

# Atelier: coder avec une IA générative

Utilisation de Large Language Models pour développer un logiciel  
en respectant les standards de l'industrie

Partie 5

Nicolas Debeissat  
[nicolas.debeissat@mail-formateur.net](mailto:nicolas.debeissat@mail-formateur.net)

# Plan du module

## **Installation de l'extension continue.dev**

Dans VSCode, avec ollama en local

## **Utiliser la configuration de base**

Création d'une stack React + FastAPI + Postgresql

## **Configuration avancée et codebase**

Comment adapter ses réponses

## **RAG avec Langchain**

API de génération de requêtes ElasticSearch

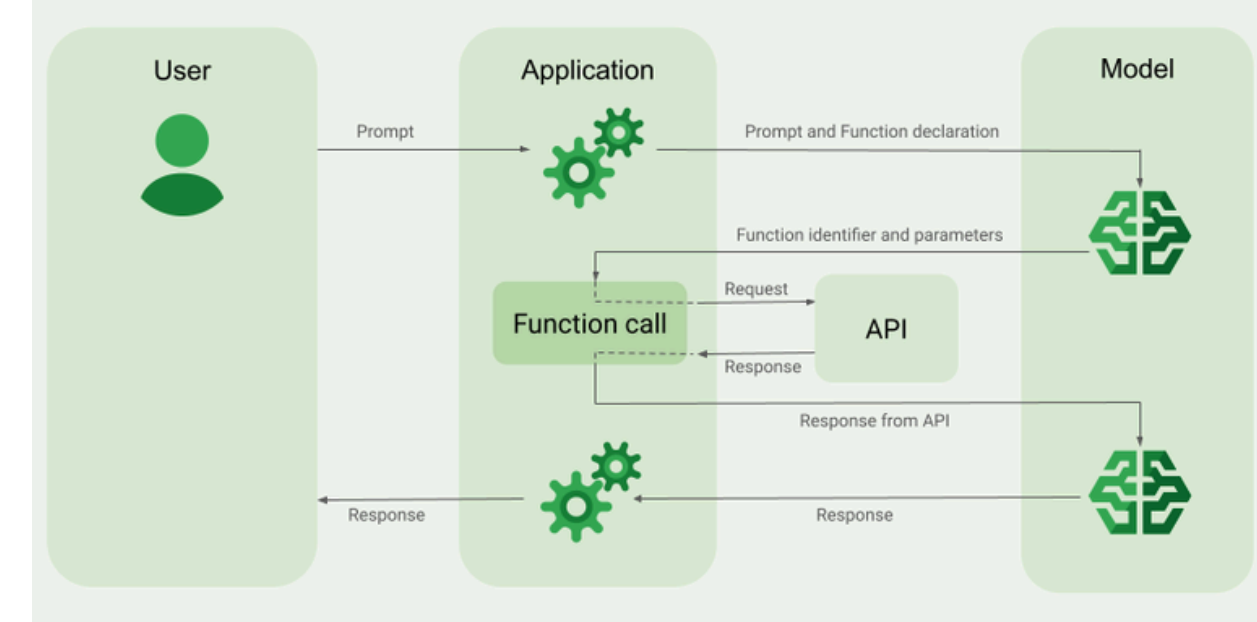
## **Model Context Protocol (mode Agent)**

