

Tutoriel

Agents · MCP · recherche documentaire

Créer un serveur MCP et un agent de support documentaire

Construire un serveur MCP en lecture seule et un agent qui choisit un outil pour répondre à une question de support à partir d'une base de connaissances locale.

Auteur **Youness Boutyour**

**DURÉE**

55 minutes

NIVEAU

Intermédiaire

FORMAT

Tutoriel pratique

Agents · MCP · recherche documentaire · publié le 2026-09-01 · local

Objectifs et livrables

« Je ne peux plus me connecter à mon compte » : pour répondre à cette demande, un assistant de support doit retrouver la bonne procédure et pouvoir la citer. Nous allons construire un serveur MCP qui donne accès à une petite base d'articles approuvés, puis un agent qui utilise ses outils pour préparer la réponse.

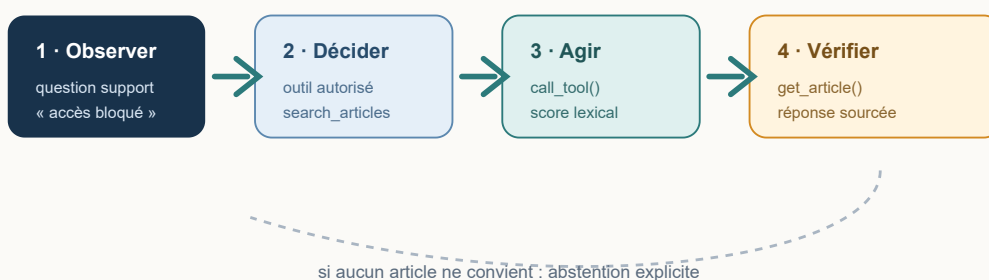
Vous préparerez le catalogue, exposerez la recherche et la lecture d'un article, puis vérifierez les outils avant de les relier à l'agent par `studio`. La recherche repose sur un score lexical simple, ce qui permet de comprendre les résultats avec les données du projet et de reproduire les essais localement.

Nous suivrons une demande depuis la découverte des outils jusqu'à la réponse citée. Un cas hors catalogue permettra aussi de voir comment l'agent indique qu'il manque d'information. Les appels restent limités aux articles autorisés ; le serveur assure la validation des entrées et l'agent utilise les résultats dans sa réponse.

Vous obtiendrez un exemple complet de support documentaire avec des tests, des sorties structurées et des traces d'appel. Les repères ci-dessous expliquent le rôle du protocole et le calcul du score ; vous pourrez vous y référer au fil de l'implémentation.

Boucle agentique minimale

Chaque transition est observable et chaque outil est borné.



Boucle minimale : observer la question, choisir un outil autorisé, récupérer une preuve, puis répondre.

```
$ python src/agent.py --question "Je ne peux plus me connecter"

"selected_tool": "search_articles -> get_article",
"article_id": "kb-001",
"score": 0.333,
"status": "grounded",

"answer": "Depuis l'écran de connexion,
sélectionnez Mot de passe oublié..."

MCP support agent: PASS
```

Sortie finale : outil appelé, article retenu, score de recherche et réponse vérifiable.

Faire passer une question par un serveur MCP

Schéma de la boucle agentique qui découvre les outils, recherche un article, lit la preuve puis répond avec sa source.



L'agent ne lit pas directement le catalogue : il découvre et appelle un outil au contrat explicite.

Lire le flux avant de coder

Le schéma décrit une boucle fermée : l'agent découvre les outils, appelle celui dont le contrat correspond à la demande, récupère une preuve et construit la réponse. Le serveur garde la maîtrise du catalogue et de la validation ; le modèle ne reçoit pas un accès direct aux fichiers.

Le score lexical est volontairement simple pour que vous puissiez expliquer chaque résultat. Un score nul doit conduire à une abstention, tandis qu'un score non nul doit encore être vérifié avec le contenu de l'article. Dans les deux cas, la trace d'appel est indispensable pour comprendre ce que l'agent a réellement consulté.

