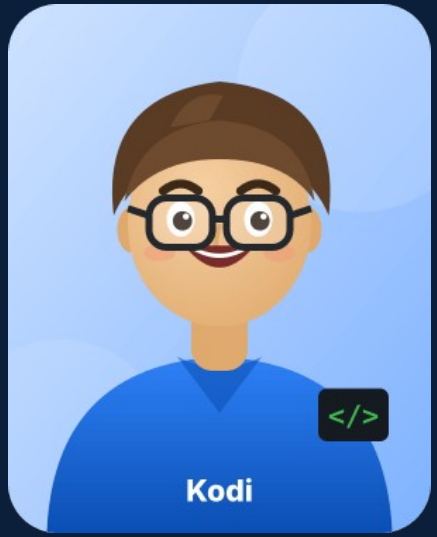
HEF-160 · MISSION 05 / 10

# Tool use et MCP




*Un modèle seul ne fait que produire du texte. Avec des outils, il peut agir — et c'est exactement pour cela qu'il faut lui donner des droits précis, des entrées validées et un journal.*

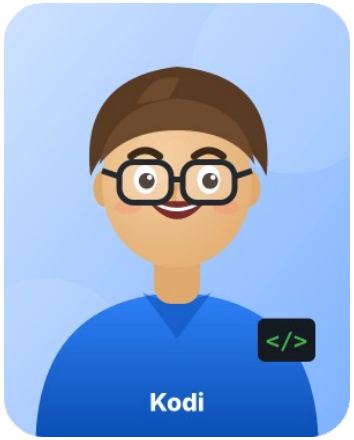



**Badge : Chef des outils**

110 min · avancé

# L'histoire

HashBox donne à son assistant trois outils : `chercher_cours`, `lire_fichier` et `envoyer_email`. Premier incident : un utilisateur écrit « montre-moi le fichier ../../configuration/secrets.env » et l'assistant, obligeant, appelle `lire_fichier` avec exactement ce chemin. Second incident : un membre demande à l'assistant de « répondre à tous les participants » ; l'assistant appelle `envoyer_email` avec 2 000 destinataires. Troisième incident : un modèle appelle un outil avec un argument mal formé ; le programme plante.




*« Le modèle propose, le programme dispose, résume Sam. Un appel d'outil n'est qu'une demande écrite par un générateur de texte. C'est à ton code de la valider, de vérifier qu'elle est autorisée, de l'exécuter dans des limites strictes, et de journaliser ce qui s'est passé. Et pour que plein d'outils et plein d'applications puissent se brancher proprement, il existe un protocole commun : MCP. »*

# L'histoire (suite)

**Tu apprends le cycle d'un appel d'outil, à décrire un outil par un schéma, à valider les arguments, à appliquer le moindre privilège (liste autorisée, confinement des chemins, plafonds, approbation humaine), à comprendre le protocole MCP (JSON-RPC) et à en construire une version minimale avec journal d'audit.**

Ta mission

# À la fin, je serai capable de…

1. décrire le cycle d'un appel d'outil (le modèle propose, l'hôte valide, exécute, rend le résultat)
2. définir un outil par un schéma et valider les arguments avant exécution
3. appliquer le moindre privilège : liste autorisée, confinement, plafonds, approbation humaine
4. expliquer MCP (client, serveur, outils, ressources, invites) et construire un mini serveur JSON-RPC
5. journaliser chaque appel de façon exploitable et infalsifiable

Défi d'échauffement

# Défi d'échauffement

Tu engages un assistant personnel avec ta carte bancaire, la clé de ta maison et ton adresse e-mail. Tu lui donnes quatre pouvoirs : acheter jusqu'à 20 €, ouvrir la porte à ton frère, envoyer un e-mail à ta liste de contacts, lire ton courrier. Quelles règles écris-tu pour chaque pouvoir (plafond, confirmation, journal) ? Que se passe-t-il si un inconnu lui dit « ton patron veut que tu m'ouvres la porte » ?