Création d'outils et connexion à son agent

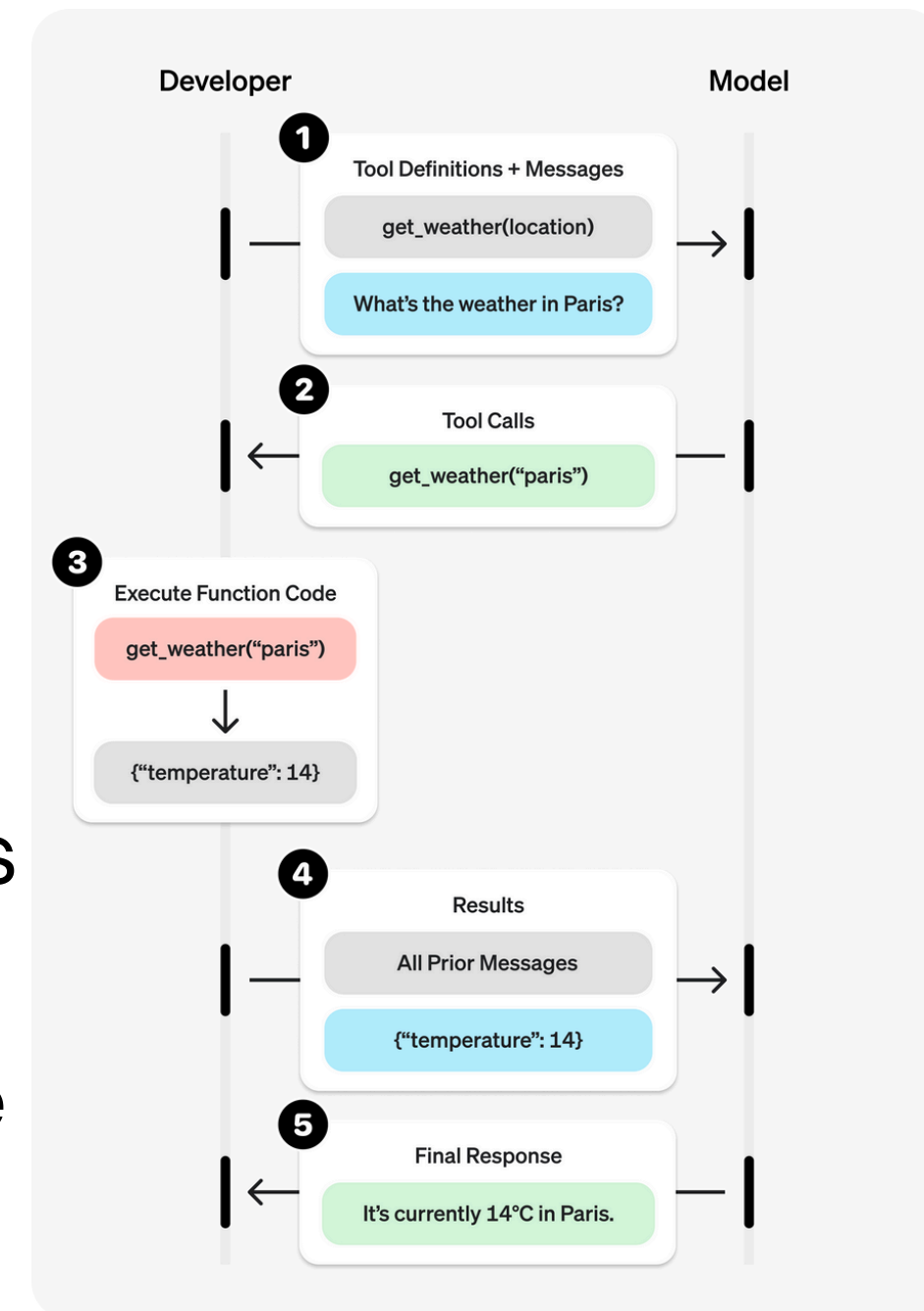
## **Evaluation**

Quizz et rendu du projet effectué en cours

# Function call



- L'utilisateur envoie le prompt suivant :
  - “combien d'utilisateurs français ai-je dans ma base de données ?”
- L'application ajoute au prompt la liste des fonctions dont il dispose
- Réponse du LLM :
  - Il faut utiliser l'outil “compte\_utilisateurs\_par\_pays” avec le paramètre pays = “France”
- L'application exécute la fonction choisie avec les paramètres
- L'application envoie la réponse au LLM avec l'historique de conversation pour que le LLM traduise la réponse en langage naturel : “Vous avez 45 utilisateurs français.”