Quelques repères sur MCP

Ce que MCP apporte à un agent

Un agent outillé peut être décrit par la boucle suivante :

1. observer une demande utilisateur ;
2. sélectionner un outil à partir de son nom, sa description et son schéma d'entrée ;
3. appeler l'outil avec des arguments validés ;
4. intégrer le résultat dans le contexte de réponse ;
5. répondre avec une preuve et une limite explicite.

MCP sépare cette boucle de la source de données. Le serveur reste responsable de l'autorisation, de la validation et de la journalisation. Les outils, les ressources et les prompts sont trois primitives différentes : un outil est appelé par le modèle, une ressource est lue par l'application et un prompt est rendu comme message. Ce tutoriel utilise uniquement des outils, car la recherche et la lecture d'un article sont des actions contrôlées par l'agent.

Modèle de recherche utilisé

La démonstration n'utilise pas d'embeddings afin de rendre chaque résultat calculable à la main. Le serveur normalise les mots, retire quelques mots fonctionnels et calcule un score lexical :

■ Sortie / exemple

```
score(q, d) = |tokens(q) ∩ tokens(d)| / |tokens(q)|
```

Ce score n'est pas une mesure de pertinence universelle. Il sert de baseline déterministe : le comportement est reproductible, le corpus tient dans un fichier JSON et une réponse de score nul peut déclencher une abstention. Dans un système réel, on comparerait cette baseline à BM25 ou à des embeddings sur un jeu de questions annotées, en mesurant notamment la précision@k, le rappel@k et le taux d'abstention correcte.

1. Préparer l'environnement

Le projet comporte un serveur et un client : le premier expose la base de connaissances, le second découvre et appelle ses outils. Ils peuvent tourner sur la même machine sans que cette séparation disparaisse. Préparez l'environnement Python avant les fichiers ; le SDK fournit les échanges MCP, mais les fonctions de recherche et la sélection des articles restent du code métier à écrire.

Gardez l'arborescence de l'archive et les versions de dépendances utilisées. Le serveur stdio sera lancé par le client avec un interpréteur et un chemin explicites : cela évite de dépendre du dossier depuis lequel vous ouvrirez le terminal. L'installation et les imports sont vérifiables immédiatement ; la découverte des outils ne le sera qu'une fois le serveur implémenté.

Créez le projet et installez le SDK Python officiel en version majeure 2. L'extra `[cli]` fournit la commande `mcp` utile pour l'inspection locale.

Terminal · Bash

```
mkdir -p mcp-support-agent/{data,src,tests,outputs,logs}
cd mcp-support-agent
python3.11 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install "mcp[cli] ≥ 2, < 3"
python -m pip freeze > requirements.lock.txt
python --version
python -c "import mcp; print('MCP SDK:', getattr(mcp, '__version__', '2.x'))"
```

Sortie obtenue

Sortie / exemple

```
Python 3.11.x
MCP SDK: 2.x
```

Commande de vérification

Le fichier `requirements.lock.txt` doit exister. La borne `<3` empêche une mise à jour majeure silencieuse ; elle devra être réévaluée lors d'une migration du SDK. Aucun serveur distant n'est requis pour les étapes suivantes.

2. Préparer une base de connaissances réaliste

Chaque entrée représente une procédure support avec un identifiant indépendant du titre. Les champs courts servent à repérer un article, tandis que le corps contient les instructions qui pourront être restituées. Vérifiez l'unicité des identifiants et la cohérence entre titre, résumé, contenu et tags : une recherche correcte ne peut pas compenser une fiche mal renseignée.

Les quatre sujets donnent des cas faciles à distinguer, comme le mot de passe et le VPN. Gardez aussi une demande sans article correspondant pour vérifier l'abstention. `json.tool` contrôle la syntaxe du fichier, pas sa qualité documentaire ni tous les champs attendus ; relisez donc les entrées avant de commencer les appels MCP.

Créez `data/knowledge.json` avec quatre articles approuvés. Les identifiants sont stables afin que l'agent puisse citer une source sans recopier tout le corpus.

