# Problématique

Dans le développement logiciel, la détection et la correction des erreurs représentent un défi majeur, notamment en raison de la diversité des sources d'anomalies (logs, stack traces, captures d'écran, retours utilisateurs, etc.) et de la complexité croissante des applications. Les méthodes traditionnelles de débogage reposent souvent sur une analyse manuelle, ce qui est chronophage et sujet à des erreurs humaines. De plus, les solutions existantes peinent à offrir une approche unifiée et intelligente pour interpréter ces anomalies et proposer des correctifs pertinents.

L'absence d'un système capable de :

— Comprendre multimodalement les erreurs (texte, images, logs structurés).

— Proposer des corrections automatisées en s'appuyant sur des modèles de langage (LLMs) et des techniques de RAG (Retrieval-Augmented Generation).

— S'intégrer facilement dans des environnements DevOps existants.

rend ce projet particulièrement pertinent. Comment concevoir une application IA capable d'analyser efficacement ces différentes sources d'erreurs et de générer des solutions adaptées ?

# Objectifs

Ce projet vise à développer une application intelligente et modulaire permettant de détecter et corriger automatiquement les anomalies logicielles. Les objectifs spécifiques incluent :

1. Analyse Multimodale des Erreurs :
   — Implémenter un système capable d'interpréter des données hétérogènes (stack traces, logs texte, captures d'écran, etc.).
   — Utiliser des techniques de RAG pour enrichir les requêtes avec une base de connaissances (documentation technique, résolutions d'erreurs courantes).
2. Génération Automatique de Correctifs :
   — Exploiter des LLMs (via Ollama) pour suggérer des corrections précises et contextualisées.
3. Intégration et Scalabilité :
   — Développer un backend Spring Boot flexible, couplé à LangChain4J pour orchestrer les appels IA.
   — Permettre une extension future via des connecteurs pour différents outils de monitoring.
4. Optimisation et Évaluation :
   — Mesurer l'efficacité du système via des métriques de précision (taux de détection, pertinence des correctifs).
   — Comparer avec des solutions existantes pour démontrer l'amélioration apportée.

# Chapitre 1

## 1.1 Langages, Frameworks et outils utilisés

### 1.1.1 Java


FIGURE 1.1 – Logo de Java

Java est un langage de programmation orienté objet, robuste et multiplateforme, largement utilisé dans le développement d'applications d'entreprise. Sa forte typographie, sa gestion automatique de la mémoire (via le garbage collector) et son écosystème riche (bibliothèques, frameworks) en font un choix idéal pour les systèmes backend complexes.

### 1.1.2 Spring

FIGURE 1.2 – Logo de Spring

Spring est un framework modulaire pour Java, simplifiant le développement d'applications grâce à l'inversion de contrôle (IoC) et la programmation orientée aspect (AOP).

### 1.1.3 Spring Boot

FIGURE 1.3 – Logo de Spring Boot

Spring Boot étend Spring en fournissant des configurations automatiques, un serveur embarqué (Tomcat, Netty) et des outils clés en main (Spring Data, Spring Security), permettant de créer des applications standalone rapidement.

### 1.1.4 Maven


FIGURE 1.4 – Logo de Maven

Outil de build automatisé pour projets Java, qui gère Les dépendances (téléchargement auto), le packaging (JAR/WAR), et les cycles de compilation/test.

### 1.1.5 LangChain4j

FIGURE 1.5 – Logo de LangChain4j

LangChain4J est une bibliothèque Java inspirée de LangChain (Python), conçue pour intégrer facilement des LLMs (Modèles de Langage) dans des applications. Elle offre des abstractions pour la gestion des prompts, le RAG, les appels aux modèles (OpenAI, Ollama, etc.), et la connexion à des bases de données vectorielles.

### 1.1.6 Ollama

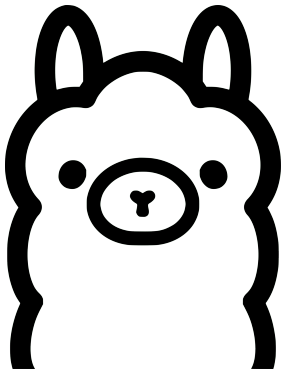

FIGURE 1.6 – Logo de Ollama

Ollama est un outil open-source permettant d'exécuter localement des LLMs (comme Llama 3, Mistral, Gemma) sans dépendre d'une API externe. Il est idéal pour prototyper des solutions IA offline, contrôler les coûts et la confidentialité des données, et personnaliser finement les modèles via des modelfiles.