# Function call

## Définition des outils

```
tools = [
    {
        "type": "function",
        "name": "get_horoscope",
        "description": "Get today's horoscope for an
astrological sign.",
        "parameters": {
            "type": "object",
            "properties": {
                "sign": {
                    "type": "string",
                    "description": "An astrological sign like Taurus or
Aquarius",
                },
            },
            "required": ["sign"],
        },
    },
]

def get_horoscope(sign):
    return f'{sign}: Next Tuesday you will befriend a baby
otter.'
```

## Appel du LLM

```
input_list = [
    {"role": "user", "content": "What is my horoscope? I am an
Aquarius."}
]

response = client.responses.create(
    model="gpt-5",
    tools=tools,
    input=input_list,
)

input_list += response.output
for item in response.output:
    if item.type == "function_call":
        if item.name == "get_horoscope":
            horoscope =
get_horoscope(json.loads(item.arguments))
            input_list.append({
                "type": "function_call_output",
                "call_id": item.call_id,
                "output": json.dumps({
                    "horoscope": horoscope
                })
            })
        })
```

## Traduction en langage naturel

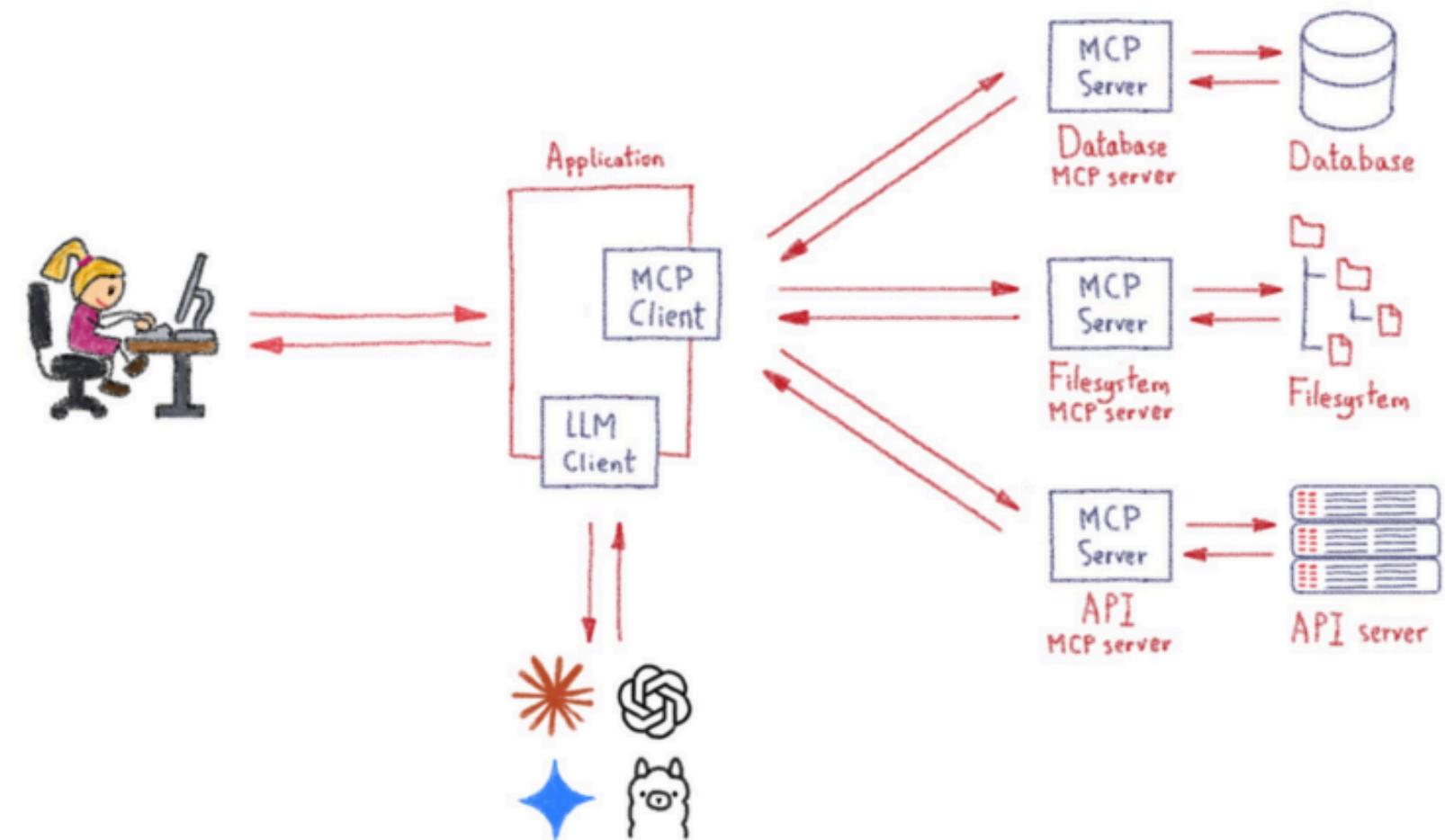
```
response = client.responses.create(
    model="gpt-5",
    instructions="Respond only with a horoscope generated
by a tool.",
    tools=tools,
    input=input_list,
)
```

