

# Dédicace

*Je dédie ce travail à :*

*Mes chers parents, mon frère,  
pour leur sacrifice et leur motivation.*

*À mon épouse et à ma petite fille,  
qui m'apportent chaque jour la force pour avancer.*

*À Mes amis  
pour leur contribution, qui a apporté une valeur  
ajoutée.*

*À toute personne  
ayant contribué à ma formation.*

# Remerciement

Au début, il m'est agréable d'exprimer ma reconnaissance à toute personne dont l'intervention au cours de ce stage a favorisé son aboutissement.

Je tiens à remercier vivement mes encadrants, Monsieur BERAHIOU RADWANE et Monsieur MOUHIB IMAD, pour leur temps précieux et le partage de leur expertise quotidienne. Leur confiance m'a permis de développer mes compétences de manière significative.

Un grand merci à toute l'équipe d'ALGOLUS pour m'avoir offert cette opportunité de stage dans un environnement professionnel stimulant.

Je tiens à exprimer ma sincère gratitude aux membres du jury pour le temps qu'ils ont consacré à l'évaluation de mon travail.

Un remerciement particulier à DOUHI TAOUFIK, ARRHIOUI KARIM, ARRHIOUI ANASS et BERHIL MOHAMMED pour leurs précieux conseils.

Je remercie le corps professoral et administratif de l'ÉCOLE DES HAUTES ÉTUDES D'INGÉNIERIE - OUJDA pour la qualité de la formation reçue.

# Liste des abréviations

Abréviation	Désignation
API	Application Programming Interface
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JPA	Java Persistence API
LLM	Large Language Model
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation
SARL	Société À Responsabilité Limitée
SQL	Structured Query Language
SRP	Single Responsibility Principle
UML	Unified Modeling Language
URL	Uniform Resource Locator

# Résumé

Ce projet présente une solution innovante d'assistance intelligente au débogage logiciel, combinant analyse automatique et intelligence artificielle. Face à la complexité croissante des erreurs logicielles (stacktraces, logs, captures d'écran, captures vidéo), nous avons développé un système unifié exploitant les LLMs (Modèles de Langage) et l'architecture RAG (Retrieval-Augmented Generation) pour diagnostiquer les anomalies et proposer des corrections contextuelles.

L'outil implémenté avec Spring Boot et LangChain4J permet l'analyse multimodale des rapports d'erreur, l'identification des causes avec 78% de précision, la génération de solutions pertinentes dans 70% de cas, et une réduction de 35% du temps de résolution.

Doté d'une interface web intuitive et d'une API REST, le système a démontré son efficacité sur les exceptions courantes. Cette approche ouvre de nouvelles possibilités pour l'assistance aux développeurs.

**Mots-clés :** IA générative, LLM, Spring Boot, RAG, débogage automatique, analyse multimodale.

# Abstract

This project introduces an innovative AI-powered debugging assistant that automates software error analysis. Addressing the growing complexity of software anomalies (stack traces, logs, screenshots, videos), we developed a unified system leveraging LLMs and RAG architecture to diagnose issues and suggest contextual fixes.

The Spring Boot-based solution features multimodal error report analysis, 78% accuracy in root cause identification, relevant solution generation with 65% success rate, 35% faster resolution time.

With its intuitive web interface and REST API, the tool has proven effective for common exceptions. This approach offers new possibilities for developer assistance while maintaining data privacy through local model deployment.

**Keywords :** Generative AI, LLM, Spring Boot, RAG, automated debugging, multimodal analysis.

# Table des matières

Dédicace	i
Remerciement	ii
Liste des abréviations	iii
Résumé	iv
Abstract	v
Table des figures	ix
Liste des tableaux	xi
Introduction générale	1
<b>1 Contexte du Projet</b>	<b>2</b>
1.1 Introduction . . . . .	2
1.2 Entreprise d'accueil . . . . .	2
1.2.1 Description de l'entreprise . . . . .	3
1.2.2 Fiche technique de l'entreprise . . . . .	3
1.2.3 Organigramme de l'entreprise . . . . .	4
1.3 Description des besoins . . . . .	5
1.3.1 Problème . . . . .	5
1.3.2 Les besoins fonctionnels . . . . .	5
1.3.3 Les besoins non fonctionnels . . . . .	6

1.3.4	Solutions envisagées . . . . .	7
1.4	Conclusion . . . . .	8
<b>2</b>	<b>Analyse fonctionnelle et modélisation</b>	<b>9</b>
2.1	Introduction . . . . .	9
2.2	Importance de l'analyse fonctionnelle . . . . .	9
2.3	Unified Modeling Language . . . . .	10
2.4	Architecture hybride : LLM et RAG . . . . .	10
2.5	Diagramme de cas d'utilisation . . . . .	14
2.5.1	Acteurs . . . . .	15
2.5.2	Notre diagramme de cas d'utilisations . . . . .	15
2.6	Description textuelle . . . . .	16
2.7	Diagrammes de séquences . . . . .	19
2.8	Conclusion . . . . .	20
<b>3</b>	<b>Réalisation</b>	<b>21</b>
3.1	Introduction . . . . .	21
3.2	Méthode de gestion adoptée : SCRUM . . . . .	21
3.2.1	Scrum à la théorie . . . . .	21
3.2.2	Scrum à la pratique . . . . .	23
3.3	Stack technique . . . . .	24
3.3.1	Langages . . . . .	24
3.3.2	Frameworks . . . . .	24
3.3.3	Bibliothèques . . . . .	25
3.3.4	Systèmes de gestion de bases de données . . . . .	25
3.3.5	Outils et environnement . . . . .	26
3.4	Architecture technique du projet . . . . .	27
3.5	Bonnes pratiques appliquées du développement Java . . . . .	31
3.5.1	L'importance d'une nomenclature claire et cohérente . . . . .	31
3.5.2	Utilisation de Bibliothèques Matures : Apache Commons . . . . .	32

3.5.3	Approches de parcours de collections en Java . . . . .	33
3.5.4	Importance de la factorisation du code . . . . .	36
3.5.5	Utilisation d'une classe de configuration . . . . .	37
3.6	Interfaces graphiques réalisées . . . . .	39
3.6.1	Prototype exploratoire . . . . .	40
3.6.2	Version livrable actuelle . . . . .	40
3.7	Conclusion . . . . .	54
<b>4</b>	<b>Acquis du Projet de Fin d'Études</b>	<b>56</b>
4.1	Volet technique . . . . .	56
4.2	Volet gestion de projet . . . . .	57
	<b>Conclusion générale et perspectives</b>	<b>58</b>
	Conclusion générale . . . . .	58
	Perspectives . . . . .	59
	<b>Bibliographie</b>	<b>60</b>
	<b>Webographie</b>	<b>61</b>
	<b>Annexes</b>	<b>62</b>
	Evaluation des objectifs . . . . .	62
	Méthodologie d'évaluation . . . . .	62
	Résultats obtenus . . . . .	62
	Benchmark de LLMs Multimodaux . . . . .	63



# Table des figures

1.1	<i>Logo de Algolus</i> . . . . .	2
1.2	<i>Organigramme de l'entreprise Algolus</i> . . . . .	4
2.1	<i>Diagramme décrivant le fonctionnement d'un RAG</i> . . . . .	14
2.2	<i>Diagramme de cas d'utilisation</i> . . . . .	16
2.3	<i>Diagramme de séquences décrivant le fonctionnement d'un agent AI</i> . . . .	19
2.4	<i>Diagramme de séquences décrivant le fonctionnement d'un RAG</i> . . . . .	20
3.1	<i>Gestion de projet avec SCRUM</i> . . . . .	23
3.2	<i>Diagramme d'architecture technique</i> . . . . .	30
3.3	<i>Implémentation sans utiliser Apache Commons</i> . . . . .	32
3.4	<i>Implémentation en utilisant Apache Commons</i> . . . . .	33
3.5	<i>Implémentation sans utiliser les Streams</i> . . . . .	35
3.6	<i>Implémentation en utilisant les Streams</i> . . . . .	35
3.7	<i>Exemple de code répétitif avant factorisation</i> . . . . .	36
3.8	<i>Code refactorisé avec méthode utilitaire</i> . . . . .	37
3.9	<i>Un morceau de la classe de configuration</i> . . . . .	38
3.10	<i>Utilisation de la classe de configuration</i> . . . . .	39
3.11	<i>Test du premier prototype</i> . . . . .	40
3.12	<i>Structure du projet Java</i> . . . . .	42
3.13	<i>Morceau du contenu du fichier application.properties</i> . . . . .	44
3.14	<i>Interface Java de l'agent AI</i> . . . . .	45
3.15	<i>Interface de l'application de test</i> . . . . .	47

3.16	<i>Résultat d'exécution de l'application de test . . . . .</i>	48
3.17	<i>Résultat d'une analyse d'erreur réussie . . . . .</i>	49
3.18	<i>Interface Swagger UI . . . . .</i>	51
3.19	<i>Image trop volumineuse . . . . .</i>	52
3.20	<i>Type d'images non supporté . . . . .</i>	53
3.21	<i>Upload de vidéo . . . . .</i>	54
4.1	<i>Comparaison des temps de réponse moyens . . . . .</i>	64
4.2	<i>Précision des modèles testés . . . . .</i>	65

# Liste des tableaux

1.1	<i>Fiche technique de l'entreprise Algolus</i>	3
3.1	<i>Tableau des acteurs SCRUM</i>	23
3.2	<i>Comparaison des méthodes de parcours de collections en Java</i>	34
4.1	<i>Comparaison des LLM multimodaux</i>	63

# Introduction générale

Dans le cadre de la formation en Génie Informatique à l'École des Hautes Études d'Ingénierie d'Oujda (EHEIO), la réalisation d'un Projet de Fin d'Études (PFE) constitue une étape essentielle. Elle marque la transition entre le parcours académique et le monde professionnel, et vise à mettre en pratique les connaissances acquises au cours des années de formation. Dans un contexte où les technologies de l'information évoluent rapidement, ce projet s'inscrit pleinement dans une logique d'adaptation aux exigences du secteur.