### 1.1.7 IntelliJ IDEA

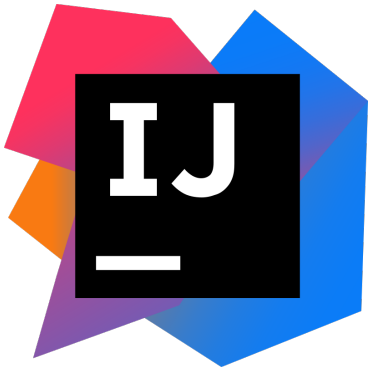

FIGURE 1.7 – Logo de IntelliJ IDEA

IntelliJ IDEA est un IDE puissant pour Java/Kotlin, développé par JetBrains. Ses avantages incluent une analyse intelligente du code (suggestions, détection d'erreurs), une intégration native avec Spring Boot et Maven/Gradle, des outils pour le débogage, le profiling et les tests, et des extensions pour l'IA (ex : GitHub Copilot).

## 1.2 Premier prototype

### 1.2.1 Introduction

Pour atteindre cet objectif ambitieux, nous avons suivi une approche incrémentale. Dans un premier temps, un prototype fonctionnel a été réalisé afin de valider les fondements techniques du projet. Ce prototype est une application basée sur un agent intelligent exploitant une architecture RAG (Retrieval-Augmented Generation), capable de répondre aux questions de l'utilisateur à partir d'un document texte ou PDF fourni. Cette première version a permis de :

— comprendre le fonctionnement du framework LangChain4j ;
— tester l'intégration avec le LLM Ollama ;
— valider le concept de récupération de contexte à partir de documents externes.

### 1.2.2 Fonctionnement d'un agent AI dans LangChain4j

Dans LangChain4j, un Agent AI orchestre les interactions entre un ChatLanguageModel (abstraction des LLMs comme OpenAI/Gemini/Ollama) et un ChatMemoryProvider (gestion de l'historique conversationnel). L'Agent formate les requêtes, intègre la mémoire contextuelle, et peut utiliser des outils externes, tandis que le LLM génère les réponses. Cette architecture modulaire permet une intégration flexible avec différents modèles et systèmes de stockage, tout en maintenant un état conversationnel cohérent.

Le diagramme de séquences suivant décrit ce processus :

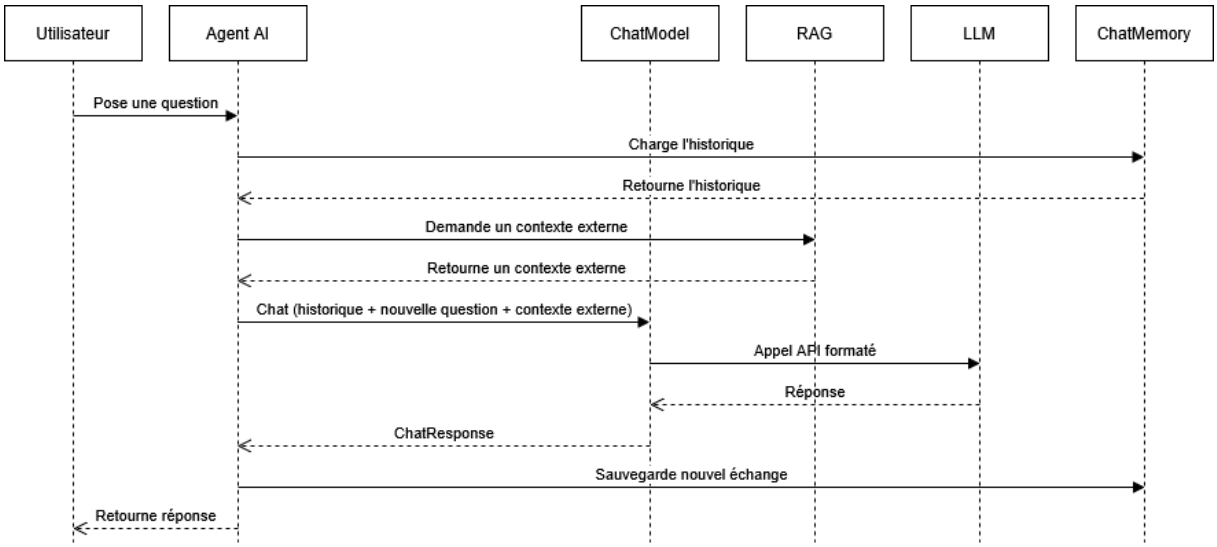


FIGURE 1.8 – Diagramme de séquences décrivant le fonctionnement d'un agent AI

Le RAG combine deux phases clés pour améliorer la génération de réponses par un LLM : la rétroinformation (retrieval) et la génération contextuelle. Dans ce prototype, le système RAG :