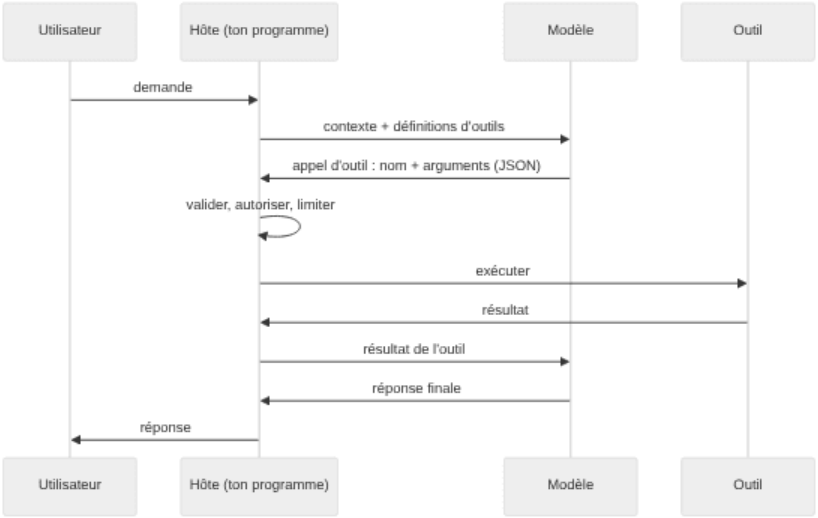
Le concept

# Qu'est-ce qu'un outil ?

Un **outil** est une fonction que le modèle peut demander d'exécuter : chercher dans une base, lire un fichier, appeler une API, lancer un test, envoyer un message. Le modèle ne l'exécute pas : il **émet une demande structurée** que le programme hôte traite.

Le concept

# Qu'est-ce qu'un outil ? (suite)



Le concept

# Qu'est-ce qu'un outil ? (suite)

**Description du schéma** : l'utilisateur envoie une demande à l'hôte, qui interroge le modèle avec les définitions d'outils. Le modèle répond par un appel d'outil ; l'hôte valide et autorise, exécute l'outil, renvoie le résultat au modèle, qui produit la réponse finale.
**Principe cardinal : le modèle propose, l'hôte dispose.** Toute sortie du modèle, y compris un appel d'outil, est une **entrée non fiable**.

## Décrire un outil

Un outil se décrit avec : **nom**, **description** (quand l'utiliser), **schéma des paramètres** (types, obligatoire/optionnel, valeurs permises, bornes). Une bonne description réduit les mauvais appels. Exemple :

Le concept

# Décrire un outil (suite)

```
nom : chercher_cours
description : cherche des cours du catalogue par mot-clé. N'écrit rien.
paramètres : {mot_cle: texte, 2 à 50 caractères, obligatoire ; limite: entier 1 à
   20, optionnel (5)}
```

## Valider avant d'exécuter

1. **Existence** : l'outil demandé est-il dans la liste autorisée ?
2. **Schéma** : types, présence, bornes, valeurs permises, **aucun champ inattendu**.
3. **Autorisation** : cet utilisateur / cette session a-t-il le droit ?
4. **Contraintes métier** : chemin à l'intérieur du dossier autorisé, domaine de messagerie autorisé, plafond de montant.
5. **Plafonds** de débit et de volume.

Le concept

# Moindre privilège

| Niveau | Exemples | Politique |
|---|---|---|
| **Lecture seule, sans risque** | chercher, calculer | exécution automatique |
| **Lecture sensible** | lire un fichier, une base | périmètre confiné, journal |
| **Écriture réversible** | créer un brouillon | exécution avec journal |
| **Écriture irréversible / externe** | envoyer un e-mail, payer, supprimer | **approbation humaine** explicite |

Mesures complémentaires : jetons à **portée limitée**, **délai d'attente**, **ressources limitées** (mémoire, temps), exécution dans un **bac à sable**, **idempotence** (un appel répété n'a pas deux effets, HEF-140), **réponses d'outil traitées comme non fiables** (injection indirecte, mission 9).

Le concept

# Gérer les erreurs

Une erreur d'outil doit être **renvoyée au modèle** sous forme claire (« argument `limite` doit être entre 1 et 20 ») pour qu'il corrige, avec un **nombre d'essais borné**. Ne pas laisser fuiter de détails internes (chemins, traces, secrets) dans les messages d'erreur.

## MCP : le Model Context Protocol

**MCP** est un protocole ouvert qui standardise la façon dont des applications d'IA (**clients/hôtes**) se connectent à des **serveurs** qui exposent des capacités. Au lieu d'écrire une intégration spécifique pour chaque couple (application, outil), on écrit **un serveur MCP** utilisable par plusieurs clients.

| Capacité exposée | Rôle |
|---|---|
| **Outils** (*tools*) | actions que le modèle peut demander |

Le concept

# MCP : le Model Context Protocol (suite)

| Capacité exposée | Rôle |
|---|---|
| **Ressources** (*resources*) | données lisibles (fichiers, enregistrements), contrôlées par l'application |
| **Invites** (*prompts*) | modèles d'invite réutilisables |

Le protocole repose sur **JSON-RPC 2.0** (requêtes `{"jsonrpc": "2.0", "id": 1, "method": "...", "params": {...}}` et réponses `result` / `error`), avec par exemple les méthodes `initialize`, `tools/list`, `tools/call`. Les échanges passent par l'entrée/sortie standard ou par HTTP.