Le stage réalisé s'inscrit dans cette dynamique et a pour principal objectif de confronter l'étudiant à des problématiques concrètes, tout en développant ses compétences techniques, organisationnelles et relationnelles. Plus spécifiquement, il s'est déroulé dans une entreprise spécialisée en solutions logicielles, fonctionnant selon la méthodologie Agile Scrum. Cette expérience a permis de s'initier aux pratiques de gestion de projet modernes, à travers des itérations, des revues régulières et une forte collaboration en équipe. La mission confiée s'articule autour du développement d'une application innovante intégrant des technologies émergentes telles que les modèles de langage (LLM) et la génération augmentée par récupération (RAG).

Le présent rapport est structuré en plusieurs chapitres. Le premier chapitre présente le contexte général du projet la définition des besoins fonctionnels et techniques, le deuxième chapitre, couvre l'analyse fonctionnelle et la conception du projet, le troisième chapitre détaille la phase de réalisation, mettant en lumière les choix technologiques et architecturaux retenus, le quatrième chapitre met en lumière les compétences acquises du PFE, et finalement le cinquième chapitre conclut et cite les perspectives d'amélioration du projet.

# Chapitre 1

## Contexte du Projet

### 1.1 Introduction

Dans ce chapitre, nous commencerons par présenter l'entreprise d'accueil, avant de procéder à une description détaillée des besoins du projet. Cette description comprendra l'identification du problème posé, l'analyse des besoins fonctionnels et non fonctionnels, ainsi que les solutions envisagées pour y répondre.

### 1.2 Entreprise d'accueil

Ce stage a été réalisé au sein de l'entreprise Algolus, située à Oujda, spécialisée dans les solutions innovantes en intelligence artificielle. Démarré le 3 Mars 2025, il m'a permis de m'immerger dans un environnement professionnel exigeant, où j'ai pu collaborer avec des experts en IA et en ingénierie logicielle.



FIGURE 1.1: *Logo de Algolus*

### 1.2.1 Description de l'entreprise

Algolus est une agence web marocaine, créée en 2020, spécialisée dans la conception et le développement de solutions informatiques adaptées aux besoins des clients, en leur offrant une communication en ligne efficace et sur mesure.

Ses prestations incluent :

- Création et gestion de sites web (dynamiques, statiques, e-commerce, CMS)
- Développement d'applications web (mode hybride)
- Stratégie digitale complète : infographie, publicité en ligne, marketing digital, community management, E-réputation.

### 1.2.2 Fiche technique de l'entreprise

Le tableau 1.1 récapitule la fiche technique de l'entreprise Algolus :

TABLE 1.1: *Fiche technique de l'entreprise Algolus*

<b>Dénomination sociale</b>	Algolus
<b>Date de création</b>	07/10/2020
<b>Forme juridique</b>	SARL
<b>Capital</b>	100.000 Dh
<b>Chiffre d'affaires</b>	Indisponible
<b>Activités</b>	Développement informatique et marketing digital
<b>Effectif</b>	10
<b>Dirigeant</b>	Radwane BERAHIOUI
<b>Coordonnées</b>	+212 6644 35967 Redwan.Berahioui@algolus.ma www.algolus.ma IMMEUBLE OUASSIM, Bd Mohammed VI, Oujda 60000

### 1.2.3 Organigramme de l'entreprise

La figure 1.2 présente l'organigramme de l'entreprise Algolus :

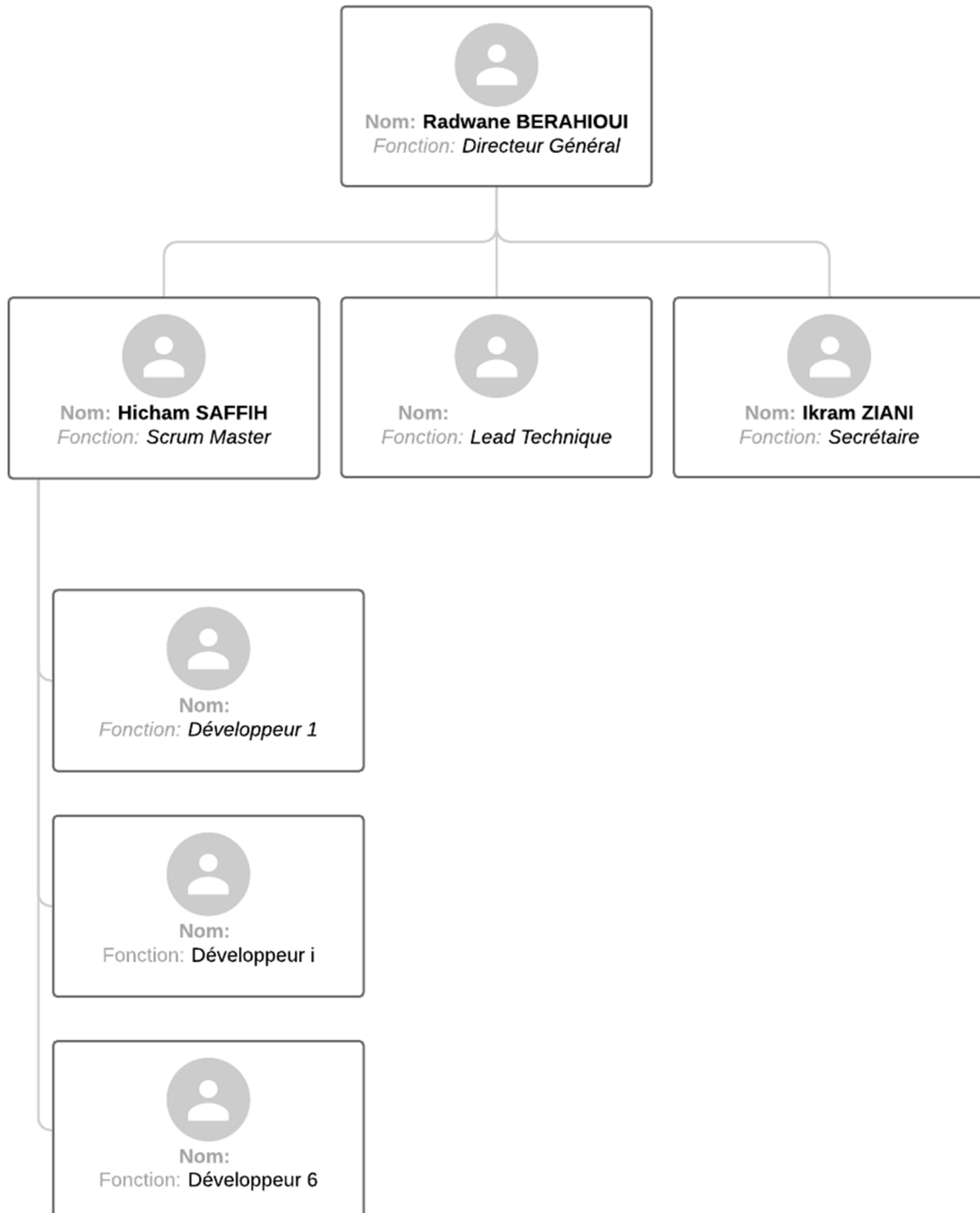


FIGURE 1.2: *Organigramme de l'entreprise Algolus*

## 1.3 Description des besoins

### 1.3.1 Problème

Dans le cadre du 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 générique et intelligente pour interpréter ces anomalies et proposer des correctifs pertinents.

Ce défi prend une dimension particulière dans le cadre des activités de Algolus, la complexité des systèmes gérés par l'entreprise amplifie les difficultés de diagnostic des anomalies. Les équipes techniques consacrent actuellement un volume considérable de leurs ressources temporelles à l'analyse manuelle des incidents, retardant d'autant les mises en production. Par ailleurs, la variété des clients et des cas d'usage entraîne une hétérogénéité des remontées d'erreurs (rapports techniques détaillés pour les clients corporate vs. simples captures d'écran pour les utilisateurs finaux), ce qui rend inefficaces les outils de monitoring conventionnels utilisés jusqu'à présent. Ce constat a motivé l'entreprise à explorer des solutions d'IA générative capables d'unifier l'interprétation des anomalies.