JSON

```
[
  {
    "id": "kb-001",
    "title": "Réinitialiser un mot de passe oublié",
    "summary": "Procédure de récupération d'accès au compte après expiration ou oubli du mot de passe.",
    "body": "Depuis l'écran de connexion, sélectionnez Mot de passe oublié, indiquez l'adresse professionnelle et utilisez le lien reçu dans les quinze minutes. Le lien expire après trente minutes. Le support ne demande jamais le mot de passe.",
    "tags": ["compte", "connexion", "mot de passe"]
  },
  {
    "id": "kb-002",
    "title": "Diagnostiquer une connexion VPN lente",
    "summary": "Premiers contrôles lorsqu'un accès VPN devient lent ou instable.",
    "body": "Vérifiez le réseau local, reconnectez le client VPN et comparez le débit sans VPN. Si le problème persiste, transmettez l'heure, la région et le nom du profil sans transmettre de secret.",
    "tags": ["vpn", "réseau", "latence"]
  },
  {
    "id": "kb-003",
    "title": "Comprendre un prélèvement sur une facture",
    "summary": "Identifier une ligne de facturation et demander une vérification au service financier.",
    "body": "Relevez la référence de facture et la période concernée. Ne transmettez pas de numéro complet de carte. Le support peut ouvrir une demande de vérification mais ne modifie pas la facture.",
    "tags": ["facture", "paiement", "finance"]
  },
  {
    "id": "kb-004",
    "title": "Restaurer un fichier depuis la sauvegarde",
    "summary": "Demander la restauration d'un fichier supprimé dans le périmètre autorisé.",
    "body": "Indiquez le chemin logique, la date approximative et le propriétaire du fichier. Une restauration doit être validée par le propriétaire des données avant toute écriture.",
    "tags": ["sauvegarde", "fichier", "restauration"]
  }
]
```

Sortie obtenue

Terminal · Bash

```
python -m json.tool data/knowledge.json >/dev/null && echo "knowledge.json: PASS"
```

Sortie / exemple

knowledge.json: PASS

Le jeu de données contient assez de distracteurs pour vérifier que l'agent ne renvoie pas systématiquement le premier article. Il ne contient ni donnée personnelle réelle, ni secret, ni contenu nécessitant un accès en écriture.

Commande de vérification

Terminal · Bash

```
python -c "import json; data=json.load(open('data/knowledge.json', encoding='utf-8')); assert len(data) == 4; assert {item['id'] for item in data} == {'kb-001', 'kb-002', 'kb-003', 'kb-004'}; print('articles: 4')"
```

Sortie / exemple

articles: 4

3. Écrire le serveur MCP en lecture seule

La recherche calcule un recouvrement de mots avec les articles, puis trie les correspondances. Ce score lexical aide à sélectionner une fiche dans ce petit catalogue ; il ne mesure ni une probabilité de réponse correcte ni une compréhension sémantique générale. `search_knowledge_base` renvoie des aperçus ; `get_article` permet ensuite de récupérer le contenu de l'identifiant retenu.

Gardez ces deux contrats distincts : la recherche accepte une formulation et une limite, la lecture attend un identifiant connu. Les annotations de types décrivent les arguments exposés, mais les fonctions doivent encore vérifier leurs valeurs métier. Le chemin du catalogue reste fixé côté serveur, et les diagnostics sont écrits dans le journal plutôt que sur stdout, réservé aux messages du protocole.

Créez `src/server.py`. `MCPServer` dérive les schémas d'entrée des annotations Python et transforme les fonctions décorées en outils appelables par un hôte MCP.