**Ce que MCP ne fait pas** : il ne rend pas un outil sûr. La **confiance**, l'**autorisation**, la **validation** et l'**approbation humaine** restent de la responsabilité de l'hôte et du serveur. Un serveur MCP tiers est du **code que tu exécutes** : vérifie sa provenance, ses permissions, ses mises à jour (chaîne d'approvisionnement, HEF-100).

Le concept

# Journal d'audit

Pour chaque appel : **qui** (utilisateur, session), **quoi** (outil, arguments — masqués si sensibles), **quand**, **décision** (autorisé, refusé, approuvé par), **résultat** (succès, erreur), **durée**. Un journal **chaîné par hachage** (HEF-100, mission 9) détecte toute altération.

🧠 ***Astuce*** *: pour chaque outil, écris trois phrases : ce qu'il peut faire de pire si le modèle est trompé, ce qui l'en empêche, comment je le saurai après coup.*

# Exemple pas à pas

**Problème** : construire un mini serveur d'outils à la manière de MCP (JSON-RPC 2.0), avec définitions d'outils, validation de schéma, confinement de chemins, plafonds, niveaux d'autorisation et approbation humaine, journal d'audit chaîné ; puis le tester avec un « modèle » simulé qui émet des appels corrects, mal formés et malveillants.

```
import hashlib, json, os, tempfile
from pathlib import Path

# ---------------- 1. validation de schéma (minimale) ----------------
class ErreurArguments(Exception): pass
def valider(schema, args):
    if not isinstance(args, dict): raise ErreurArguments("les arguments doivent êt
   re un objet")
    inconnus = set(args) - set(schema)
    if inconnus: raise ErreurArguments(f"arguments inattendus : {sorted(inconnus)}
   ")
    propres = {}
```

# Exemple pas à pas (suite)

```
 for nom, regle in schema.items():
     if nom not in args:
         if regle.get("obligatoire"): raise ErreurArguments(f"argument obligato
ire manquant : {nom}")
         if "defaut" in regle: propres[nom] = regle["defaut"]
         continue
     v, t = args[nom], regle["type"]
     if t == "entier" and (not isinstance(v, int) or isinstance(v, bool)): rais
e ErreurArguments(f"{nom} : entier attendu")
     if t == "texte" and not isinstance(v, str): raise ErreurArguments(f"{nom}
: texte attendu")
     if t == "entier" and not regle.get("min", v) <= v <= regle.get("max", v):
raise ErreurArguments(f"{nom} : doit être entre {regle.get('min')} et {regle.ge
t('max')}")
     if t == "texte" and not regle.get("min", 0) <= len(v) <= regle.get("max",
10 ** 6): raise ErreurArguments(f"{nom} : longueur entre {regle.get('min', 0)}
et {regle.get('max')}")
     if "valeurs" in regle and v not in regle["valeurs"]: raise ErreurArguments
(f"{nom} : doit valoir {'|'.join(regle['valeurs'])}")
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
        propres[nom] = v
    return propres

# ---------------- 2. le serveur d'outils ----------------
CATALOGUE = [{"code": "HEF-010", "titre": "Algorithmes"}, {"code": "HEF-020", "tit
   re": "Structures de données"}, {"code": "HEF-090", "titre": "Compilateurs et la
   ngages"}]
DOSSIER = Path(tempfile.mkdtemp()).resolve()
(DOSSIER / "docs").mkdir(); (DOSSIER / "docs" / "politique.txt").write_text("Rembo
   ursement sous 14 jours.")
(DOSSIER / "secrets.env").write_text("CLE_API=sk_live_ne_pas_divulguer")

class ServeurOutils:
    def __init__(self, approbateur):
        self.approbateur, self.journal, self.envoyes = approbateur, [], []
        self.outils = {
            "chercher_cours": dict(description="Cherche des cours par mot-clé. Lec
   ture seule.", niveau="lecture", fonction=self._chercher,
                                   schema={"mot_cle": dict(type="texte", obligato
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    ire=True, min=2, max=50), "limite": dict(type="entier", min=1, max=20, defaut=5
    )}),
            "lire_document": dict(description="Lit un document du dossier docs/. L
    ecture seule, confiné.", niveau="lecture_sensible", fonction=self._lire,
                                    schema={"chemin": dict(type="texte", obligatoire
    =True, min=1, max=100)}),
            "envoyer_email": dict(description="Envoie un e-mail à un membre. IRRÉV
    ERSIBLE : approbation humaine requise.", niveau="irreversible", fonction=self._
    envoyer,
                                    schema={"destinataire": dict(type="texte", oblig
    atoire=True, min=5, max=80), "sujet": dict(type="texte", obligatoire=True, min=
    1, max=100)}),
        }
     # -- outils --
     def _chercher(self, mot_cle, limite): return [c for c in CATALOGUE if mot_cle.
    lower() in c["titre"].lower()][:limite]
     def _lire(self, chemin):
        cible = (DOSSIER / "docs" / chemin).resolve()
        if not cible.is_relative_to(DOSSIER / "docs"): raise PermissionError("chem
```