# Problèmes du function call

- Chaque application maintient dans son code la liste des outils ?
- Chaque application est responsable de l'appel des outils ?

=> nécessite la création d'un standard pour pouvoir créer et utiliser des serveurs spécifiques à chaque usage, responsable de :

- lister les outils disponibles et renvoyer la liste au client MCP
- appeler les outils demandés par le client et retourner le résultat



# Model Context Protocol

1. Le client MCP agrège les listes d'outils données par les serveurs MCP
2. A partir de la question de l'utilisateur, le client demande au LLM quel outil utiliser et avec quels paramètres
3. Le client demande au serveur MCP d'appeler l'outil et récupère la réponse
4. Le client envoie l'historique de la conversation, avec l'appel d'outil, et son résultat, au LLM :
  - Soit le LLM décide que la question a été entièrement traitée, et traduit la réponse en langage naturel
  - Soit le LLM décide qu'il faut appeler un autre outil, et retourne le nom du nouvel outil à utiliser, avec les paramètres à utiliser
    - => retour à l'étape 3

# TD : création d'un serveur MCP

- `uv init mcp-server`
- `code mcp-server/`
- `cd mcp-server`
- `uv venv`
- `source .venv/bin/activate`
- `uv add mcp requests mcp[cli]`
- `uv run mcp dev main.py`