- Charge d'abord un fichier source (texte, PDF, etc.).
- Utilise un DocumentPaser pour extraire les données brutes, et les découper en 'chunks'.
- Utilise un EmbeddingModel qui convertit ces chunks en vecteurs d'embedding (représentations numériques sémantiques).
- Utilise un EmbeddingStoreIngestor pour stocker ces vecteurs dans un EmbeddingStore.
- Utilise un Retriever qui interroge l'EmbeddingStore pour trouver les documents les plus pertinents par rapport à la question de l'utilisateur.

Le diagramme de séquences suivant détaille ces étapes :

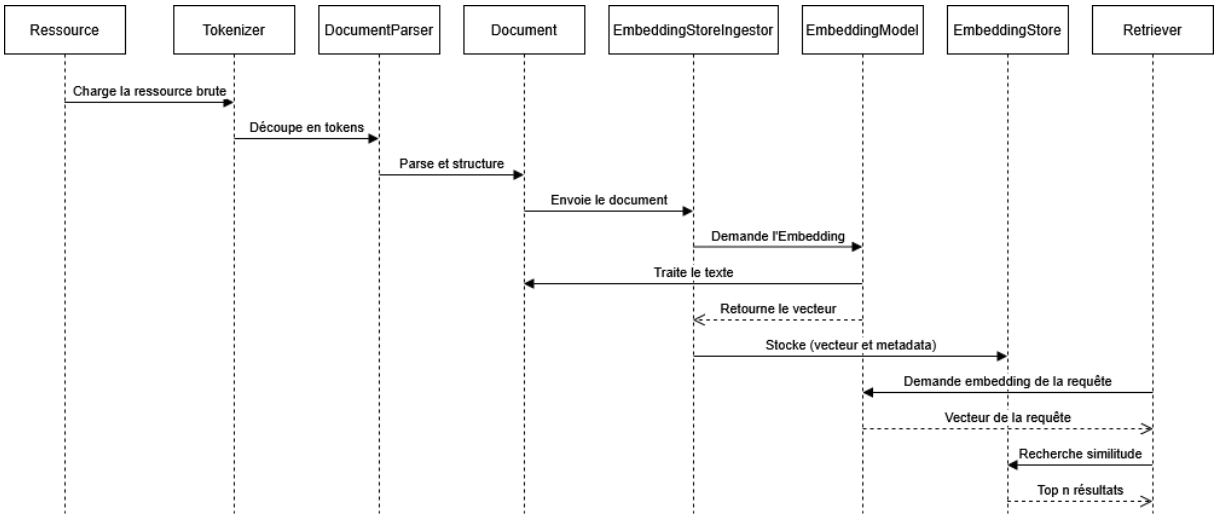


FIGURE 1.9 – Diagramme de séquences décrivant le fonctionnement du RAG

Pour tester ce prototype, nous avons fourni un fichier PDF, son contenu est une lettre de recommandation pour une étudiante appelée Nour, on a ensuite envoyé une question à propos de cette étudiante à l'agent AI, en utilisant un contrôleur web :

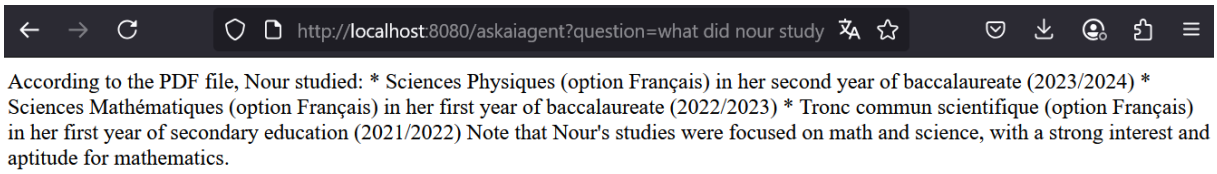


FIGURE 1.10 – Test du RAG