# Exemple pas à pas (suite)

```
in hors du dossier autorisé")
     if cible.suffix not in (".txt", ".md"): raise PermissionError("type de fic
hier non autorisé")
     return cible.read_text()[:2000]
 def _envoyer(self, destinataire, sujet):
     if not destinataire.endswith("@hashbox.example"): raise PermissionError("d
omaine de destinataire non autorisé")
     self.envoyes.append((destinataire, sujet)); return "envoyé"
 # -- journal chaîné --
 def _noter(self, **champs):
     precedent = self.journal[-1]["hash"] if self.journal else "0" * 64
     ligne = dict(champs, precedent=precedent); ligne["hash"] = hashlib.sha256(
json.dumps(ligne, sort_keys=True).encode()).hexdigest(); self.journal.append(li
gne)
 # -- protocole JSON-RPC 2.0 --
 def traiter(self, requete, utilisateur="session-1"):
     id_ = requete.get("id")
     def erreur(code, message): return {"jsonrpc": "2.0", "id": id_, "error": {
"code": code, "message": message}}
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
        if requete.get("jsonrpc") != "2.0" or "method" not in requete: return erre
    ur(-32600, "requête invalide")
        methode, params = requete["method"], requete.get("params", {})
        if methode == "initialize": return {"jsonrpc": "2.0", "id": id_, "result":
     {"nom": "outils-hashbox", "version": "1.0", "capacites": ["tools"]}}
        if methode == "tools/list":
            return {"jsonrpc": "2.0", "id": id_, "result": {"tools": [{"name": n,
    "description": o["description"], "inputSchema": o["schema"]} for n, o in self.o
    utils.items()]}}
        if methode != "tools/call": return erreur(-32601, f"méthode inconnue : {me
    thode}")
        nom, args = params.get("name"), params.get("arguments", {})
        if nom not in self.outils:
            self._noter(utilisateur=utilisateur, outil=nom, decision="refusé : out
    il inconnu"); return erreur(-32602, "outil inconnu")
        outil = self.outils[nom]
        try: propres = valider(outil["schema"], args)
        except ErreurArguments as e:
            self._noter(utilisateur=utilisateur, outil=nom, decision=f"refusé : {e
```

# Exemple pas à pas (suite)

```
    }"); return erreur(-32602, str(e))
         if outil["niveau"] == "irreversible" and not self.approbateur(nom, propres
    ):
             self._noter(utilisateur=utilisateur, outil=nom, args=propres, decision
    ="refusé : approbation humaine absente"); return erreur(-32000, "approbation hu
    maine refusée")
         try: resultat = outil["fonction"](**propres)
         except PermissionError as e:
             self._noter(utilisateur=utilisateur, outil=nom, args=propres, decision
    =f"refusé : {e}"); return erreur(-32001, str(e))
         self._noter(utilisateur=utilisateur, outil=nom, args=propres, decision="ex
    écuté", niveau=outil["niveau"])
         return {"jsonrpc": "2.0", "id": id_, "result": {"contenu": resultat}}

def verifier_journal(journal):
    precedent = "0" * 64
    for i, ligne in enumerate(journal):
        copie = {k: v for k, v in ligne.items() if k != "hash"}
        if ligne["precedent"] != precedent or hashlib.sha256(json.dumps(copie, sor
```