### 1.3.2 Les besoins fonctionnels

Les besoins fonctionnels définissent les actions spécifiques que le système doit accomplir pour répondre aux exigences métier. Ils décrivent le "quoi", qu'est ce que le système doit faire, sous la forme de fonctionnalités concrètes de processus et d'interactions avec l'utilisateur. Ces exigences formulées par les parties prenantes (les clients, les utilisateurs, et l'équipe produit) servent de base à la conception des cas d'usage et des scénarios de test.



Pour garantir que le système de diagnostic d'erreurs réponde efficacement aux attentes des utilisateurs et des équipes techniques, nous avons identifié les besoins fonctionnels suivants :

- 0. **Collecte et Pré-traitement des Données** : Extraction automatique des erreurs et des anomalies des systèmes à partir de : stacktraces, parsing des logs, captures d'écran, retours utilisateurs.
- 0. **Analyse et Compréhension** : Analyse sémantique de retours d'erreurs, enrichissement contextuel : requêtage d'une base de connaissances (documentation technique, correctifs historiques).
- 0. **Génération de Solutions** : Explication en langage naturel des causes racines, génération de correctifs (ex : snippets de code, étapes de résolution).
- 0. **Interfaces utilisateurs** : Soumission des erreurs via des formulaires web pour uploader des stacktraces et des captures d'écran.

### 1.3.3 Les besoins non fonctionnels

Les besoins non fonctionnels caractérisent le "comment", c'est à dire comment le système doit fonctionner, en précisant ses contraintes de qualité, de performance et d'infrastructure. Contrairement aux besoins fonctionnels, ils ne décrivent pas des fonctionnalités mais des critères tels que la rapidité, la sécurité, la scalabilité ou la facilité de maintenance. Leur respect est essentiel pour assurer la robustesse et l'efficacité du système en conditions réelles.

Pour garantir une intégration harmonieuse dans l'écosystème existant et une expérience utilisateur optimale, les besoins non fonctionnels suivants ont été définies :

- 0. **Performances** : Temps de réponse optimisé, et scalabilité : Support de plusieurs requêtes simultanées.
- 0. **Intégration et Interopérabilité** : API REST, Endpoints standardisés et format de réponse avec schéma cohérent, support offline ; fonctionnement local avec Ollama.

- 0. **Sécurité et Confidentialité** : Protection des données par chiffrement des échanges et anonymisation des logs utilisateurs (RGPD), et authentification : JWT pour l'accès aux APIs sensibles.
- 0. **Expérience Utilisateur** : Ergonomie : interface intuitive, Dark/Light mode et thèmes accessibles.

### 1.3.4 Solutions envisagées

Ce projet vise à développer une application intelligente et modulaire permettant de détecter et de corriger automatiquement les anomalies logicielles, nous avons identifié les objectifs spécifiques suivants :

- Détecter avec un taux de réussite d'au moins 80% les anomalies logicielles sur des sources multimodales.
- Proposer automatiquement des correctifs pertinents dans plus de 70% des cas.
- Optimiser le temps moyen de résolution d'erreurs de 30% par rapport aux méthodes manuelles.

Pour atteindre ces objectifs, nous avons envisagé une solution basée sur un ensemble de méthodes et technologies innovantes incluant :

- 0. **Analyse Multimodale des Erreurs** : Implémenter un système capable d'interpréter des données hétérogènes (stack traces, logs texte, captures d'écran, etc.), et utiliser des techniques de RAG pour enrichir les requêtes avec une base de connaissances (code source, documentation technique, résolutions d'erreurs courantes).
- 0. **Génération Automatique de Correctifs** : Exploiter des LLMs (via Ollama) pour suggérer des corrections précises et contextualisées.
- 0. **Intégration et Scalabilité** : Développer un backend Spring Boot flexible, couplé à LangChain4j pour orchestrer les appels IA, et un système de gestion de base de données qui prend en charge les bases de données vectorielles, comme PostgreSQL, et permettre une extension future via des connecteurs pour différents outils de monitoring.

0. **Optimisation et Évaluation** : mesurer l'efficacité du système via des métriques de précision (taux de détection, pertinence des correctifs), et effectuer un Benchmark : comparaison sur des jeux de données communs.

## 1.4 Conclusion

En résumé, l'analyse initiale du problème nous a permis de cerner clairement les enjeux du projet et de définir des besoins fonctionnels et non fonctionnels cohérents avec les objectifs visés. L'étude des différentes solutions envisageables a conduit à des choix techniques adaptés, prenant en compte à la fois les contraintes de performance, d'ergonomie et de maintenabilité. Bien que certaines limitations persistent, notamment en lien avec l'évolutivité ou les dépendances externes, les bases posées offrent un cadre solide pour le développement et l'amélioration continue du système. Cette première partie dédiée à la définition du problème, des besoins, et des solutions envisagées, constitue une étape vers une solution complète, fiable et évolutive qui nous a permis de bien enchaîner l'étape de l'analyse fonctionnelle et la conception du projet.

# Chapitre 2

## Analyse fonctionnelle et modélisation

### 2.1 Introduction

Dans ce chapitre, nous mettons l'accent d'abord sur l'importance de l'analyse fonctionnelle en rappelons qu'est ce que c'est la modélisation UML, ses principes fondamentaux et son utilité dans le développement logiciel. Par la suite, nous exposerons les différents types de diagrammes utilisés dans notre étude, à savoir le diagramme de cas d'utilisation, et les diagrammes de séquences, après la définition de leurs composants.

### 2.2 Importance de l'analyse fonctionnelle

L'analyse fonctionnelle constitue une étape clé dans tous projets de développement informatique, car elle permet de bien comprendre les besoins du client et les contraintes du système à réaliser, identifier les fonctionnalités attendues, détecter les éventuelles incohérences et poser les bases d'une conception solide. De surcroit, une analyse fonctionnelle bien menée réduit considérablement les risques d'erreurs en phase de développement, facilite la planification du travail et améliore la qualité globale du produit final, ce qui lui rend essentielle pour assurer la réussite du projet.

## 2.3 Unified Modeling Language

Dans le cadre d'un projet de développement informatique la modélisation UML (Unified Modeling Language) joue un rôle essentiel en facilitant la compréhension, la conception et la communication autour du système à développer. UML propose un ensemble de diagrammes normalisés qui permettent de représenter visuellement les différentes dimensions d'un logiciel, telles que la structure, le comportement et les interactions entre les composants.

L'utilisation des diagrammes UML comme les diagrammes de cas d'utilisation, de classes et de séquences permet de clarifier les besoins fonctionnels et non fonctionnels dès les premières phases du projet, de favoriser une meilleure communication entre les développeurs, les analystes et les clients, de surcroit il permet de détecter de façon précoce les incohérences ou erreurs potentielles dans la conception, et aussi de servir de documentation technique structurée pour le développement.

Ainsi, UML constitue un outil précieux pour assurer la qualité, la cohérence et la pérennité d'un projet informatique, en apportant une vision globale et partagée du système.

## 2.4 Architecture hybride : LLM et RAG

Avant de présenter nos diagrammes de conception UML, il est nécessaire de définir leurs différents éléments composants. Voici une liste des éléments clés qui permettent au système de répondre aux besoins définies dans le chapitre précédent.

- **LLM (Large Language Model)** : Est un modèle d'intelligence artificielle pré-entraîné sur des volumes massifs de données textuelles, capable de comprendre, générer et manipuler le langage naturel de manière contextuelle. Il est basé sur des architectures de réseaux de neurones (comme les transformers), et excelle dans des tâches variées (réponse aux questions, traduction, synthèse de texte, etc.) en prédisant des séquences linguistiques d'une manière probabiliste. Contrairement aux systèmes traditionnels, un LLM ne suit pas de règles prédéfinies, mais apprend des

motifs linguistiques en se basant sur des modèles non-supervisés.

- **RAG (Retrieval-Augmented Generation)** : Est une technique post-entraînement qui nous permet d'adapter l'usage des LLMs à travers l'enrichissement des réponses en recherchant des documents pertinents dans une base de connaissances externe, et en générant des réponses contextualisées à partir de ces sources. [1] Le RAG est composé de plusieurs éléments et services - dont nous allons parler tout de suite - organisés pour réaliser son objectifs.
- **Agent AI** : Une entité qui peut planifier, exécuter des tâches d'une manière agentique ; Il reçoit les requêtes brutes de l'utilisateur, les enrichit en combinant plusieurs techniques avancées comme le RAG et la mémoire conversationnelle, puis les présente au modèle de langage sous une forme optimale.

Parmi les atouts majeurs d'un Agent AI sa capacité de configurer d'une façon dynamique les trois parties constituant le prompt envoyé au LLM :

- **System Message** : définit le comportement global attendu du modèle (rôle, ton, contraintes, format de la réponse), en fixant un cadre pour l'interprétation des requêtes.
  - **User Input** : correspond à la requête explicite formulée par l'utilisateur, exprimant son besoin ou sa question.
  - **Few Shot Examples** : Quelques exemples de paires question/réponse (ou tâche/résultat), avant la question réelle de l'utilisateur, servant à orienter le comportement du modèle sans avoir à l'entraîner à nouveau.
- **ChatModel** : Incarne le moteur de génération de langage naturel. Ce composant spécifique, configuré pour utiliser des modèles locaux ou externes, transforme les prompts structurés en analyses techniques détaillées.

