TP 4 : Spring AI

De l'IA locale aux Agents Autonomes distribués

Architecture Agentic avec MCP (Model Context Protocol)

Saoudi Haythem

Table des matières

1 Introduction et Objectifs

Ce TP a pour objectif de transformer votre compréhension théorique des Large Language Models (LLM) en compétences techniques concrètes pour construire des agents intelligents autonomes capables d'interagir avec le monde réel.

Feuille de route

1. **Setup & Configuration :** Installation Ollama + Llama 3.2, et création d'un projet Spring Boot
2. **La Mémoire :** Comprendre l'amnésie des LLM et implémenter une mémoire conversationnelle.
3. **Le Streaming :** Améliorer l'expérience utilisateur (UX) avec des réponses progressives.
4. **L'Agent Local :** Donner des "mains" (Outils) à l'IA via le *Function Calling*.
5. **Architecture MCP :** Déporter les outils vers un serveur distant standardisé (Concept Avancé).

2 Préparation de l'environnement

Avant d'écrire la moindre ligne de code Java, nous devons préparer le moteur d'intelligence artificielle.

2.1 Démarrage du modèle (Ollama)

Nous utilisons Ollama pour exécuter le modèle `llama3.2` localement. Cela garantit la confidentialité des données et la gratuité des inférences.

Listing 1 – Terminal : Démarrage du modèle

```
# 1. Telecharger et lancer le modele (si ce n'est pas deja fait)
ollama run llama3.2

# 2. Laissez ce terminal ouvert en arriere-plan.
```

2.2 Initialisation du Projet Spring Boot

Comme les TPs précédents étaient basés sur Spring Boot, nous restons dans l'écosystème Spring en utilisant Spring AI, le framework officiel pour intégrer l'intelligence artificielle dans les applications Java enterprise avec une API unifiée et une intégration native à Spring Boot.

Rendez-vous sur `start.spring.io` ou utilisez votre IDE :

— **Project :** Maven
— **Language :** Java 17+
— **Spring Boot :** 3.4.0 ou supérieur
— **Dépendances nécessaires :**

— `Spring Web` (pour créer les APIs REST).
— `Spring AI Ollama` (le connecteur pour parler au LLM).

2.2.1 Configuration (application.properties)

Configurez la connexion vers votre instance Ollama locale :

Listing 2 – Configuration Ollama

```
spring.application.name=demo-ai-agent
# Adresse par defaut d'Ollama
spring.ai.ollama.base-url=http://localhost:11434
# Le modele doit correspondre a celui lance dans le terminal
spring.ai.ollama.chat.options.model=llama3.2
```

3 Étape 1 : Le Chatbot et le problème de la Mémoire

Par défaut, les LLM sont **stateless** (sans état). Chaque requête est indépendante. Si vous ne renvoyez pas l'historique, l'IA oublie tout immédiatement.

3.1 Démonstration de l'Amnésie

Créez un `ChatController` simple sans gestion de mémoire.

```
@RestController
public class ChatController {
    private final ChatClient chatClient;

    public ChatController(ChatClient.Builder builder) {
        this.chatClient = builder.build();
    }

    @GetMapping("/chat")
    public String chat(@RequestParam String message) {
        return chatClient.prompt().user(message).call().content();
    }
}
```

Listing 3 – ChatController : Version Amnésique

Testez par vous-même

1. http://localhost:8080/chat?message=Jem'appelleHaythem
   → "Bonjour Haythem !"
2. http://localhost:8080/chat?message=Quelestmonnom?
   → "Je ne connais pas votre nom."

3.2 La Solution : Chat Memory Advisor

Spring AI fournit un concept d'**Advisor** (intercepteur) qui gère l'historique automatiquement. Il injecte l'historique de conversation dans le prompt système avant chaque appel.

```
public ChatController(ChatClient.Builder builder, ChatMemory chatMemory)
    {
    // InMemoryChatMemory stocke l'historique en RAM
    this.chatClient = builder
        .defaultAdvisors(MessageChatMemoryAdvisor.builder(chatMemory).
           build())
        .build();
}
```

Listing 4 – Ajout de la Mémoire Conversationnelle

Validation : Relancez l'application. L'IA se souvient maintenant de votre nom tant que l'application tourne (mémoire volatile).

