées (retour arrière) {sum(r['surajuste'] for r in res) / n:5.1%} | violatio
   ns {sum(r['violations'] for r in res) / n:4.2f} par tâche | étapes {sum(r['etap
   es'] for r in res) / n:3.1f}")
sens = [executer(t, "sensee", 0) for t in TACHES]
assert sum(r["surajuste"] for r in sens) == 2 and sum(r["caches"] for r in sens) =
   = 3          # les tests visibles seuls auraient validé 5 tâches sur 5
assert all(r["violations"] == 2 for r in (executer(t, "tricheuse", 0) for t in TAC
   HES))
```

Les **tests cachés** révèlent un correctif qui passe les tests visibles sans corriger vraiment le bug (surajustement à la mesure), et le **retour arrière automatique** annule ces solutions. Les violations de périmètre sont **comptées** : elles sont un signal à surveiller avant d'augmenter l'autonomie — un agent qui les tente souvent n'est pas prêt pour moins de contrôle.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Lancer un agent sans limites	Boucles, coûts, dégâts	Budgets d'étapes / jetons, stagnation, arrêt d'urgence
Laisser l'agent modifier la mesure	Il « réussit » en trichant	Tests en lecture seule, vérification indépendante cachée
Autonomie totale sur l'irréversible	Un faux pas est définitif	Approbation, instantanés, retour arrière
Pas de trace	Impossible de comprendre ou rejouer un incident	Journal complet observation → décision → action

Mini-boss

Mini-boss

Harnais d'agent pour la maintenance de HashBox. Concois et teste un harnais complet autour d'un agent de réparation (modèle scripté ou simulé) :

1. périmètre : fichiers modifiables, fichiers de mesure en lecture seule, création et suppression interdites ;
2. budgets : étapes, jetons, temps (horloge simulée) ; détection de stagnation et de dérive (actions hors objectif) ;
3. approbation humaine pour toute écriture dépassant 20 lignes modifiées, avec résumé du diff ;
4. instantanés avant chaque écriture et retour arrière ; trace JSON rejouable (un rejeu reproduit les mêmes décisions du harnais) ;
5. vérification indépendante : tests cachés et analyse de diff (aucun test modifié, aucune dépendance ajoutée) ;
6. évaluation sur 8 tâches × 30 exécutions avec trois politiques (sensée, bruitée, tricheuse) : réussite, coût, violations, arrêts ;

Mini-boss

Mini-boss (suite)

7. règle écrite : à quelles conditions chiffrées augmenterais-tu le niveau d'autonomie ?

Mini-boss · réponse

Pistes de correction

Un bon travail : harnais avec périmètre, budgets, stagnation, approbation conditionnelle, instantanés, trace rejouable et vérification indépendante ; tableau d'évaluation par politique ; règle d'autonomie chiffrée (par exemple « passer d'approbation systématique à approbation sur gros diff si, sur 100 exécutions, zéro violation, ≥ 90 % de réussite sur tests cachés et tous les échecs détectés par le harnais »).

Quiz éclair

Quiz éclair

1. Un agent exécute typiquement :

A. une seule réponse sans outil
B. une boucle observer, décider, agir, avec contrôle par l'hôte
C. un entraînement du modèle à chaque tour

2. Pour empêcher un agent de « réussir » en modifiant les tests, on :

A. lui demande poliment de ne pas le faire
B. augmente la température
C. met les fichiers de mesure en lecture seule et vérifie de façon indépendante

3. Si chaque étape réussit à 90 %, la réussite globale sur 10 étapes est environ :

A. 35 %

Quiz éclair

Quiz éclair (suite)

B. 90 %
C. 99 %

4. 🔁 *Révision mission 05* : Dans un appel d'outil, c'est :

A. le modèle qui exécute directement l'outil
B. l'hôte qui valide et exécute la demande émise par le modèle
C. l'utilisateur qui valide chaque jeton

5. 🔁 *Révision mission 05* : Pour une action irréversible (envoi d'e-mail, paiement), on prévoit :
A. aucun contrôle pour aller plus vite
B. une exécution automatique sans limite
C. une approbation humaine explicite et un journal

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quelle tâche répétitive confierais-tu à un agent, et que se passerait-il au pire s'il se trompait ?
2. Quel signal te ferait réduire l'autonomie d'un agent que tu utilises ?

À retenir

À retenir

1. Boucle observer–décider–agir, avec contrôle de l'hôte à chaque action.
2. Modes de panne : boucle, dérive, triche sur la mesure, action destructrice, coût.
3. Harnais : budgets, stagnation, périmètre, approbation, instantanés, trace.
4. Vérification indépendante (tests cachés) contre Goodhart.
5. Les erreurs se composent : vérifier souvent et à bon marché.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Dompteur d'agents

Prochaine mission : Évaluation et benchmarks