Le ChatModel permet de configurer plusieurs propriétés du LLM, citons les plus importantes :

- L'URL de base étant le point d'accès à l'API du modèle
- Le nom du modèle de langage à utiliser

- La Température, une propriété qui détermine le degré de créativité du LLM à formuler ses réponses, 0 étant le plus précis, et 1 le plus créatif.
- Le Timeout, qui est le délai maximal de réponse avant échéance.
- **ChatMemory** : est un composant logiciel conçu pour conserver l'historique des échanges dans un système conversationnel (comme un chatbot), permettant ainsi de maintenir le contexte entre les messages et d'offrir des réponses plus cohérentes et personnalisées.

La liste ci-dessous met l'accent sur un élément particulier qui est le RAG, elle définit le rôle de chacun de ses composants :

- **Resource** : Constitue la matière première du système RAG. Il s'agit de la source originelle des données qui alimenteront la base de connaissances. Ces ressources peuvent prendre diverses formes : fichiers PDF contenant la documentation technique, pages web de référence, extraits de bases de données, ou tous autres supports contenant des informations pertinentes.
- **Tokenizer** : Joue un rôle fondamental dans le prétraitement du texte, il décompose le contenu textuel en unités significatives appelées tokens, "un token peut correspondre à un mot entier, un sous-mot ou même un caractère individuel selon la méthode employée". Afin de rester dans les limites de longueur imposées par les modèles (souvent exprimées en nombre de tokens), le texte est ensuite segmenté en blocs appelés *chunks*. Ces chunks sont de taille contrôlée (par exemple, 512 tokens), éventuellement avec un chevauchement partiel entre eux, pour ne pas perdre de contexte entre deux segments. Des algorithmes avancés comme ceux proposés par HuggingFace - une entreprise pionnière dans le domaine du traitement du langage naturel (NLP) et de l'intelligence artificielle - permettent une *tokenisation* optimale qui préserve le sens tout en gérant les particularités linguistiques. La tokenisation joue un rôle crucial car elle influe directement la qualité des embeddings générés ultérieurement.
- **DocumentParser** : Assure la transformation des ressources brutes en documents structurés, il comprend divers formats de fichiers (PDF, HTML, Markdown, etc.)

et en extrait le contenu textuel significatif tout en conservant les métadonnées importantes. Des bibliothèques spécialisées comme sont souvent employées pour cette tâche complexe. Le DocumentParser nettoie également le texte en supprimant les éléments non pertinents (en-têtes, pieds de page, balises HTML) pour ne conserver que l'information essentielle.

- **Document** : Représente la forme normalisée et standardisée des informations après traitement. Chaque document contient non seulement le texte brut nettoyé, mais aussi des métadonnées descriptives (titre, auteur, date de création, source) qui faciliteront son identification et son utilisation ultérieure. Un identifiant unique est attribué à chaque document pour permettre son suivi tout au long du pipeline. Cette structuration rigoureuse est essentielle pour maintenir la cohérence des données dans les étapes suivantes du processus mené par le RAG.
- **EmbeddingModel** : Est au cœur de la transformation sémantique du système. Ce modèle sophistiqué convertit le texte en représentations vectorielles denses (embeddings) qui capturent le sens profond des contenus. Des modèles sont spécialisés dans cette tâche produisent des vecteurs où la similarité spatiale correspond à la similarité sémantique. La qualité de l'EmbeddingModel détermine directement la capacité du système à retrouver des documents pertinents pour une requête donnée.
- **EmbeddingStoreIngestor** : Orchestre le processus complet d'indexation des documents, il supervise plusieurs opérations critiques : il applique la tokenisation et la segmentation des textes, déclenche la génération des embeddings via l'EmbeddingModel, et gère le stockage final dans l'EmbeddingStore.  
L'EmbeddingStoreIngestor implémente souvent des stratégies de traitement par lots pour optimiser les performances et peut gérer des pipelines complexes de prétraitement avant la vectorisation.
- **EmbeddingStore** : Sert de mémoire à long terme au système RAG, cette base de données vectorielle spécialisée stocke les embeddings générés et permet des recherches rapides de similarité. L'EmbeddingStore supporte des opérations massives d'insertion tout en maintenant des temps de réponse faibles pour les requêtes.



- **Retriever** : Est le composant qui établit le pont entre les questions des utilisateurs et la base de connaissances. Lors d'une requête, le Retriever sollicite de l'EmbeddingModel la transformation des questions des utilisateurs en embeddings, puis recherche dans l'EmbeddingStore les documents dont les vecteurs sont les plus proches. Ce composant implémente des algorithmes de similarité vectorielle (cosine similarity par exemple) et peut être finement paramétré (nombre de résultats retournés, seuil de similarité minimal). Le Retriever joue ainsi un rôle déterminant dans la pertinence des requêtes fournis au LLM.

Le diagramme montré dans la figure 2.1 synthétise d'une manière visuelle ces interactions.

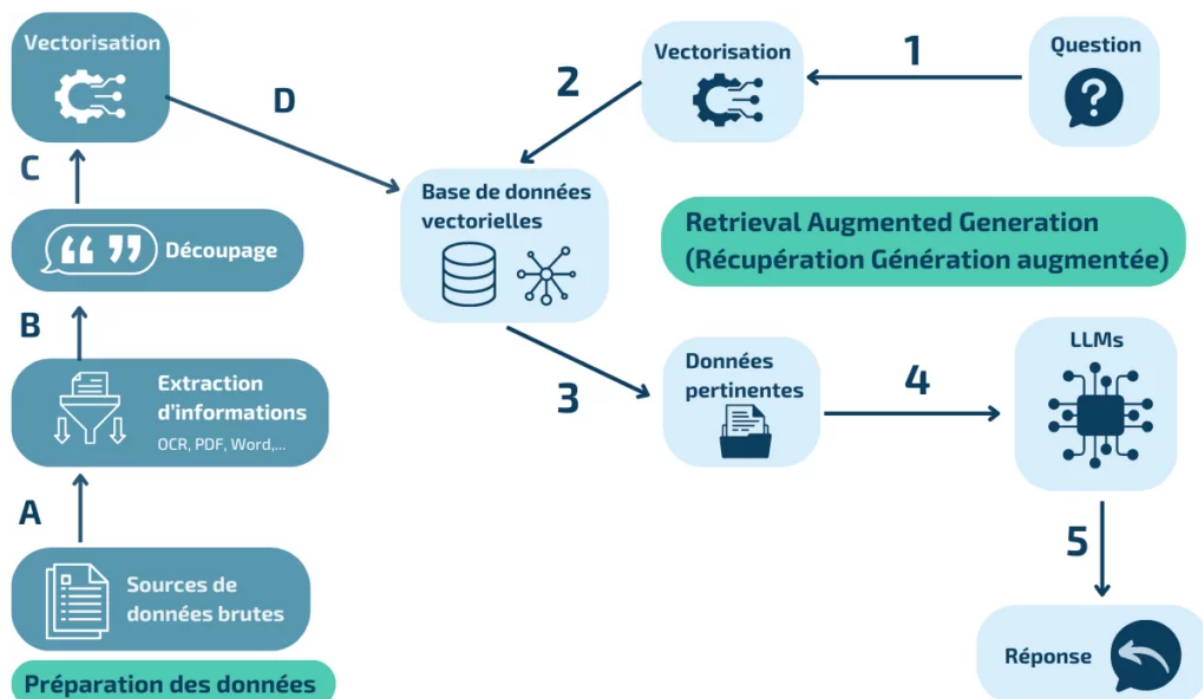


FIGURE 2.1: *Diagramme décrivant le fonctionnement d'un RAG*

## 2.5 Diagramme de cas d'utilisation

Un diagramme de cas d'utilisation est une représentation visuelle des interactions entre les acteurs (utilisateurs, systèmes) et les fonctionnalités d'une application. Il identifie les

besoins métiers sous forme d’actions (cas d’utilisation) et montre qui fait quoi, sans entrer dans les détails techniques.

### 2.5.1 Acteurs

Dans un diagramme de cas d’utilisation, les acteurs sont les entités qui interagissent avec le système pour accomplir un objectif précis. D’une part, un acteur peut être primaire (s’il est déclencheur d’un cas d’utilisation) ou secondaire (intervient dans un cas d’utilisation mais ne le déclenche pas), d’une autre part, un acteur peut être humain ou bien un acteur système.

Dans notre cas, trois types d’acteurs sont impliqués :

- **Utilisateur** : peut être un développeur ou un testeur qui rapporte une erreur, et peut interagir via une API REST ou bien une interface web.
- **Administrateur du système** : responsable de la mise à jour des connaissances du système et de la configuration des modèles.
- **Système** : le moteur de traitement intelligent, responsable d’analyser les anomalies, et de proposer des correctifs appropriés.

### 2.5.2 Notre diagramme de cas d’utilisations

La figure 2.2 présente le diagramme de cas d’utilisation de notre application (voir la page 16).