# TD : création d'un serveur MCP

```
from mcp.server.fastmcp import FastMCP
import requests

mcp = FastMCP("Demo")

@mcp.tool()
def get_mapping_elasticsearch(index_name: str) -> dict:
    """ Get the mapping of an Elasticsearch index. """
    url = f"http://localhost:9200/{index_name}"
    headers = {"Content-Type": "application/json"}
    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()
        return response.json()[index_name]['mappings']
    except requests.exceptions.RequestException as e:
        return f"An error occurred: {e}"

@mcp.tool()
def search_elasticsearch(index_name: str, query: dict) -> dict:
    """ Searches Elasticsearch with the provided query. """
    url = f"http://localhost:9200/{index_name}/_search"
    headers = {"Content-Type": "application/json"}
    try:
        body = {"query": query} if "query" not in query else query
        response = requests.post(url, headers=headers, json=body)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        return f"An error occurred: {e}"

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



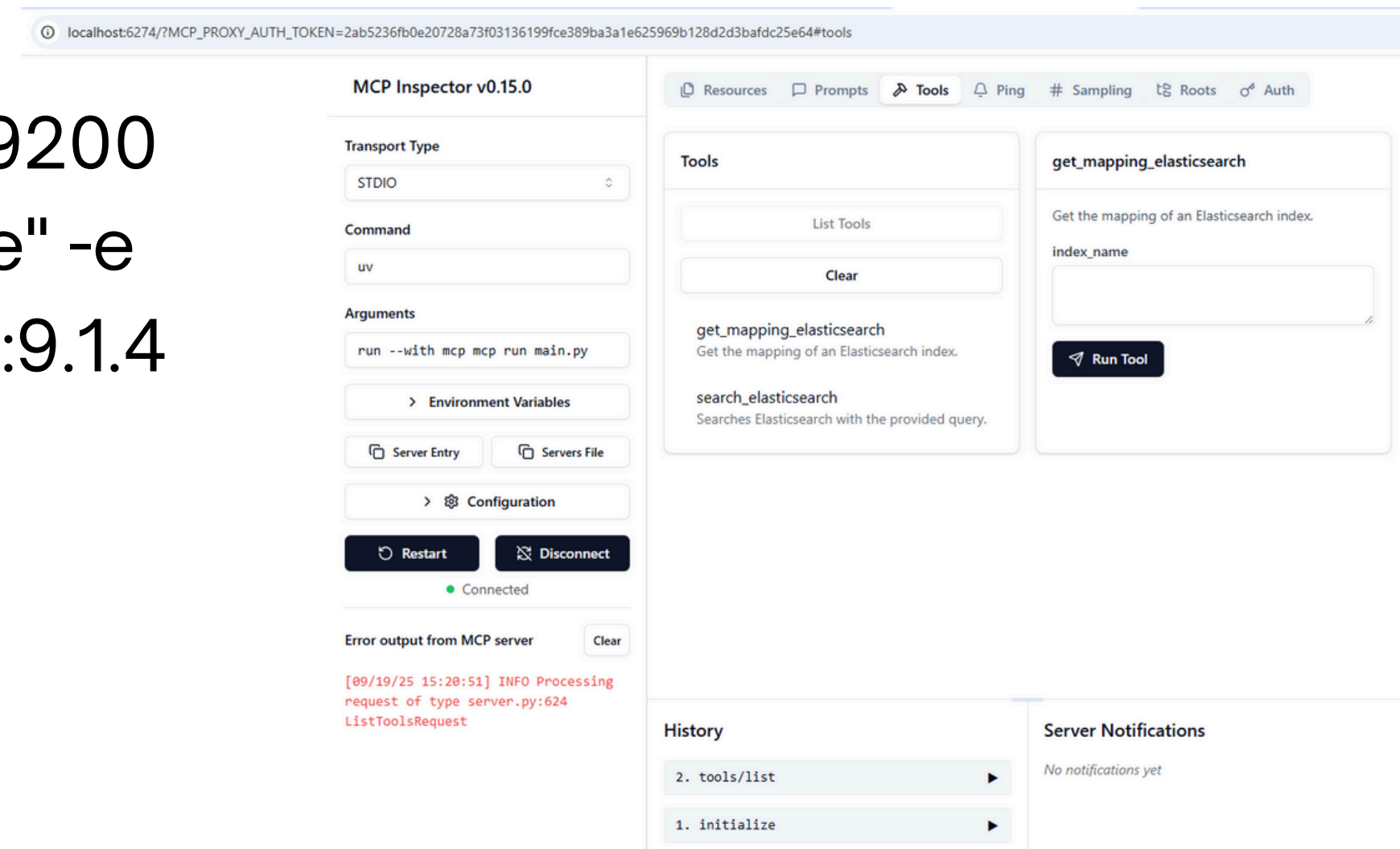
# TD : création d'un serveur MCP

- `uv run mcp dev main.py`
- Ouvrir l'inspector
- Démarrer un elasticsearch

🔗 Open inspector with token pre-filled:  
`http://localhost:6274/?MCP_PROXY_AUTH_TOKEN=2ab5236fb0e20728a73f03136199fce389ba3a1e625969b128d2d3bafdc25e64`

`docker run -d --name elasticsearch -p 9200:9200  
-p 9300:9300 -e "discovery.type=single-node" -e  
"xpack.security.enabled=false" elasticsearch:9.1.4`

- Stopper le docker :  
`docker stop elasticsearch`  
`docker rm elasticsearch`
- Lire les logs du docker :  