Python

```
from __future__ import annotations

import json
import logging
import re
from pathlib import Path
```

```

from mcp.server.mcpserver import MCPServer

ROOT = Path(__file__).resolve().parents[1]
DATA_FILE = ROOT / "data" / "knowledge.json"
LOG_FILE = ROOT / "logs" / "mcp-calls.log"
LOG_FILE.parent.mkdir(exist_ok=True)

logging.basicConfig(
    filename=LOG_FILE,
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(message)s",
)

mcp = MCPServer("support-knowledge-base")

STOP_WORDS = {
    "a", "au", "aux", "avec", "dans", "de", "des", "du", "en", "et",
    "je", "la", "le", "les", "me", "mon", "ne", "pas", "plus", "pour",
    "que", "sur", "un", "une", "à", "ce", "cette", "mon", "mes",
}

def load_articles() → list[dict]:
    """Charge uniquement le catalogue local approuvé."""
    payload = json.loads(DATA_FILE.read_text(encoding="utf-8"))
    if not isinstance(payload, list):
        raise ValueError("knowledge.json doit contenir une liste")
    return payload

def tokens(text: str) → set[str]:
    words = re.findall(r"[a-z0-9àâçéèëîïôöùüÿæ-]+", text.casefold())
    return {word for word in words if word not in STOP_WORDS and len(word) > 2}

def lexical_score(query: str, article: dict) → float:
    query_tokens = tokens(query)
    article_tokens = tokens(" ".join([
        article["title"], article["summary"], article["body"], *article["tags"]
    ]))
    if not query_tokens:
        return 0.0
    return round(len(query_tokens & article_tokens) / len(query_tokens), 3)

@mcp.tool()
def search_articles(query: str, limit: int = 3) → list[dict]:
    """Recherche des articles approuvés sans exposer le corpus complet."""
    normalized_limit = max(1, min(limit, 5))
    if not query.strip():
        logging.warning("tool=search_articles query=empty results=0")
        return []

```

```
ranked = [
    {
        "id": article["id"],
        "title": article["title"],
        "summary": article["summary"],
        "score": lexical_score(query, article),
    }
    for article in load_articles()
]
ranked = [item for item in ranked if item["score"] > 0]
ranked.sort(key=lambda item: (-item["score"], item["id"]))
result = ranked[:normalized_limit]
logging.info("tool=search_articles query=%r results=%d", query, len(result))
return result

@mcp.tool()
def get_article(article_id: str) -> dict:
    """Retourne le contenu d'un article identifié, sans accès à un chemin arbitraire."""
    wanted = article_id.strip()
    for article in load_articles():
        if article["id"] == wanted:
            logging.info("tool=get_article article_id=%s result=found", wanted)
            return article
    logging.warning("tool=get_article article_id=%s result=not_found", wanted)
    return {"status": "not_found", "article_id": wanted}

if __name__ == "__main__":
    mcp.run()
```

Sortie obtenue

Terminal · Bash

```
python -m compileall -q src && echo "server.py: PASS"
```

Sortie / exemple

```
server.py: PASS
```

Commande de vérification

Terminal · Bash

```
python -m compileall -q src && echo "server.py: PASS"
```

Contrôles de sécurité

- `get_article` accepte un identifiant, jamais un chemin fourni par l'utilisateur.
- `search_articles` borne `limit` à cinq résultats.
- Le serveur ne déclare aucune fonction d'écriture.
- Les logs vont dans `stderr` ou dans un fichier ; `stdout` reste réservé aux messages MCP du transport stdio.
- Le serveur renvoie un résultat vide ou `not_found` plutôt que d'inventer un article.

4. Vérifier le serveur sans agent

Le client en mémoire permet de tester le contrat sans lancer un autre processus : il vérifie la liste des outils, leurs appels et le contenu des résultats. Commencez par une recherche attendue, puis contrôlez que l'identifiant obtenu permet de lire le bon article. Une réussite isole le fonctionnement des handlers de celui du transport.

Ce test ne prouve pas encore que le lancement stdio fonctionnera dans votre terminal ou dans un autre hôte. La compilation vérifie, elle, la syntaxe Python, sans exécuter les scénarios métier. Utilisez donc chaque contrôle pour ce qu'il examine : syntaxe, découverte, recherche et lecture. Pour une demande inconnue, vérifiez l'absence de résultat plutôt qu'une fiche choisie par défaut.

Le SDK permet de connecter un client directement à l'objet serveur en mémoire. Cette forme est utile pour un contrôle rapide : il n'y a ni port, ni subprocess, mais la liste et l'appel des outils passent par la couche de protocole.