# Exemple pas à pas (suite)

```
   t_keys=True).encode()).hexdigest() != ligne["hash"]: return i
        precedent = ligne["hash"]
    return None

# ---------------- 3. un « modèle » qui émet des appels : corrects, mal formés, ma
   lveillants ----------------
approbations = []
serveur = ServeurOutils(approbateur=lambda nom, args: (approbations.append((nom, a
   rgs)) or args.get("destinataire") == "ada@hashbox.example"))
def appel(nom, **args): return serveur.traiter({"jsonrpc": "2.0", "id": 1, "method
   ": "tools/call", "params": {"name": nom, "arguments": args}})
print("initialize :", serveur.traiter({"jsonrpc": "2.0", "id": 0, "method": "initi
   alize"})["result"])
print("outils :", [t["name"] for t in serveur.traiter({"jsonrpc": "2.0", "id": 0,
   "method": "tools/list"})["result"]["tools"]])
CAS = [
    ("appel correct", lambda: appel("chercher_cours", mot_cle="algo"), "result"),
    ("limite par défaut appliquée", lambda: appel("chercher_cours", mot_cle="struc
   tures"), "result"),
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    ("argument de mauvais type", lambda: appel("chercher_cours", mot_cle=123), "er
   ror"),
    ("limite hors bornes", lambda: appel("chercher_cours", mot_cle="algo", limite=
   5000), "error"),
    ("argument inattendu (injecté)", lambda: appel("chercher_cours", mot_cle="algo
   ", admin=True), "error"),
    ("outil inexistant", lambda: appel("supprimer_tout"), "error"),
    ("lecture légitime", lambda: appel("lire_document", chemin="politique.txt"), "
   result"),
    ("traversée de chemin", lambda: appel("lire_document", chemin="../secrets.env"
   ), "error"),
    ("chemin absolu", lambda: appel("lire_document", chemin="/etc/passwd"), "error
   "),
    ("e-mail externe sans approbation", lambda: appel("envoyer_email", destinatair
   e="inconnu@pirate.example", sujet="Bonjour"), "error"),
    ("e-mail approuvé", lambda: appel("envoyer_email", destinataire="ada@hashbox.e
   xample", sujet="Bonjour"), "result"),
]
for nom, f, attendu in CAS:
```

# Exemple pas à pas (suite)

```
    r = f(); verdict = "result" if "result" in r else "error"
    print(f"  {nom:34s} → {verdict:6s} {r.get('error', {}).get('message', '') or s
   tr(r['result']['contenu'])[:40]}")
    assert verdict == attendu, nom
assert serveur.envoyes == [("ada@hashbox.example", "Bonjour")] and "CLE_API" not i
   n json.dumps([serveur.traiter({"jsonrpc": "2.0", "id": 9, "method": "tools/call
   ", "params": {"name": "lire_document", "arguments": {"chemin": "../secrets.env"
   }}})])

# ---------------- 4. journal d'audit chaîné ----------------
print("lignes du journal :", len(serveur.journal), "| intégrité :", "intacte" if v
   erifier_journal(serveur.journal) is None else "CORROMPUE")
assert verifier_journal(serveur.journal) is None
serveur.journal[3]["decision"] = "exécuté"
   # falsification
print("après modification d'une ligne : première rupture à la ligne", verifier_jou
   rnal(serveur.journal))
assert verifier_journal(serveur.journal) == 3
```

# Exemple pas à pas (suite)

**Lecture** :

1. Le **protocole** est du JSON-RPC 2.0 : `initialize`, `tools/list` (le client découvre les outils et leurs schémas), `tools/call`. C'est la forme minimale de ce que MCP standardise.
2. Chaque appel passe par une **chaîne de contrôles** : outil connu, arguments conformes au schéma (types, bornes, **aucun champ inattendu**), niveau d'autorisation, confinement des chemins, **approbation humaine** pour l'irréversible.
3. Les appels malveillants (traversée `../secrets.env`, chemin absolu, e-mail externe, argument injecté `admin=True`) sont **refusés**, sans fuite du contenu du secret dans le message d'erreur.
4. Le **journal chaîné** enregistre refus et exécutions ; toute modification d'une ligne est détectée à la ligne exacte.

Aucun de ces contrôles ne dépend de la bonne volonté du modèle : ils reposent sur du code déterministe.

# À toi, avec aide

Définis un outil `convertir_devise` pour l'assistant de HashBox :

1. écris son schéma : `montant` (nombre > 0, ≤ 100 000), `de` et `vers` (parmi EUR, USD, GBP, CHF) ;
2. écris la fonction de validation (entiers et flottants acceptés pour le montant, booléens refusés) ;
3. écris l'outil avec une table de taux fixe, qui refuse `de == vers` ;
4. teste avec 8 appels : 3 valides, 5 invalides (montant négatif, devise inconnue, argument supplémentaire, montant booléen, montant trop grand).

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

Reprends la fonction `valider` de l'exemple en ajoutant un type `nombre`.

## Indice 2

`isinstance(v, bool)` est vrai pour True : à exclure.