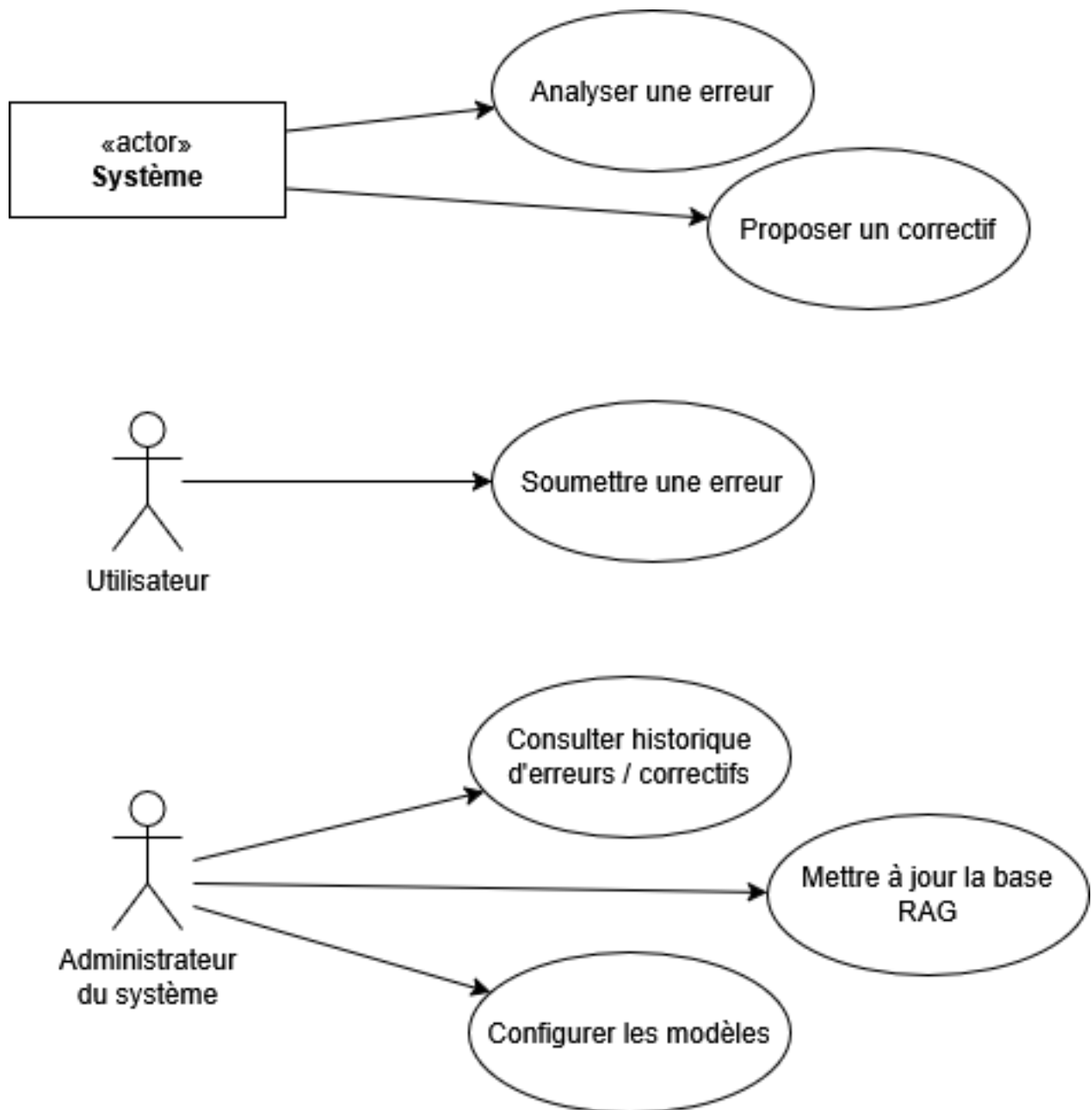


FIGURE 2.2: *Diagramme de cas d'utilisation*

## 2.6 Description textuelle

La description du déroulement et les préconditions des actions pour chaque cas d'utilisation est une étape méticuleuse qui nécessite une grande réflexion. Nous commençons souvent par une première description basée sur les informations que nous obtenons. Gé-

néralement, une description textuelle comporte l'acteur principal (entité initiatrice), l'objectif (finalité métier), et les préconditions (conditions préalables de déclenchement). Le déroulement est ensuite décrit à travers deux scénarios complémentaires : le scénario nominal, qui présente la séquence optimale d'actions aboutissant au résultat attendu, et les scénarios alternatifs, qui couvrent les exceptions et parcours d'erreur.

**Acteur Principal** : Utilisateur, développeur ou mainteneur logiciel confronté à une erreur technique ou fonctionnelle.

**Objectif** : Fournir un diagnostic précis et des correctifs pour des erreurs logicielles en combinant une description textuelle (stacktrace, logs), et une capture d'écran ou vidéo contextuelle.

### **Préconditions**

- 0. Accès à l'interface web de l'application
- 0. Services backend et modèle LLM (Ollama) opérationnels
- 0. Base de connaissances locale (RAG) préchargée avec le code source pertinent

### **Scénario Nominal**

#### **0. Soumission de la requête**

- Upload par l'utilisateur :
  - Description textuelle de l'erreur
  - Optionnellement une capture d'écran/vidéo
- Envoi au contrôleur

#### **0. Traitement par l'Agent AI**

- Récupération du contexte via `chatMemory`
- Enrichissement via requête RAG :
  - Extraction d'extraits de code pertinents
  - Documents techniques liés

#### **0. Appel au Modèle LLM**

- Formatage de la requête incluant :

- Description utilisateur
- Métadonnées multimédias
- Contexte historique + résultats RAG
- Réponse structurée en 3 parties :
  - 1 - Analyse de l'erreur : [diagnostic]
  - 2 - Source : [fichier:ligne]
  - 3 - Correctifs : [solutions]

## 0. Post-Traitement

- Sauvegarde dans `chatMemory`
- Journalisation de la transaction

## 0. Affichage du résultat

- Présentation claire avec mise en forme des chemins et snippets

## Scénarios Alternatifs

### A1. Erreur sans capture d'écran

- Ignore l'extraction visuelle
- Utilise uniquement texte et RAG

### A2. Échec du RAG

- Utilise le contexte historique uniquement
- Journalise un avertissement

### A3. Timeout du LLM

- Nouvelle tentative après un délai
- Retourne message d'erreur clair

## 2.7 Diagrammes de séquences

Un diagramme de séquences est un type de diagramme UML utilisé pour modéliser les interactions entre les différents objets ou composants d'un système dans un scénario précis, il met en évidence l'ordre chronologique des messages échangés entre les acteurs et les objets, les interactions dynamiques entre les éléments du système, et la durée de vie des objets participant au scénario.

Vu la complexité du système, et pour plus de visibilité, Nous avons élaboré deux diagrammes de séquences, un premier diagramme de séquences illustré dans la figure 2.3 décrivant les interactions entre les entités générales, et un deuxième diagramme de séquences montré dans la figure 2.4 concentré sur les interactions au sein du RAG.