4 Étape 2 : Amélioration UX avec le Streaming

L'appel `.call()` est bloquant : l'utilisateur attend la fin complète de la génération pour voir le texte. Pour une expérience fluide, utilisons `.stream()` et `Flux`.

```
import reactor.core.publisher.Flux;

@GetMapping("/stream")
public Flux<String> chatStream(@RequestParam String message) {
    return chatClient.prompt()
        .user(message)
        .stream() // Renvoie les tokens un par un
        .content();
}
```

Listing 5 – Endpoint Streaming

5 Étape 3 : L'Agent Local (Outils & Function Calling)

C'est ici que nous transformons notre simple Chatbot en véritable Agent.

5.1 Qu'est-ce qu'un Agent IA ?

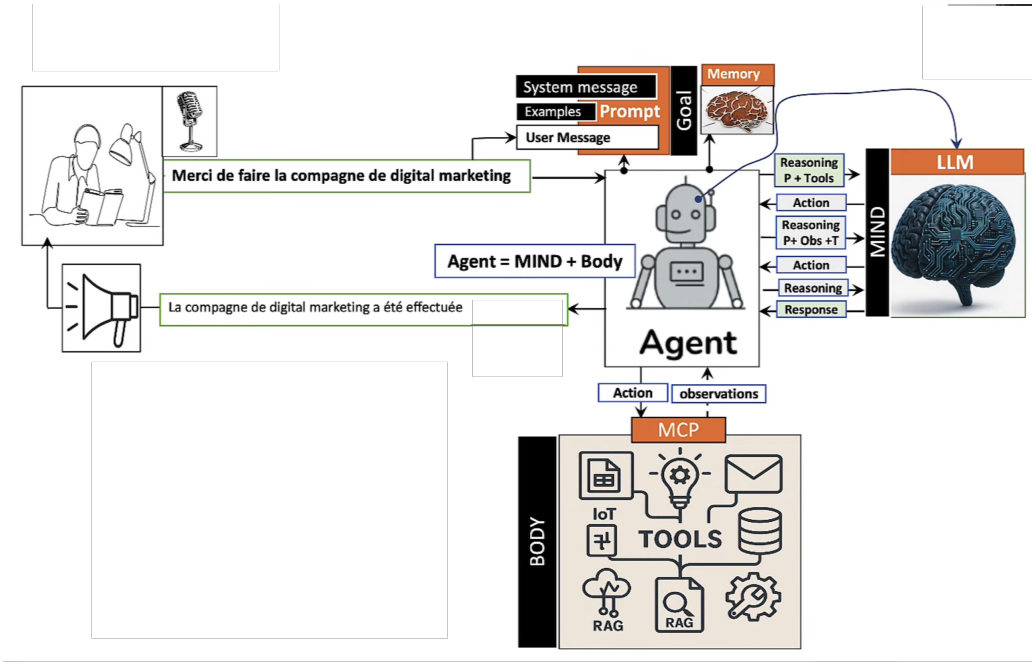

FIGURE 1 – Architecture Agentic AI : Agent = Mind + Body

Comme illustré sur la figure ci-dessus, un **Agent IA** n'est pas seulement un modèle de langage qui génère du texte. C'est un système autonome composé de deux parties principales :

— **Le Cerveau (MIND - LLM) :** C'est le moteur de raisonnement (Reasoning). Il analyse la demande de l'utilisateur, planifie les étapes nécessaires et décide s'il doit agir.

— **Le Corps (BODY - Tools) :** Ce sont les capacités d'action de l'agent. Contrairement à un chatbot classique qui est enfermé dans sa boîte de texte, l'agent possède des "mains" (API, Base de données, Calculatrice) pour interagir avec le monde extérieur.

Le pattern ReAct (Reasoning + Acting) : L'agent fonctionne en boucle :

1. Il **Raisonne** sur la question.
2. Il décide d'une **Action** (appeler un outil).
3. Il reçoit l'**Observation** (le résultat de l'outil).
4. Il formule la réponse finale.

5.2 Création des Outils (@Tool)

Créez le package `tools` et la classe `ProduitTools`. L'annotation `@Tool` est cruciale : la description en langage naturel guide l'IA pour savoir **quand** utiliser cet outil.

```java
package com.isitcom.demospringia.tools;

import org.springframework.ai.tool.annotation.Tool;
import org.springframework.ai.tool.annotation.ToolParam;
import org.springframework.stereotype.Component;
import java.util.List;

record Produit(int id, String nom, double prix){}

@Component
public class ProduitTools {

    @Tool(name = "getproduct", description = "Recupere un produit via
        son ID")
    public Produit getProduct(@ToolParam(description = "id du produit")
        int id) {
         return new Produit(id, "PC Gamer Ultra", 4000);
    }

    @Tool(name = "getallproducts", description = "Recupere la liste de
        tous les produits")
    public List<Produit> getListProduit(){
         return List.of(
             new Produit(1, "PC Gamer Ultra", 4000),
             new Produit(2, "Clavier Mecanique", 40)
         );
    }
}
```