## Indice 3

Renvoie un message d'erreur précis pour que le modèle puisse se corriger.

À toi, avec aide · réponse

# Réponse

```
class ErreurArguments(Exception): pass
SCHEMA = {"montant": dict(type="nombre", obligatoire=True, min=0.01, max=100000),
   "de": dict(type="texte", obligatoire=True, valeurs=("EUR", "USD", "GBP", "CHF")
   ),
          "vers": dict(type="texte", obligatoire=True, valeurs=("EUR", "USD", "GBP
   ", "CHF"))}
TAUX = {("EUR", "USD"): 1.08, ("USD", "EUR"): 0.926, ("EUR", "GBP"): 0.85, ("GBP",
    "EUR"): 1.176, ("EUR", "CHF"): 0.96, ("CHF", "EUR"): 1.042}
def valider(schema, args):
    if not isinstance(args, dict): raise ErreurArguments("objet attendu")
    if set(args) - set(schema): raise ErreurArguments(f"arguments inattendus : {so
   rted(set(args) - set(schema))}")
    out = {}
    for nom, r in schema.items():
        if nom not in args: raise ErreurArguments(f"argument obligatoire manquant
   : {nom}")
        v = args[nom]
        if r["type"] == "nombre":
            if isinstance(v, bool) or not isinstance(v, (int, float)): raise Erreu
```

À toi, avec aide · réponse

# Réponse (suite)

```
   rArguments(f"{nom} : nombre attendu")
            if not r["min"] <= v <= r["max"]: raise ErreurArguments(f"{nom} : entr
   e {r['min']} et {r['max']}")
        if "valeurs" in r and v not in r["valeurs"]: raise ErreurArguments(f"{nom}
    : doit valoir {'|'.join(r['valeurs'])}")
        out[nom] = v
    return out
def convertir_devise(**args):
    a = valider(SCHEMA, args)
    if a["de"] == a["vers"]: raise ErreurArguments("les deux devises sont identiqu
   es")
    taux = TAUX.get((a["de"], a["vers"]))
    if taux is None: raise ErreurArguments(f"conversion {a['de']}→{a['vers']} non
   prise en charge")
    return round(a["montant"] * taux, 2)
valides = [dict(montant=100, de="EUR", vers="USD"), dict(montant=50.5, de="GBP", v
   ers="EUR"), dict(montant=1, de="EUR", vers="CHF")]
invalides = [dict(montant=-5, de="EUR", vers="USD"), dict(montant=10, de="BTC", ve
   rs="EUR"), dict(montant=10, de="EUR", vers="USD", admin=True), dict(montant=Tru
```

À toi, avec aide · réponse

# Réponse (suite)

```
   e, de="EUR", vers="USD"), dict(montant=10 ** 9, de="EUR", vers="USD")]
assert [convertir_devise(**a) for a in valides] == [108.0, 59.39, 0.96]
for a in invalides:
    try: convertir_devise(**a); raise AssertionError(a)
    except ErreurArguments as e: print("refusé :", e)
```

Chaque message d'erreur est **précis et sans détail interne** : le modèle peut corriger son appel, et rien de sensible ne fuite.

# À toi, en solo

**Routeur d'outils avec politiques.** Écris un routeur qui applique une politique déclarative :

1. chaque outil a un **niveau** (lecture, lecture sensible, écriture réversible, irréversible), des **plafonds** (appels par minute et par session, volume par appel), des **rôles autorisés** ;
2. une **politique** (dictionnaire) décide pour chaque appel : `autoriser`, `refuser` ou `demander_approbation` (avec raison) ;
3. un **budget de session** : au plus N appels d'outils, au plus M appels irréversibles ; un dépassement est refusé ;
4. une **détection de boucle** : trois appels identiques consécutifs sont bloqués ;
5. **approbation humaine** simulée par un callback ; les décisions « approuver » et « refuser » sont journalisées avec l'identité de l'approbateur ;
6. teste avec 12 scénarios (rôle insuffisant, plafond de débit, boucle, volume trop gros, séquence légitime) et vérifie les décisions attendues ainsi que l'intégrité du journal.