Créez `tests/test_mcp_server.py` :

```
Python

import asyncio

from mcp import Client

from src.server import mcp

async def check_server() → None:
    async with Client(mcp) as client:
        tools = await client.list_tools()
        names = {tool.name for tool in tools.tools}
        assert names == {"search_articles", "get_article"}

    result = await client.call_tool(
        "search_articles",
        {"query": "Je ne peux plus me connecter à mon compte", "limit": 3},
```

```
)
assert result.is_error is False
text = " ".join(getattr(block, "text", "") for block in result.content)
assert "kb-001" in text

def test_server_contract() → None:
    asyncio.run(check_server())
```

Lancez le test :

Terminal · Bash

```
pytest -q
```

Sortie obtenue

Sortie / exemple

```
1 passed in 0.4s
```

Ce test contrôle deux invariants importants : l'interface publiée reste limitée aux deux outils prévus et une question de connexion trouve l'article `kb-001`. Il ne dépend d'aucun modèle de langage.

5. Construire un agent qui utilise le serveur par stdio

Le client lance le serveur avec `sys.executable`, ouvre une session et vérifie les outils disponibles avant le premier appel. La boucle est ici explicite : chercher, choisir un résultat, lire son article, puis construire la réponse. Ce déroulement rend les décisions observables sans dépendre d'un modèle pour choisir chaque appel.

La réponse de cette démonstration reprend le contenu documentaire : elle n'est pas une synthèse libre produite automatiquement par un LLM. Si la recherche ne trouve rien, la boucle s'arrête sur l'abstention. Le lecteur de résultats distingue le contenu structuré des blocs textuels JSON ; examinez d'abord le statut d'erreur avant d'interpréter ces données comme un article valide.

Créez `src/agent.py`. Le programme joue le rôle de l'agent : il découvre les outils, choisit `search_articles`, puis appelle `get_article` pour obtenir la preuve complète. Le choix est déterministe afin de rendre l'exécution reproductible ; dans une application LLM, le planner pourrait sélectionner l'outil à partir des mêmes noms, descriptions et schémas.

 Python

```
from __future__ import annotations

import argparse
import asyncio
import json
import sys
from pathlib import Path

from mcp import Client, StdioServerParameters

ROOT = Path(__file__).resolve().parents[1]


def result_value(result):
    """Accepte le résultat structuré du SDK et garde un fallback texte."""
    structured = result.structured_content
    if structured is not None:
        if isinstance(structured, dict) and set(structured) == {"result"}:
            return structured["result"]
        return structured
    for block in result.content:
        text = getattr(block, "text", None)
        if text:
            return json.loads(text)
    return None


async def answer(question: str) → dict:
    server = StdioServerParameters(
        command=sys.executable,
        args=[str(ROOT / "src" / "server.py)],
        cwd=str(ROOT),
    )
    async with Client(server) as client:
        tools = await client.list_tools()
        available = sorted(tool.name for tool in tools.tools)
        if "search_articles" not in available or "get_article" not in available:
            raise RuntimeError(f"outils requis absents: {available}")

        search_result = await client.call_tool(
            "search_articles", {"query": question, "limit": 3}
        )
        hits = result_value(search_result) or []
        if not hits:
            return {
                "question": question,
                "tools": available,
                "selected_tool": "search_articles",
                "status": "insufficient_context",
                "answer": "Je ne trouve pas d'article approuvé pour cette demande.",
            }
```

```
best = hits[0]
article_result = await client.call_tool(
    "get_article", {"article_id": best["id"]}
)
article = result_value(article_result)
if not article or article.get("status") == "not_found":
    raise RuntimeError("l'article retourné par la recherche est introuvable")
return {
    "question": question,
    "tools": available,
    "selected_tool": "search_articles → get_article",
    "article_id": article["id"],
    "score": best["score"],
    "title": article["title"],
    "answer": article["body"],
    "status": "grounded",
}

def main() → None:
    parser = argparse.ArgumentParser()
    parser.add_argument("--question", required=True)
    args = parser.parse_args()
    print(json.dumps(asyncio.run(answer(args.question)), ensure_ascii=False, indent=2))

if __name__ == "__main__":
    main()
```

Le code final contient aussi `src/llm_client.py`, un adaptateur optionnel pour un endpoint compatible `/v1/chat/completions`. Il n'est pas appelé dans la démonstration : garder le planner déterministe permet de tester le serveur MCP sans coût, réseau ou modèle local. Pour passer à un planner LLM, faites-lui produire uniquement un objet `{"tool": "search_articles", "query": "..."}` , validez que `tool` appartient à l'allowlist, puis réutilisez le même appel `call_tool`. Le modèle ne doit jamais fournir un chemin de fichier ou une commande shell au serveur.