Listing 6 – ProduitTools.java

5.3 L'Agent Complet

Nous combinons **Mémoire** + **Outils** + **System Prompt** (pour définir la personnalité).

```java
@RestController
public class ChatController {
    private final ChatClient chatClient;

    public ChatController(ChatClient.Builder builder,
                          ProduitTools produitTools,
                          ChatMemory chatMemory) {
        this.chatClient = builder
            // 1. SYSTEM PROMPT : Definition du role
            .defaultSystem("""
            Tu es un assistant qui ne repond qu'aux questions sur
                les produits.
            - Utilise seulement tes outils pour repondre.
            - Si la question ne concerne pas les produits,
              repond exactement : "Je ne sais pas"
            - Ne donne jamais d'informations inventees.
            """)
            // 2. TOOLS : Injection des outils
            .defaultTools(produitTools)
```

```
19              // 3. MEMORY : Conservation du contexte
20              .defaultAdvisors(MessageChatMemoryAdvisor.builder(
                    chatMemory).build())
21              .build();
22      }
23
24      @GetMapping("/agent")
25      public Flux<String> ask(@RequestParam String message) {
26          return this.chatClient.prompt().user(message).stream().
                content();
27      }
28  }
```

Listing 7 – ChatController.java : Version Agent Final

6 Étape 4 : Architecture Distribuée (MCP)

Dans les étapes précédentes, les outils étaient dans la même application que l'agent. Dans cette étape, les outils sont déplacés dans un **serveur distant** : c'est une approche plus modulaire.

6.1 Qu'est-ce que le MCP (Model Context Protocol) ?

Le **MCP** est un protocole qui permet à une application d'IA de se connecter à des **outils externes** (API, base de données, services, fichiers). L'objectif est de rendre cette connexion **standard** et **réutilisable**.

6.1.1 Pourquoi MCP est utile ?

Sans MCP, chaque application doit intégrer chaque outil "à la main". Quand on ajoute un nouvel outil, on recode souvent une partie de l'intégration.

Avec MCP, l'agent peut utiliser des outils de façon plus simple :

— Un **serveur MCP** publie des outils (ex : `getProducts`, `searchProducts`).

— Un **client MCP** (dans l'agent) se connecte au serveur et **découvre** les outils disponibles.

6.1.2 MCP Server vs MCP Client

— **MCP Server** : il **exécute** les outils et renvoie les résultats.

— **MCP Client** : il est dans l'agent et **appelle** les outils du serveur.

6.1.3 MCP vs APIs REST : est-ce un remplacement ?

Non. MCP ne remplace pas REST. REST reste très utilisé pour construire des services classiques (web, mobile, intégrations métier).

Idée importante : MCP est une couche plus adaptée aux **agents IA**. Au lieu d'appeler des endpoints REST un par un "à la main", l'agent utilise un serveur MCP qui décrit les outils (nom, description, paramètres) et standardise leur appel.

— **REST** : on appelle des endpoints (il faut connaître l'URL, les paramètres, la doc).

— **MCP** : l'agent se connecte, **découvre** les outils, puis les appelle.

6.1.4 Les transports (comment ça communique ?)

Le protocole MCP définit **comment** les données sont échangées (JSON-RPC 2.0), mais il laisse le choix du **canal de communication**. On appelle cela le "transport". Spring AI MCP supporte trois modes :

1. stdio (Standard Input/Output)

— **Principe** : Communication via les flux système (stdin/stdout)

— **Usage** : Serveur MCP lancé comme processus fils sur la **même machine** que l'agent

— **Avantage** : Performance maximale (pas de réseau, pas de sérialisation HTTP)

— **Limitation** : Impossible de communiquer à distance

— **Cas d'usage** : Développement local, outils système

2. SSE - Server-Sent Events (Legacy)

- **Principe** : Utilise deux canaux HTTP séparés
  - `GET /sse` : le client écoute les événements du serveur (flux continu)
  - `POST /messages` : le client envoie ses requêtes
- **Usage** : Communication **à distance** (serveur sur une autre machine)
- **Statut** : Ancienne approche, maintenue pour compatibilité avec serveurs MCP existants
- **Inconvénient** : Deux connexions à gérer