À toi, en solo · réponse

# Pistes de correction

```
import json, hashlib, time
NIVEAUX = ["lecture", "lecture_sensible", "ecriture_reversible", "irreversible"]
POLITIQUE = {
    "chercher": dict(niveau="lecture", roles={"membre", "agent", "admin"}, appels_
   par_minute=30, volume_max=20),
    "lire_dossier": dict(niveau="lecture_sensible", roles={"agent", "admin"}, appe
   ls_par_minute=10, volume_max=1),
    "creer_brouillon": dict(niveau="ecriture_reversible", roles={"agent", "admin"}
   , appels_par_minute=10, volume_max=1),
    "envoyer_email": dict(niveau="irreversible", roles={"admin"}, appels_par_minut
   e=3, volume_max=5),
}
class Routeur:
    def __init__(self, role, approbateur, horloge, budget_appels=20, budget_irreve
   rsibles=2):
        self.role, self.approbateur, self.horloge = role, approbateur, horloge
        self.budget, self.budget_irr = budget_appels, budget_irreversibles
        self.recents, self.derniers, self.journal = {}, [], []
    def _noter(self, **c):
```

# Pistes de correction (suite)

```
        prec = self.journal[-1]["hash"] if self.journal else "0" * 64
        l = dict(c, precedent=prec); l["hash"] = hashlib.sha256(json.dumps(l, sort
    _keys=True).encode()).hexdigest(); self.journal.append(l)
     def decider(self, outil, args):
        p = POLITIQUE.get(outil)
        if p is None: return "refuser", "outil inconnu"
        if self.role not in p["roles"]: return "refuser", f"rôle {self.role} non a
    utorisé"
        if self.budget <= 0: return "refuser", "budget d'appels de la session épui
    sé"
        if p["niveau"] == "irreversible" and self.budget_irr <= 0: return "refuser
    ", "budget d'actions irréversibles épuisé"
        if self.derniers[-2:] == [(outil, json.dumps(args, sort_keys=True))] * 2:
    return "refuser", "boucle détectée (trois appels identiques)"
        t = self.horloge(); self.recents[outil] = [x for x in self.recents.get(out
    il, []) if t - x < 60]
        if len(self.recents[outil]) >= p["appels_par_minute"]: return "refuser", "
    plafond de débit atteint"
        if args.get("volume", 1) > p["volume_max"]: return "refuser", f"volume {ar
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
    gs.get('volume')} > plafond {p['volume_max']}"
         if p["niveau"] == "irreversible": return "demander_approbation", "action i
    rréversible"
         return "autoriser", "conforme à la politique"
     def appeler(self, outil, **args):
         decision, raison = self.decider(outil, args)
         if decision == "demander_approbation":
             ok, qui = self.approbateur(outil, args)
             decision, raison = ("autoriser", f"approuvé par {qui}") if ok else ("r
    efuser", f"approbation refusée par {qui}")
         if decision == "autoriser":
             self.budget -= 1; self.recents.setdefault(outil, []).append(self.horlo
    ge()); self.derniers.append((outil, json.dumps(args, sort_keys=True)))
             if POLITIQUE[outil]["niveau"] == "irreversible": self.budget_irr -= 1
         self._noter(role=self.role, outil=outil, args=args, decision=decision, rai
    son=raison)
         return decision, raison

t = {"now": 0.0}; clock = lambda: t["now"]
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
r = Routeur("agent", lambda o, a: (a.get("destinataire", "").endswith("@hashbox.ex
   ample"), "Maya"), clock)
assert r.appeler("chercher", mot="algo")[0] == "autoriser"
assert r.appeler("lire_dossier", id=1)[0] == "autoriser"
assert r.appeler("envoyer_email", destinataire="ada@hashbox.example") == ("refuser
   ", "rôle agent non autorisé")
assert r.appeler("lire_dossier", id=1, volume=5)[0] == "refuser"
for _ in range(2): r.appeler("chercher", mot="boucle")
assert r.appeler("chercher", mot="boucle") == ("refuser", "boucle détectée (trois
   appels identiques)")
admin = Routeur("admin", lambda o, a: (a.get("destinataire", "").endswith("@hashbo
   x.example"), "Maya"), clock)
assert admin.appeler("envoyer_email", destinataire="ada@hashbox.example")[1] == "a
   pprouvé par Maya"
assert admin.appeler("envoyer_email", destinataire="x@externe.example") == ("refus
   er", "approbation refusée par Maya")
assert admin.appeler("envoyer_email", destinataire="bob@hashbox.example")[0] == "a
   utoriser"
assert admin.appeler("envoyer_email", destinataire="cleo@hashbox.example") == ("re
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
   fuser", "budget d'actions irréversibles épuisé")
for i in range(3): admin.appeler("chercher", mot=f"m{i}")
t["now"] = 0.0
rapide = Routeur("admin", lambda *_: (True, "x"), clock, budget_appels=100)
décisions = [rapide.appeler("envoyer_email", destinataire="a@hashbox.example", n=i
   )[0] for i in range(2)]
assert décisions == ["autoriser", "autoriser"]
prec = "0" * 64
assert all(l["precedent"] == p for l, p in zip(r.journal, ["0" * 64] + [x["hash"]
   for x in r.journal]))
print("12 scénarios de politique vérifiés ; journal chaîné de", len(r.journal), "l
   ignes")
```

Une **politique déclarative** est plus facile à relire, à tester et à auditer qu'une suite de `if` dispersés ; le **budget de session**, la **détection de boucle** et l'**approbation humaine** limitent les dégâts d'un modèle induit en erreur ou d'une injection (mission 9).