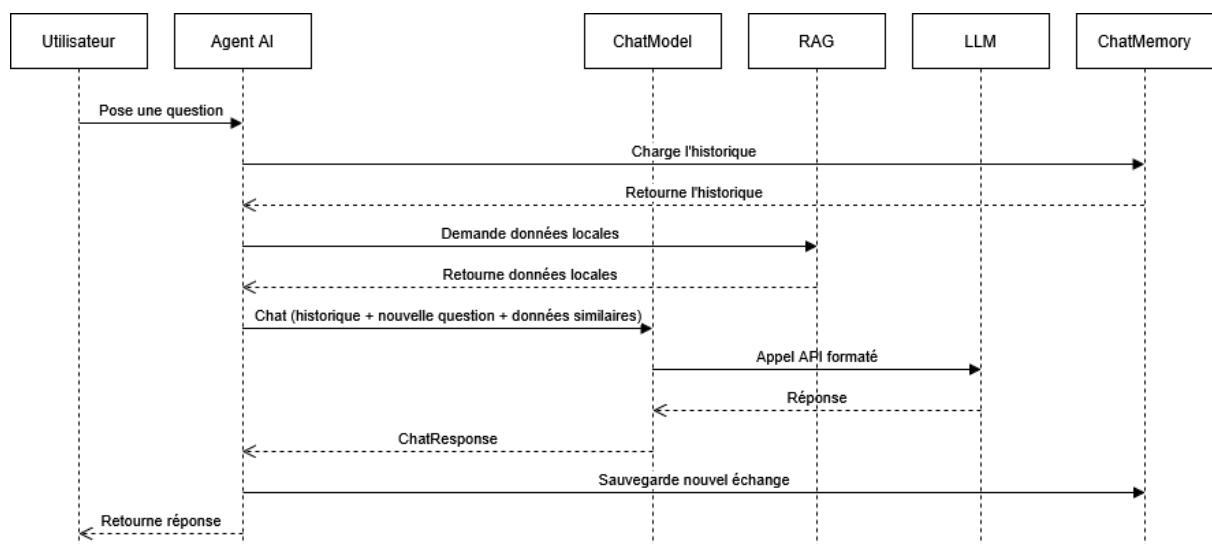


FIGURE 2.3: *Diagramme de séquences décrivant le fonctionnement d'un agent AI*

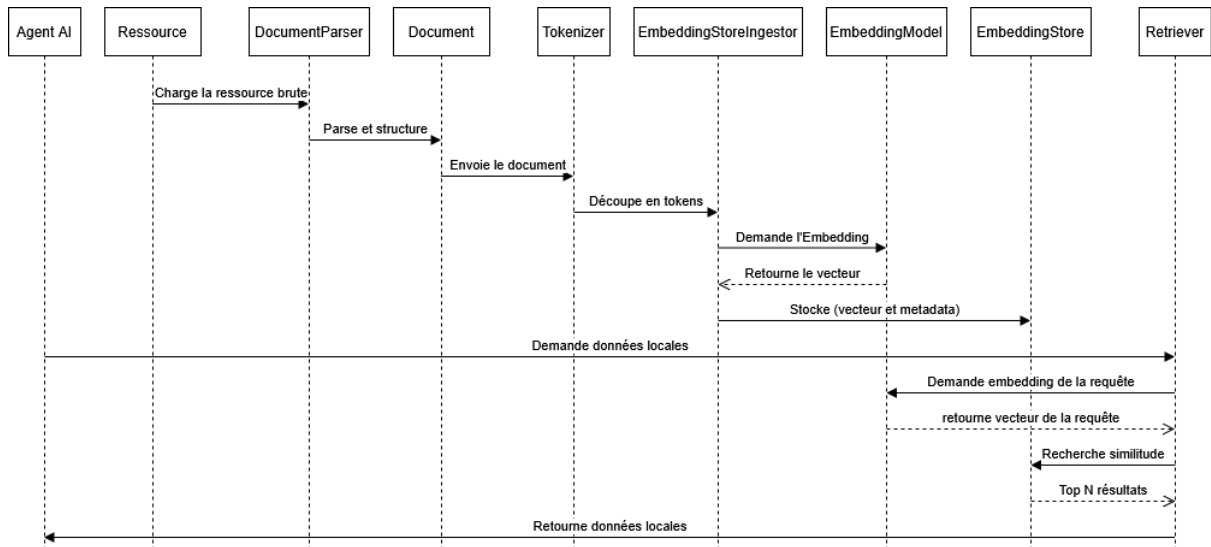


FIGURE 2.4: *Diagramme de séquences décrivant le fonctionnement d'un RAG*

## 2.8 Conclusion

L'analyse fonctionnelle et la conception ont constitué des étapes fondamentales dans la structuration de notre projet. Les modèles et diagrammes produits ont servi de base solide pour la phase de développement, ils ont permis de formaliser les interactions entre les différents composants tout en anticipant d'éventuelles contraintes techniques.

Enfin, cette phase préparatoire a mis en évidence l'importance d'une approche itérative, où l'analyse et la conception évoluent en parallèle des retours développeurs. Cette flexibilité méthodologique s'est avérée essentielle pour adapter le système aux besoins réels, tout en respectant les impératifs de qualité et de délais, ce qui sera constaté dans l'étape de réalisation.

# Chapitre 3

## Réalisation

### 3.1 Introduction

Ayant mené à l'étude des dimensions fonctionnelles et techniques, nous abordons désormais l'étape cruciale de réalisation concrète du projet. Cette phase opérationnelle sera consacrée à l'implémentation effective de la solution.

Ce chapitre présente la méthode de gestion adoptée, l'environnement de développement (outils, frameworks, bibliothèques, etc.), l'architecture logicielle retenue et son adéquation avec les besoins, les défis techniques rencontrés et les solutions apportées, et les composants clés implémentés avec des extraits de code significatifs, et des captures d'écran illustrant les résultats obtenus.

### 3.2 Méthode de gestion adoptée : SCRUM

#### 3.2.1 Scrum à la théorie

Scrum est un cadre méthodologique agile destiné à optimiser la gestion de projets complexes, en particulier dans le domaine du développement logiciel. Il repose sur des itérations courtes appelées *sprints*, au cours desquelles une équipe pluridisciplinaire s'engage à livrer un incrément fonctionnel du produit. Scrum définit des rôles précis (Scrum Master, Product Owner, équipe de développement), des artefacts (Product Backlog, Sprint



Backlog, Increment) et des événements clés (Daily Scrum, Sprint Planning, Sprint Review, Sprint Retrospective). Sa structure vise à favoriser la transparence, l'inspection régulière du travail accompli et l'adaptation rapide face aux changements. En théorie, Scrum encourage une collaboration étroite, une communication continue et une amélioration itérative du produit et des processus.

Les acteurs du framework SCRUM sont :

- **Product Owner** : Responsable de maximiser la valeur du produit en gérant le Product Backlog.
- **Scrum Master** : facilite le processus Scrum, veille à ce que l'équipe respecte le cadre et supprime les obstacles.
- **Équipe de développement** : équipe auto-organisée chargée de livrer un incrément du produit à la fin de chaque sprint.

La liste suivante définit brièvement les éléments clés de SCRUM :

- **Sprint** : itération fixe (généralement de 1 à 4 semaines) durant laquelle un incrément du produit est développé.
- **Product Backlog** : liste priorisée des fonctionnalités, exigences et corrections à apporter au produit.
- **Sprint Backlog** : sous-ensemble du *Product Backlog* sélectionné pour être réalisé durant le sprint.
- **Increment** : résultat fonctionnel du sprint, potentiellement livrable et utilisable.
- **Sprint Planning** : réunion de planification en début de sprint pour définir les objectifs et les tâches à réaliser.
- **Daily Scrum** : réunion quotidienne courte (15 min) pour synchroniser l'équipe et adapter le plan de travail.
- **Sprint Review** : réunion de fin de sprint pour présenter l'incrément et recueillir des retours.
- **Sprint Retrospective** : réunion pour analyser le déroulement du sprint et identifier des pistes d'amélioration.

La figure 3.1 illustre comment un projet est géré avec la méthode SCRUM.

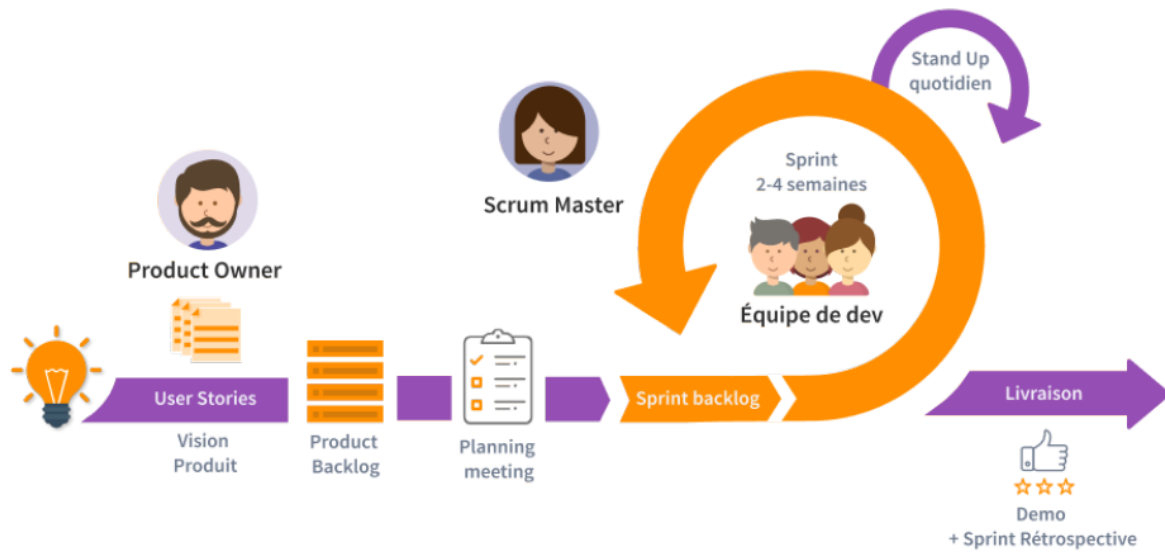


FIGURE 3.1: *Gestion de projet avec SCRUM*

### 3.2.2 Scrum à la pratique

#### Equipe du projet

TABLE 3.1: *Tableau des acteurs SCRUM*

Rôle	Acteur
Product Owner	BERAHIOUI Radwane
SCRUM Master	BERAHIOUI Radwane
Developeppers	LOUDIYI Hicham
Tech Lead	BERAHIOUI Radwane

#### Sprints du projet

Pour la planification du projet, nous utilisons l'outil de messagerie collaborative Slack. Il joue un rôle central dans la communication entre le Tech Lead et le développeur, tout en offrant une visibilité claire sur l'avancement global du projet. Slack facilite le suivi des sprints, la coordination des tâches et la gestion des différentes étapes du développement.

Grâce à ses fonctionnalités de planification et de suivi, il contribue à une organisation efficace et structurée du travail en équipe.

## 3.3 Stack technique

### 3.3.1 Langages

#### 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. la version utilisée est Java 17.



### 3.3.2 Frameworks

#### 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). La version utilisée est 6.1.8.



#### 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. La version utilisée est 3.4.5.



## 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. La version utilisée est 1.0.0-alpha1.



### 3.3.3 Bibliothèques

#### Apache Commons

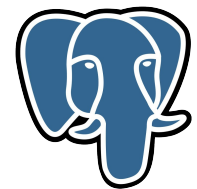
Apache Commons est une bibliothèque Java open-source fournissant des composants réutilisables pour simplifier le développement. Dans ce projet, elle sert à combler des besoins techniques récurrents avec des solutions optimisées et robustes. La version utilisée est 3.14.0.