3. Streamable HTTP (Moderne - Recommandé)

- **Principe** : Un seul endpoint HTTP bidirectionnel
  - `POST /mcp` : connexion unique pour tous les échanges
- **Usage** : Communication **à distance** (architecture distribuée)
- **Avantages** :
  - Architecture simple (un seul endpoint)
  - Bidirectionnel (client et serveur peuvent initier des messages)
  - Multiplexage (plusieurs requêtes simultanées sur la même connexion)
  - Performance optimisée
- **Statut** : Protocole moderne, recommandé pour tous les nouveaux projets

Choix pour ce TP

Dans ce TP, on utilise Streamable HTTP pour plusieurs raisons :

- Architecture distribuée réaliste (serveur sur port 9090, agent sur port 8080)
- Mode recommandé par Spring AI pour la production
- Simple à configurer (une seule URL à définir)
- Facile à tester avec MCP Inspector

6.1.5 Comparaison visuelle

Critère	**stdio**	**SSE**	**Streamable**
Localisation	Même machine	À distance	À distance
Nombre d'endpoints	stdin/stdout	2 (GET + POST)	1 (POST)
Bidirectionnel	Oui	Non (2 canaux)	Oui (natif)
Performance			
Complexité config	Simple	Moyenne	Simple
Production	Rare	Legacy	Recommandé

TABLE 1 – Comparaison des modes de transport MCP

6.1.6 Schéma de Communication

Listing 8 – Architecture Streamable (TP)

```
AGENT (Client)                        SERVEUR MCP
Port 8080                             Port 9090
    |                                      |
    | POST /mcp                            |
    |------------------------------------->|
    |                                      |
    | <==============================>     |
    |   CONNEXION BIDIRECTIONNELLE         |
    |   (requetes et reponses)             |
    |                                      |
    | Requete: tools/call                  |
    |------------------------------------->|
    |                                      |
    | Reponse: tool_result                 |
    |<-------------------------------------|
    |                                      |

UN SEUL ENDPOINT - COMMUNICATION DANS LES DEUX SENS
```

6.2 Architecture Modulaire du Projet

Pour une meilleure organisation et séparation des responsabilités, nous allons ajouter un module serveur MCP à notre projet existant.

6.2.1 Création du Module Serveur MCP

Exercice pratique

Objectif : Créer un nouveau module Spring Boot pour le serveur MCP dans votre projet existant.

Instructions :

1. Dans votre IDE, faites clic droit sur le projet parent
2. Sélectionnez `New > Module`
3. Configurez le nouveau module :
   - Type : `Spring Boot Module`
   - Name : `mcp-tools-server`
   - Parent : Votre projet existant
4. Ajoutez les dépendances suivantes :
   - `Spring Web`
   - `Spring AI MCP Server WebMVC Starter`

6.3 Module MCP Tools Server

6.3.1 Configuration Maven du Serveur

```
1 <properties>
```

```
    <spring-ai.version>1.1.1</spring-ai.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.ai</groupId>
        <artifactId>spring-ai-starter-mcp-server-webmvc</artifactId>
    </dependency>
</dependencies>
```

Listing 9 – mcp-tools-server/pom.xml (extrait)

Vérification Spring AI

Vérifiez la version Spring AI dans votre `pom.xml` :

```
<spring-ai.version>1.1.1</spring-ai.version>
```

Pourquoi ? Spring AI version 1.1.1 garantit la compatibilité MCP et l'auto-configuration des starters.

6.3.2 Fichier de Configuration Spring Boot du Serveur

Listing 10 – mcp-tools-server/src/main/resources/application.properties

```
server.port=9090
spring.application.name=mcp-tools-server

# Transport MCP (Streamable HTTP)
spring.ai.mcp.server.protocol=streamable
```

6.3.3 Implémentation des Outils MCP

```
package com.isitcom.mcpserver.tools;

import org.springframework.ai.mcp.server.annotation.McpTool;
import org.springframework.ai.mcp.server.annotation.McpToolParam;
import org.springframework.stereotype.Component;

import java.util.List;

record Produit(int id, String nom, double prix) {}

@Component
public class ProduitTools {

    @McpTool(name = "getProduct",
            description = "Recupere un produit specifique via son
               identifiant")
    public Produit getProduct(
```

```
            @McpToolParam(description = "Identifiant unique du produit",
                required = true) int id) {
        return new Produit(id, "PC Gamer Ultra", 4000.0);
    }

    @McpTool(name = "getProducts",
            description = "Recupere la liste complete des produits
               disponibles")
    public List<Produit> getProducts() {
        return List.of(
            new Produit(1, "pc portable", 5000.0),
            new Produit(2, "clavier", 100.0),
            new Produit(3, "souris", 50.0)
        );
    }

    @McpTool(name = "searchProducts",
            description = "Recherche des produits par mot-cle dans leur
               nom")
    public List<Produit> searchProducts(
            @McpToolParam(description = "Mot-cle de recherche", required
                = true) String keyword) {
        return getProducts().stream()
            .filter(p -> p.nom().toLowerCase().contains(keyword.
               toLowerCase()))
            .toList();
    }
}
```