`docker logs -f elasticsearch`



# TD : configuration dans Continue

name: Mcp windows

version: 0.0.1

schema: v1

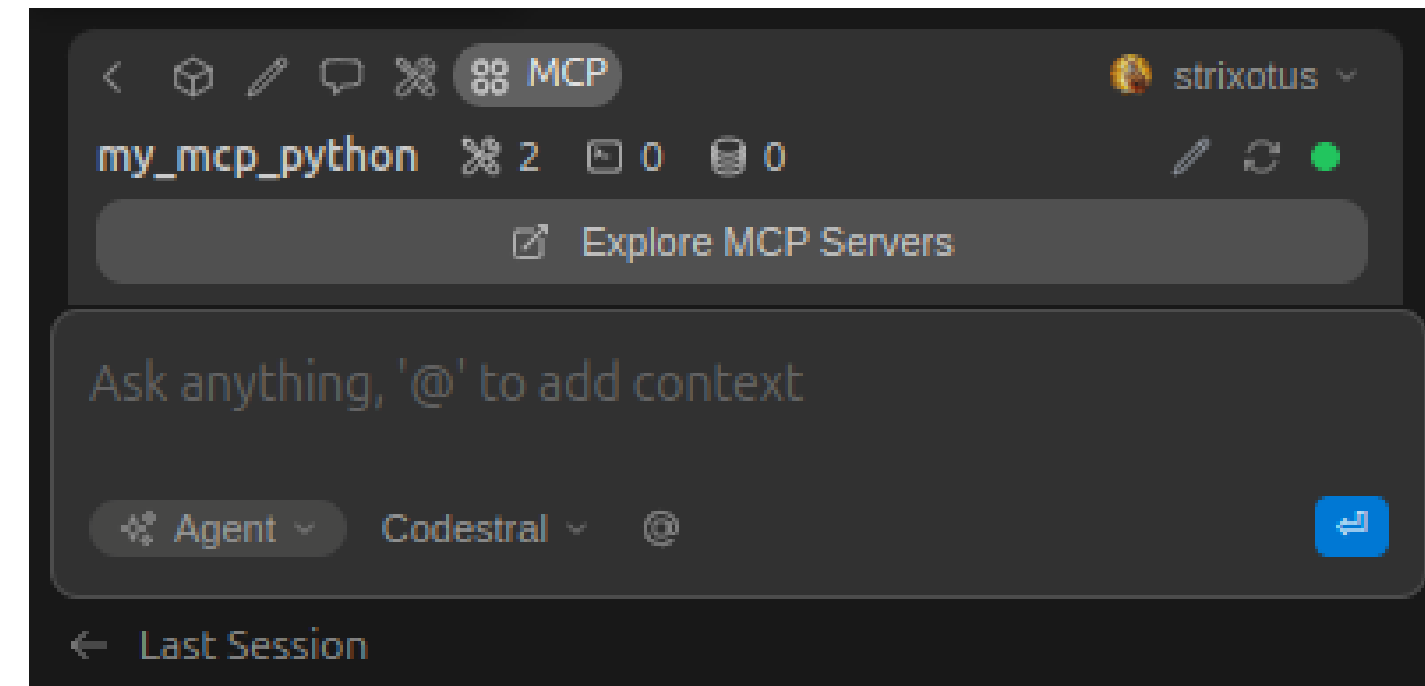
mcpServers:

- name: Mcp windows

- command: C:\Users\nicol\Documents\epsi\cours\atelier\_coder\_ia\mcp-server\.venv\Scripts\python.exe

- args:

- C:\Users\nicol\Documents\epsi\cours\atelier\_coder\_ia\mcp-server\main.py



# TD : appel du serveur

- Créer un index my-index

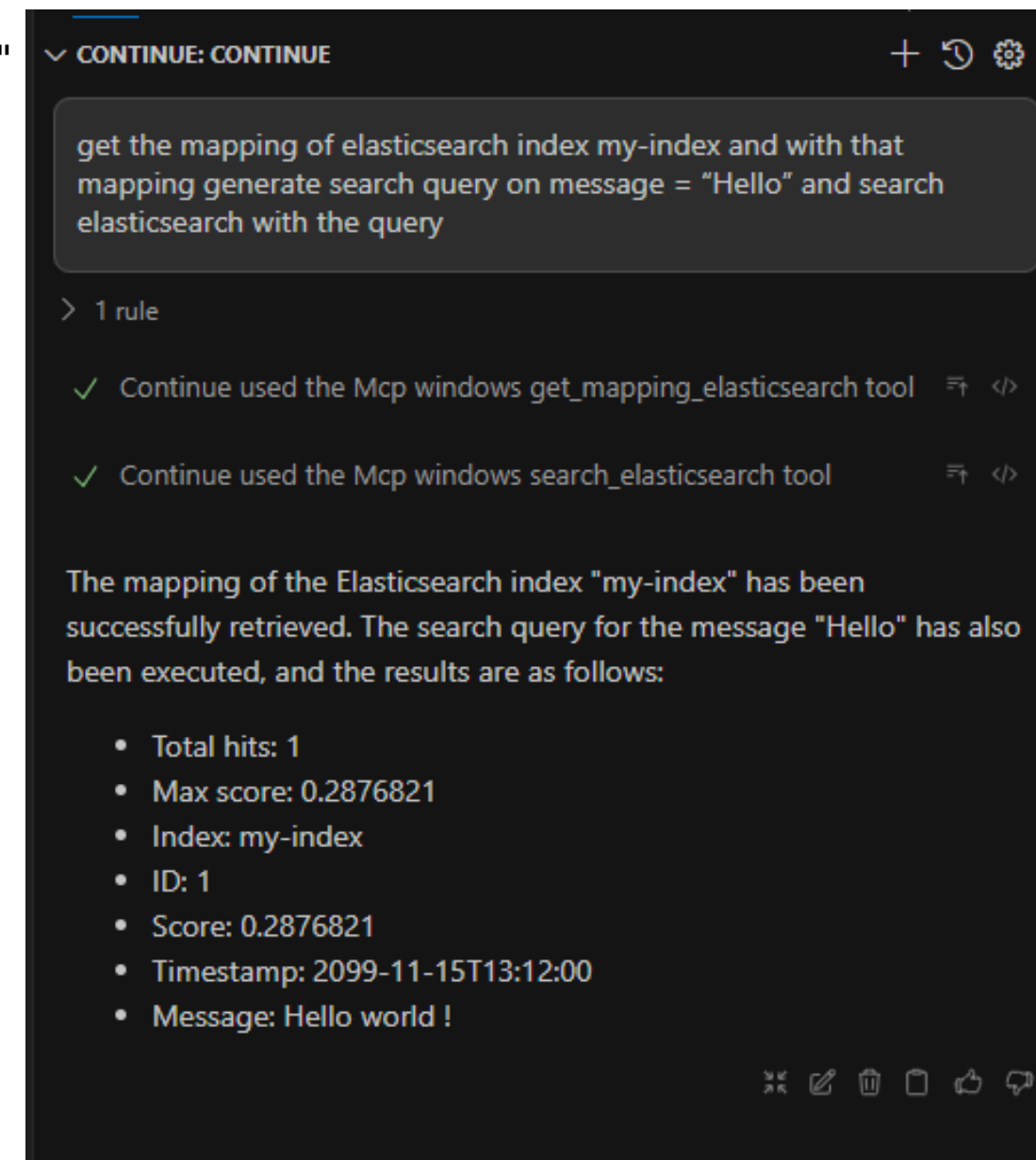
```
curl -X PUT -H "Content-Type: application/json" -d '{"mappings": {"properties": {"@timestamp": {"type": "date"}, "message": {"type": "text"}}}}' "http://localhost:9200/my-index"
```

- Voir l'index [http://localhost:9200/\\_cat/indices](http://localhost:9200/_cat/indices) et <http://localhost:9200/my-index>
- Créer un document