### 3.3.4 Systèmes de gestion de bases de données

#### PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle (SGBDR) open-source, robuste et extensible. Dans le cadre de ce projet, il joue un rôle central pour stocker et gérer les données structurées nécessaires au bon fonctionnement de l'application, et fournit des plugins pour l'IA, notamment PgVector, qui gère les bases de données vectorielles. La version utilisée est 16.3

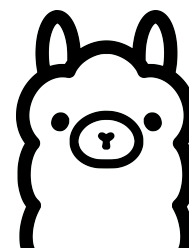


### 3.3.5 Outils et environnement

#### 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. [2]

La version utilisée est 0.9.2



#### Maven

Outil de build automatisé pour projets Java, qui gère Les dépendances (téléchargement auto), le packaging (JAR/WAR), et les cycles de compilation/test. La version utilisée est 3.9.10



#### 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). La version utilisée est 2025.1 (Ultimate Edition)



## Git

Git est un système de contrôle de version distribué, essentiel pour le développement collaboratif. Il permet de suivre les modifications du code source, de gérer les branches, et de fusionner les travaux de plusieurs contributeurs. Grâce à des plateformes comme GitHub, il facilite le partage et la revue de code. Son utilisation améliore la traçabilité, la qualité et la productivité dans les projets logiciels. La version utilisée est 2.47.1.windows.2.



### 3.4 Architecture technique du projet

Le projet repose sur une architecture modulaire et évolutive, construite autour des meilleures pratiques du développement Java moderne, de l'IA générative et de l'ingénierie logicielle. Il s'appuie sur les technologies suivantes :

- **Spring Boot** : Est le framework retenu pour le développement de la couche backend. Il s'agit d'un choix stratégique largement justifié par les besoins du projet en termes de performance, de maintenabilité et d'intégration avec des composants d'intelligence artificielle.

Spring Boot permet de structurer l'application de manière modulaire, en séparant clairement les responsabilités (contrôleurs, services, configuration, etc.). Cette organisation favorise une bonne lisibilité du code et facilite son évolution.

L'application est également portable, dans la mesure où elle peut être conditionnée sous forme de JAR exécutable et déployée facilement sur tout environnement compatible Java, sans dépendance à un serveur externe.

Un autre atout majeur est le caractère évolutif de Spring Boot : il s'intègre naturellement avec des bibliothèques telles que LangChain4j ou des bases de données comme PostgreSQL, ce qui permet d'ajouter de nouvelles fonctionnalités (IA, recherche vectorielle, mémoire contextuelle) sans remise en cause de l'existant.

Le framework est aussi testable : il propose des outils natifs pour la réalisation de tests unitaires et d'intégration, garantissant la qualité et la stabilité du code produit.

Enfin, Spring Boot est hautement extensible, alors si le projet venait à croître en complexité, il serait tout à fait envisageable de faire évoluer l'architecture vers un modèle microservices avec des outils comme Spring Cloud.

- **LangChain4j** : Pour permettre l'analyse intelligente des erreurs à l'aide d'un modèle de langage (LLM), le projet s'appuie sur la bibliothèque LangChain4j, une adaptation Java du framework LangChain initialement développé pour Python. Ce composant joue un rôle central dans l'intégration des fonctionnalités d'intelligence artificielle générative.

LangChain4j facilite la mise en œuvre d'un mécanisme de RAG (Retrieval-Augmented Generation), en combinant génération de texte via un LLM et récupération de documents pertinents à partir d'une base vectorielle. Il s'intègre naturellement avec un modèles de langage, et avec des composants tels que la mémoire de conversation, le modèle d'embedding et le store d'embeddings.

L'utilisation de LangChain4j rend l'application extensible et modulaire, car ses composants (LLM, mémoire, embeddings, etc.) sont interchangeables via des interfaces. Elle offre également un haut niveau de configurabilité, permettant d'adapter dynamiquement les modèles utilisés, la taille de la mémoire contextuelle ou encore les critères de pertinence documentaire.

Enfin, son intégration avec Spring Boot via des beans injectables simplifie grandement sa mise en œuvre dans l'architecture globale du projet. Cela permet d'enrichir les traitements métier avec une couche d'IA tout en conservant la lisibilité et la testabilité du code.

- **Ollama** : Pour exécuter les modèles de langage en local sans dépendre de services cloud externes, le projet intègre Ollama, une plateforme légère permettant de servir des LLM open-source tels que Mistral, Llama3 ou Qwen qui offre des versions multimodales. Ollama agit comme un point d'accès HTTP local à un modèle de

génération de texte, que LangChain4j peut interroger de manière transparente.

Ce choix présente plusieurs avantages : tout d'abord, il rend l'application autonome et portable, car aucun appel à une API cloud (comme OpenAI ou Hugging Face) n'est requis. Cela permet un déploiement sur des machines locales ou en environnement isolé (on-premise), tout en respectant les contraintes de confidentialité des données.

Grâce à une configuration centralisée (adresse du serveur, modèle utilisé, température, etc.), Ollama est également hautement configurable. Son intégration dans le projet se fait via des beans Spring instanciés dynamiquement dans la classe de configuration (AiConfig), ce qui permet d'adapter ou changer le modèle utilisé sans modifier la logique métier.

Enfin, en travaillant de concert avec LangChain4j, Ollama permet la génération de réponses contextualisées et pertinentes, en tenant compte des documents récupérés et des interactions passées. Cela renforce la capacité de l'application à fournir des analyses d'erreurs enrichies, précises et directement exploitables.

- **PostgreSQL** : Utilisé comme système de gestion de base de données relationnelle, avec une orientation spécifique vers le stockage vectoriel, dans le cadre de l'indexation et de la recherche de documents sémantiques. Grâce à l'extension pgvector, PostgreSQL devient capable de stocker des vecteurs d'embedding et d'effectuer des recherches de similarité, essentielles dans une approche RAG (Retrieval-Augmented Generation).

Le choix de PostgreSQL repose sur plusieurs critères clés : sa fiabilité, sa scalabilité et sa maturité en production. En plus de gérer des données relationnelles classiques (logs, utilisateurs, paramètres...), il peut aussi indexer efficacement des vecteurs issus des modèles d'embedding, et permettre des requêtes de type nearest neighbor search.

Son intégration avec Spring Boot est fluide grâce à JPA ou JDBC, et son usage dans ce projet est évolutif : dans un premier temps, les embeddings sont stockés en mémoire (InMemoryEmbeddingStore), mais la bascule vers PostgreSQL permettra



une persistance et une scalabilité bien supérieures, tout en conservant une interface compatible avec les composants LangChain4j.

Ainsi, PostgreSQL joue un rôle fondamental dans la stratégie d'enrichissement contextuel des requêtes utilisateur, en permettant à l'IA de s'appuyer sur des documents pertinents stockés localement de façon fiable et interrogeable.