Listing 11 – ProduitTools.java - Outils MCP

Auto-Configuration Spring AI (Serveur)

C'est tout ! Spring AI auto-configure automatiquement :

- La détection des beans annotés `@McpTool`
- L'enregistrement des outils dans le serveur MCP
- L'exposition du endpoint Streamable utilisé dans ce TP : `POST /mcp`
- La sérialisation/désérialisation JSON-RPC 2.0

6.4 Test avec MCP Inspector

Avant de connecter notre agent au serveur MCP, nous utilisons **MCP Inspector** pour tester et déboguer le serveur.

6.4.1 Installation et Lancement

Listing 12 – Installation / Lancement MCP Inspector

```
# Option 1 : Utilisation directe avec npx
npx @modelcontextprotocol/inspector
```

6.4.2 Configuration de l'Inspector

— **Transport Type** : `Streamable HTTP`

— **URL** :

```
http://localhost:9090/mcp
```

6.5 Module MCP Agent Client

Après avoir validé le serveur avec MCP Inspector, nous configurons le projet agent pour consommer les outils MCP.

6.5.1 Configuration Maven du Client

Exercice : Dépendance MCP Client

Objectif : Ajouter la dépendance MCP Client au projet agent existant et vérifier la version Spring AI.

1. Ouvrez le fichier `pom.xml` du projet agent
2. Vérifiez que `<spring-ai.version>` est **1.1.1**
3. Ajoutez la dépendance `spring-ai-starter-mcp-client`

```
<properties>
    <spring-ai.version>1.1.1</spring-ai.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.ai</groupId>
        <artifactId>spring-ai-starter-mcp-client</artifactId>
    </dependency>
</dependencies>
```

Listing 13 – mcp-agent-client/pom.xml (extrait)

6.5.2 Fichier de Configuration Spring Boot de l'Agent

Ajout de l'URL du serveur MCP

Dans le fichier `mcp-agent-client/src/main/resources/application.properties`, ajoutez la ligne suivante :

```
spring.ai.mcp.client.streamable-http.connections.mcpproduct.
   url=http://localhost:9090/mcp
```

Important : Utilisez la même URL que celle testée dans MCP Inspector.

6.5.3 Modification du Contrôleur pour utiliser les Outils MCP

Adaptation du ChatController

Important : La connexion MCP ne suffit pas : il faut injecter les callbacks MCP dans le `ChatClient`.

Modifications :

1. Injecter `SyncMcpToolCallbackProvider` dans le constructeur
2. Remplacer `.defaultTools(...)` par `.defaultToolCallbacks(...)`

```java
package com.isitcom.agent.controller;

import org.springframework.ai.chat.client.ChatClient;
import org.springframework.ai.chat.client.advisor.
    MessageChatMemoryAdvisor;
import org.springframework.ai.chat.memory.ChatMemory;
import org.springframework.ai.mcp.SyncMcpToolCallbackProvider;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;

@RestController
public class ChatController {

    private final ChatClient chatClient;

    public ChatController(ChatClient.Builder builder,
                          ChatMemory memory,
                          SyncMcpToolCallbackProvider tools) {

        this.chatClient = builder
                .defaultSystem("""
                    Tu es un assistant qui ne repond qu'aux questions
                       sur les produits.
                    - Utilise seulement tes outils pour repondre.
                    - Si la question ne concerne pas les produits,
                      repond exactement : "Je ne sais pas"
                    - Ne donne jamais d'informations inventees.
                    """)
                .defaultAdvisors(MessageChatMemoryAdvisor.builder(memory
                   ).build())
                .defaultToolCallbacks(tools)
                .build();
    }

    @GetMapping("/ask")
    public Flux<String> ask(@RequestParam String message) {
        return chatClient.prompt()
                .user(message)
                .stream()
                .content();
    }
}
```

Listing 14 – ChatController.java (agent)

6.5.4 Validation et Test de la Connexion MCP

Réexécution du projet agent

Ordre de démarrage recommandé :

1. Démarrez Ollama :

```
ollama run llama3.2
```

2. Démarrez le serveur MCP

3. Démarrez l'agent

Vérification attendue : l'agent doit découvrir les outils MCP et pouvoir les appeler lors des questions sur les produits.