Sortie obtenue

Terminal · Bash

```
python src/agent.py --question "Je ne peux plus me connecter à mon compte"
```

JSON

```
{
  "question": "Je ne peux plus me connecter à mon compte",
  "tools": ["get_article", "search_articles"],
```

```
"selected_tool": "search_articles → get_article",
"article_id": "kb-001",
"score": 0.333,
"title": "Réinitialiser un mot de passe oublié",
"answer": "Depuis l'écran de connexion, sélectionnez Mot de passe oublié, indiquez l'adresse professionnelle et utilisez le lien reçu dans les quinze minutes. Le lien expire après trente minutes. Le support ne demande jamais le mot de passe.",
"status": "grounded"
}
```

Commande de vérification

La sortie doit contenir `status: grounded`, l'identifiant `kb-001` et les deux noms d'outils. Le champ `score` rend la décision observable ; il ne doit pas être présenté comme une probabilité de vérité.

Testez également l'abstention :

```
Terminal · Bash

python src/agent.py --question "Quel temps fera-t-il demain sur Mars ?"
```

```
JSON

{
  "selected_tool": "search_articles",
  "status": "insufficient_context",
  "answer": "Je ne trouve pas d'article approuvé pour cette demande."
}
```

L'agent ne fabrique pas une réponse lorsqu'aucun article ne franchit le seuil lexical implicite (`score > 0`). En production, ce seuil devrait être calibré sur des questions annotées et complété par une vérification humaine pour les sujets sensibles.

6. Inspecter l'interface MCP

Inspector fournit une seconde manière d'exercer le même serveur, indépendante du script client. Vérifiez que les deux outils apparaissent avec les paramètres attendus, puis reproduisez une recherche et une lecture déjà contrôlées. Vous pourrez comparer leurs résultats au scénario Python et repérer si une différence vient des arguments ou de l'environnement de lancement.

Essayez ensuite une requête vide, une limite inhabituelle et un identifiant absent pour observer les validations. Le but n'est pas de compter seulement les appels réussis, mais de rendre les comportements compréhensibles. L'absence d'outil d'écriture réduit le périmètre exposé ; elle ne constitue pas une authentifi-

cation des clients ni un contrôle général des accès au système.

Le serveur peut être lancé avec l'outil CLI du SDK :

```
Terminal · Bash  
  
mcp dev src/server.py
```

Ouvrez l'URL affichée par l'Inspector, consultez la liste [Tools](#), puis appelez [search_articles](#) avec :

```
JSON  
  
{  
  "query": "Je ne peux plus me connecter à mon compte",  
  "limit": 3  
}
```

Sortie observée dans l'Inspector

```
Sortie / exemple  
  
search_articles(query: string, limit: integer = 3)  
get_article(article_id: string)
```

L'interface est dérivée des annotations et des docstrings. Vérifiez que le serveur n'expose pas de fonction [write_article](#), [execute_sql](#) ou [run_shell_command](#). L'Inspector confirme le contrat ; il ne remplace pas une politique d'autorisation côté serveur.

7. Comprendre le transport et les limites

En stdio, le client et le processus serveur échangent des messages structurés sur les flux du processus. Un [print](#) de diagnostic sur stdout peut être confondu avec ces messages ; utilisez les journaux prévus. Le contexte de session encadre aussi la durée de vie du sous-processus : relancer le client ne doit pas laisser des serveurs orphelins s'accumuler.