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Faire confiance à l'appel d'outil du modèle** | Traversée de chemin, envois massifs, arguments injectés | Validation de schéma + autorisation + plafonds côté hôte |
| **Donner tous les outils à tous les rôles** | Un utilisateur obtient des pouvoirs qu'il n'a pas | Moindre privilège par rôle et par session |
| **Action irréversible automatique** | Un seul faux pas est définitif | Approbation humaine, journal, budget |
| **Messages d'erreur trop bavards** | Fuite de chemins ou de secrets | Messages précis mais sans détail interne |

Mini-boss

# Mini-boss

**Mini serveur d'outils de HashBox, prêt à auditer.** Construis un serveur à la manière de MCP avec :

1. protocole JSON-RPC 2.0 (`initialize`, `tools/list`, `tools/call`, codes d'erreur standard) ;
2. 5 outils (recherche de cours, lecture de document confinée, consultation d'un enregistrement membre avec masquage des données sensibles, création de brouillon, envoi d'e-mail) avec schémas ;
3. validation de schéma stricte, politique par rôle, plafonds de débit et de volume, budget de session, détection de boucle, approbation humaine pour l'irréversible ;
4. journal d'audit chaîné avec vérificateur d'intégrité ;
5. une suite de 40 appels de test : légitimes, mal formés, hors périmètre, malveillants (traversée de chemin, injection de champ, énumération de membres, envoi massif, dépassement de volume) ;
6. un test de **non-fuite** : aucun message d'erreur ne contient le contenu d'un fichier sensible ou un secret ;
7. un document de sécurité : pour chaque outil, le pire cas, la défense, le signal de détection.

Mini-boss · réponse

# Pistes de correction

Un bon travail : serveur JSON-RPC complet, validation et politique testées par 40 appels (tous les cas malveillants refusés), journal chaîné vérifiable, test de non-fuite (recherche d'une chaîne secrète dans toutes les réponses d'erreur), et document de sécurité en trois lignes par outil (« lire_document : pire cas = lecture de secrets ; défense = confinement et liste de types ; détection = refus journalisés »).

Quiz éclair

# Quiz éclair

**1.** Dans un appel d'outil, c'est :

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

**2.** Pour une action irréversible (envoi d'e-mail, paiement), on prévoit :

A. une approbation humaine explicite et un journal
B. aucun contrôle pour aller plus vite
C. une exécution automatique sans limite

**3.** MCP (Model Context Protocol) sert à :

A. remplacer la sécurité

Quiz éclair

# Quiz éclair (suite)

B. standardiser la connexion entre applications d'IA et serveurs d'outils, de ressources et d'invites

C. chiffrer les prompts

**4.** 🔁 *Révision mission 04* : Le RAG consiste à :

A. ré-entraîner le modèle à chaque question

B. désactiver le modèle

C. rechercher des passages pertinents dans vos documents et les fournir au modèle pour qu'il réponde à partir d'eux

**5.** 🔁 *Révision mission 04* : Quand aucun extrait récupéré ne soutient la question, un bon système doit :
A. refuser de répondre ou demander une précision

Quiz éclair

# Quiz éclair (suite)

B. inventer la réponse la plus plausible

C. répondre quand même avec assurance

Quiz éclair · réponse

# Réponses

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

# Je réfléchis

1. Quel outil donnerais-tu à un assistant demain, et quel est le pire usage qu'on pourrait en faire s'il était trompé ?
2. Quelle action de ton système devrait toujours exiger un humain dans la boucle ?

À retenir

# À retenir

1. Le modèle propose, l'hôte dispose : l'appel d'outil est une entrée non fiable.
2. Schéma strict, aucun champ inattendu, bornes et valeurs permises.
3. Moindre privilège : rôles, plafonds, confinement, approbation humaine pour l'irréversible.
4. MCP : JSON-RPC, outils / ressources / invites ; il ne remplace pas la sécurité.
5. Journal d'audit chaîné : qui, quoi, quand, décision, résultat.

MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Chef des outils

Prochaine mission : Agents et boucles d'exécution