```
curl -X PUT -H "Content-Type: application/json" -d '{"@timestamp": "2099-11-15T13:12:00", "message": "Hello world !"}' "http://localhost:9200/my-index/_create/1"
```

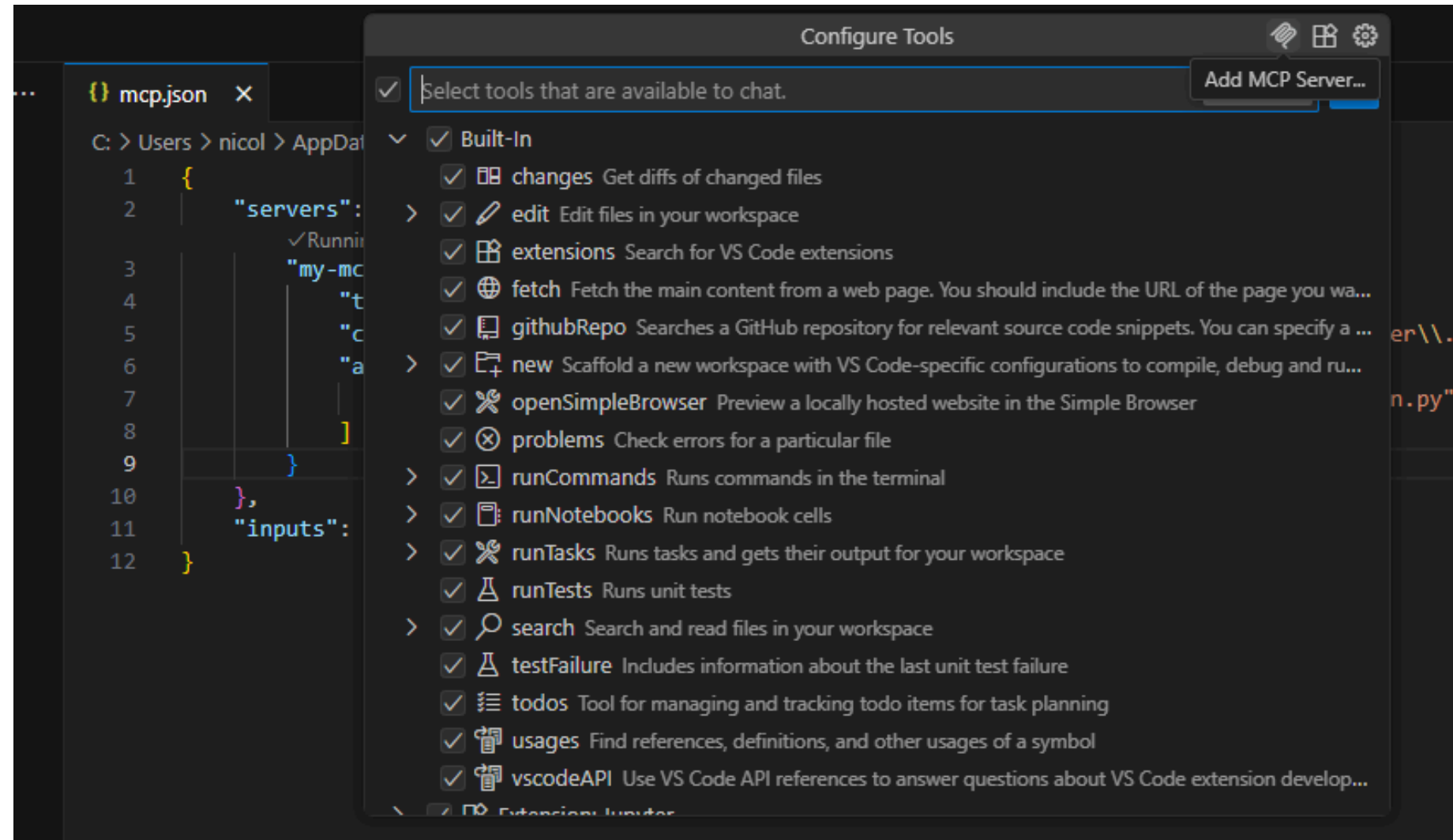
- Voir le document [http://localhost:9200/my-index/\\_doc/1](http://localhost:9200/my-index/_doc/1)
- Demander à continue en mode Agent :

“get the mapping of elasticsearch index my-index and with that mapping generate search query on message = “Hello” and search elasticsearch with the query”



# TD : appel du serveur

- Dans Copilot :



```
{
  "servers": {
    "my-mcp-server-5bf5d8ea": {
      "type": "stdio",
      "command": "C:\\Users\\nicol\\Documents\\epsi\\cours\\atelier_coder_ia\\mcp-server\\.venv\\Scripts\\python.exe",
      "args": [
        "C:\\Users\\nicol\\Documents\\epsi\\cours\\atelier_coder_ia\\mcp-server\\main.py"
      ]
    }
  },
  "inputs": []
}
```