Cette proximité locale ne dispense pas d'examiner les permissions du programme et son environnement. Le transport n'accorde pas à lui seul une autorisation métier sur les données. Passer à HTTP ajoutera des questions d'exposition réseau et d'authentification ; ce n'est pas simplement déplacer l'URL d'un outil local. La mini-série suivante reprend ces aspects sans changer le principe des contrats de lecture.

Dans cette démonstration, [StudioServerParameters](#) lance [server.py](#) comme subprocess. Le client et le serveur échangent des messages JSON-RPC délimités par des retours à la ligne sur stdin/stdout. Le serveur ne doit donc écrire aucun texte de diagnostic sur stdout ; les logs vont dans le fichier prévu.

Le même serveur peut être exposé par Streamable HTTP dans un environnement contrôlé. Le choix du transport ne transforme pas un outil en permission : l'autorisation reste dans le serveur, les arguments doivent être validés et les secrets doivent être transmis explicitement au processus avec une politique minimale.

Dans un agent connecté à un LLM, le prompt système devrait également préciser :

Sortie / exemple

```
Utilise uniquement les outils déclarés.  
Ne prétends pas avoir trouvé une information lorsque le résultat est vide.  
Ne transforme jamais un article en instruction d'écriture.  
Cite l'identifiant de l'article utilisé et demande une validation humaine pour une action sensible.
```

Résultats de référence

Pour une question couverte, la réponse doit contenir un identifiant d'article, le score calculé et le statut `grounded`. Pour une question hors corpus, le statut doit être `insufficient_context` et aucun contenu inventé ne doit être renvoyé.

Vérification finale

Exécutez la suite complète :

Terminal · Bash

```
python -m compileall -q src  
pytest -q  
python src/agent.py --question "Je ne peux plus me connecter à mon compte" | python -m json.tool >/dev/null  
test -s logs/mcp-calls.log  
echo "MCP support agent: PASS"
```

Sortie / exemple

```
1 passed in 0.4s  
MCP support agent: PASS
```

Le fichier `logs/mcp-calls.log` doit contenir au moins les appels `search_articles` et `get_article`. Ne publiez pas ce fichier s'il contient des identifiants, des requêtes sensibles ou des métadonnées internes.

Dépannage et diagnostics

- `ModuleNotFoundError: mcp` : vérifiez que l'environnement virtuel est activé et que `python -m pip install -r requirements.txt` a terminé sans erreur.
- `Client must be used within an async context manager` : placez les appels `list_tools()` et `call_tool()` dans le bloc `async with Client(...)`.
- Aucun outil n'apparaît dans l'Inspector : vérifiez que `mcp dev src/server.py` est lancé depuis la racine du projet et que `server.py` n'écrit pas de texte de diagnostic sur stdout.
- Le score vaut zéro : inspectez les tokens de la question et les tags de `knowledge.json` ; un résultat nul doit conduire à l'abstention, pas à une réponse inventée.
- Le log contient des données sensibles : ne réutilisez pas les requêtes brutes en production sans politique de masquage et durée de conservation.

Limites et prolongements

- Le score lexical est une baseline ; comparez-le à BM25 ou aux embeddings sur un corpus annoté avant de choisir une architecture.
- Le serveur local ne gère pas l'identité, les quotas ni l'audit centralisé.
- Le client de démonstration utilise un planner déterministe ; remplacez-le par un appel LLM uniquement après avoir conservé l'allowlist d'outils et les validations.
- Les appels d'écriture sont volontairement absents. Pour une action réelle, introduisez une approbation humaine, un journal d'audit et un périmètre de permissions distinct.
- Les tests en mémoire vérifient le contrat MCP ; un test d'intégration par stdio vérifie ensuite le cycle de vie du subprocess.

Le livrable final est un serveur MCP lisible, un client-agent reproductible et une base de connaissances sans secret. Le modèle peut choisir une capacité, mais il ne reçoit ni accès direct au disque ni autorité implicite sur le système.

Références

- [MCP — architecture officielle](#)
- [MCP Python SDK v2 — documentation officielle](#)
- [MCP Python SDK v2 — clients et transports](#)
- [MCP Python SDK — dépôt officiel](#)