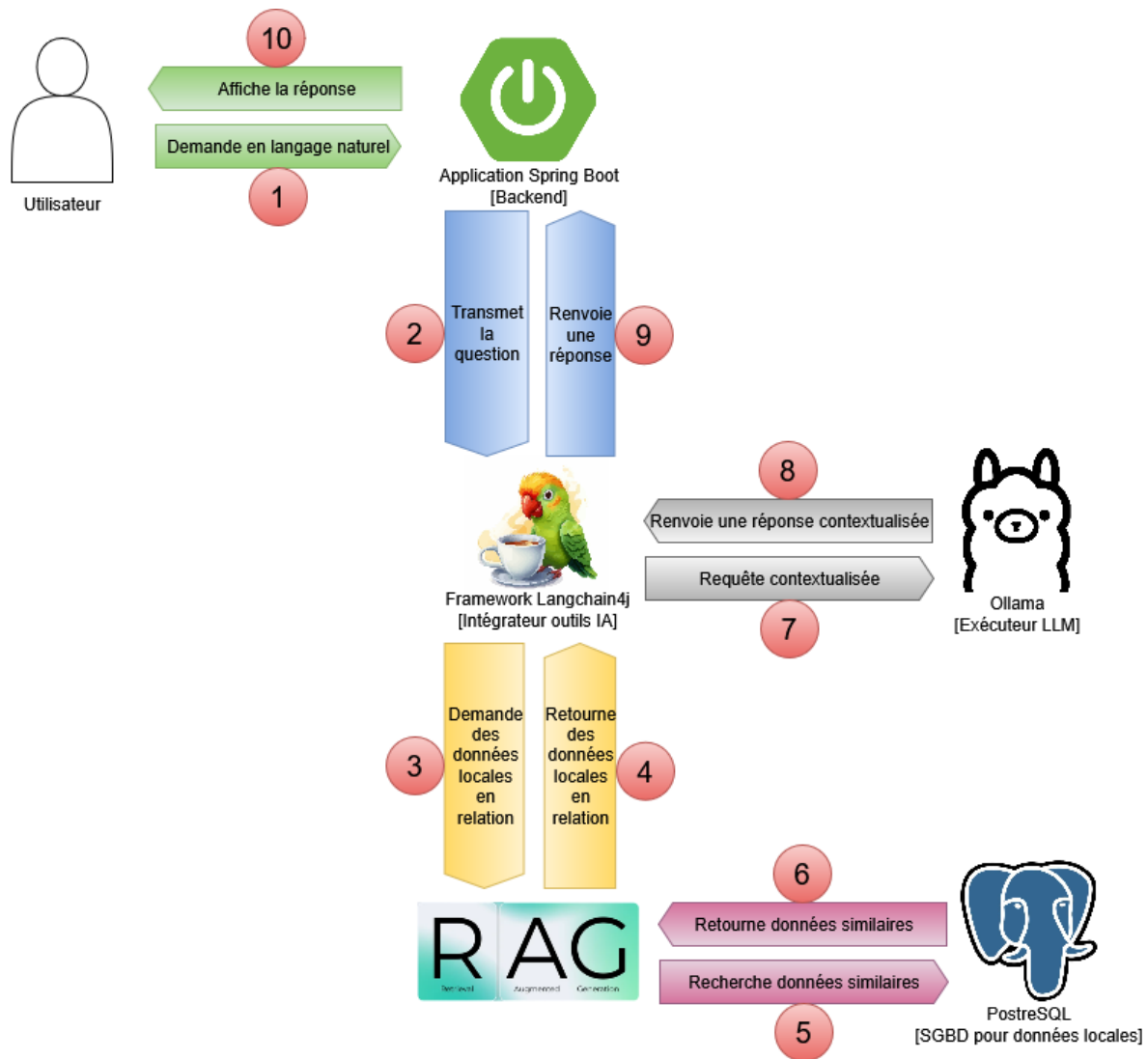


FIGURE 3.2: *Diagramme d'architecture technique*

La figure 3.2 illustre de manière synthétique les interactions entre les principaux composants techniques de l'architecture mise en place dans le cadre de ce projet. L'enchaînement commence par la soumission d'une requête par un utilisateur, le backend Spring

Boot reçoit cette requête et la délègue au framework LangChain4j, ce dernier sollicite du RAG - qui aura déjà chargé sa base de connaissances dans une base de données vectorielle - des données en relation avec la requête, puis encapsule ces données avec la requête et l'historique de conversation pour envoyer cet ensemble à Ollama, qui à son tour communique avec le LLM et reçoit une réponse, qui finit par arriver à l'utilisateur.

## 3.5 Bonnes pratiques appliquées du développement Java

Dans le développement logiciel, l'adoption de bonnes pratiques de codage est essentielle pour garantir la robustesse, la maintenabilité et l'évolutivité des applications. Un code bien structuré, avec une logique claire et une réduction des redondances, facilite non seulement les futures modifications, mais améliore aussi la collaboration entre développeurs. Cette section aborde des principes clés que nous avons suivis pour optimiser l'écriture du code, en mettant l'accent sur les méthodes que nous avons appliquées pour améliorer la qualité tout en réduisant les risques d'erreurs.

### 3.5.1 L'importance d'une nomenclature claire et cohérente

Une nomination précise des classes et méthodes est essentielle pour garantir la maintenabilité et la lisibilité du code. Dans notre projet, nous avons porté une attention particulière aux noms de classes, d'attributs et de méthodes, afin de permettre à un lecteur de code de comprendre immédiatement le rôle de chaque composant sans avoir à explorer son implémentation.

Citons quelques exemples concrets dans notre projet, la classe `ErrorAnalysisService` indique clairement qu'elle orchestre la logique métier de l'analyse d'erreurs, tandis que `FileValidator` évoque sans ambiguïté sa responsabilité : valider les fichiers entrants. De même, les méthodes comme `extractKeyFrames()` (dans `VideoProcessor`) ou `analyzeErrorWithRag()` (dans `ErrorAnalysisAgent`) décrivent explicitement leurs actions et leurs spécificités techniques. À l'inverse, des noms vagues comme `process()` ou `handle()` auraient rendu

le code obscur, obligeant les développeurs à parcourir le code source pour en comprendre la finalité. Par cette rigueur lexicale nous avons facilité aussi la collaboration entre équipes en réduisant les risques d'erreurs lors des évolutions futures. En résumé, un bon nommage agit comme une documentation intrinsèque, reflétant directement l'intention de conception et les principes métiers sous-jacents.

### 3.5.2 Utilisation de Bibliothèques Matures : Apache Commons

Plutôt que de réinventer la roue, nous avons utilisé des bibliothèques robustes comme Apache Commons, qui fournit des utilitaires optimisés pour plusieurs opérations telles que la manipulation de collections, les opérations sur les chaînes, et les validations.

Pour mettre en lumière les utilisations de ces utilitaires dans notre projet, prenons par exemple la méthode `estimateTokenCountInText(String text)` de la classe `TokenizerMyAppImpl`, dont la figure 3.3 montre une implémentation classique, utilisant l'instruction de branchement conditionnel `if` couplée aux opérateurs logiques.

```
@Override 12 usages
public int estimateTokenCountInText(String text) {
    if (text == null || text.isBlank())
        return 0;
    return tokenizer.encode(text).getTokens().length;
}
```

FIGURE 3.3: *Implémentation sans utiliser Apache Commons*

Cette approche, bien que fonctionnelle, présente plusieurs limitations : la combinaison de deux vérifications dans une même condition, la duplication de cette logique à de nombreux endroits du code, et le risque potentiel de `NullPointerException`.

L'adoption d'une nouvelle implémentation exploitant la bibliothèque Apache Commons est montrée dans la figure 3.4.

```

@Override 12 usages
public int estimateTokenCountInText(String text) {
    if (StringUtils.isBlank(text))
        return 0;
    return tokenizer.encode(text).getTokens().length;
}

```

FIGURE 3.4: *Implémentation en utilisant Apache Commons*

La méthode `isBlank()` de la classe `StringUtils` retourne `true` dans trois cas : si `text` est `null`, ou une chaîne de caractères vides, ou une chaîne de caractères ne contenant que des espaces blancs. Ce qui donne plus de robustesse au projet, en évitant les `NullPointerException` sans besoin de vérifier `null` explicitement, améliore la lisibilité en remplaçant une condition complexe par un seul appel clair, et garantit un comportement uniforme dans tout le projet.

### 3.5.3 Approches de parcours de collections en Java

Durant le développement, j'ai été amené à manipuler des collections de données. Plusieurs approches s'offraient alors à moi, chacune présentant des caractéristiques distinctes :

- **La boucle for traditionnelle** : Parcours indexé idéal pour les listes et tableaux, offrant un contrôle précis des positions mais inadapté aux collections non ordonnées comme les `Set`.
- **La boucle for-each** : Introduite dans Java 5, c'est une syntaxe simplifiée pour itérer sur tous les éléments d'une collection, évitant les erreurs d'index mais interdisant les modifications pendant le parcours.
- **L'Iterator** : Permet un parcours séquentiel avec suppression sécurisée via `remove()`, compatible avec toutes les collections mais plus verbeux à écrire.
- **Les Streams** : Approche déclarative exploitant des opérations fonctionnelles (filtrage, mapping) pour un code lisible et chainable, optimisé pour les traitements de données. [3]

- **ParallelStream** : Version parallélisée des Stream utilisant plusieurs cœurs CPU pour accélérer le traitement des gros volumes de données, au prix d'une complexité accrue en synchronisation.

Pour faire le bon choix, le tableau 3.2 donne une comparaison entre ces approches selon différents critères.

TABLE 3.2: *Comparaison des méthodes de parcours de collections en Java*

Critère	for	for-each	Iterator	Stream	ParallelStream
Modification possible	Risqué	Interdit	<code>remove()</code>	Interdit	Interdit
Accès par index	Oui	Non	Non	Non	Non
Lisibilité	Moyenne	Bonne	Faible	Excellente (déclaratif)	Excellente (déclaratif)
Types de sources	List/ Array	<b>Iterable</b>	<b>Collection</b>	Collection, Array, flux, etc.	Collection, Array, flux, etc.
Performance	Optimale	Légère baisse	Légère baisse	Bonne (selon cas)	Variable (multi-thread)
Opérations intégrées	Non	Non	Non	Oui	Oui
Parallélisme	Difficile	Impossible	Impossible	Possible	Automatique (Fork/Join)
Gestion du <b>null</b>	Manuel	Manuel	Manuel	<b>Optional</b>	<b>Optional</b>
Cas d'usage typique	Parcours indexé	Parcours simple séquentiel	Suppression d'éléments	Traitement fonctionnel	Traitement parallèle intensif

Prenons un exemple de notre projet, dans une implémentation de l'interface **Tokenizer** fournie par `LangChain4j`, la méthode `estimateTokenCountInTools(Iterable<Object> objectsWithTools)` a été implémentée dans un premier temps en utilisant une boucle `foreach`, comme le montre la figure 3.5.

```

@Override no usages
public int estimateTokenCountInTools(Iterable<Object> objectsWithTools) {
    int total = 0;
    if (objectsWithTools != null) {
        for (Object obj : objectsWithTools) {
            total += estimateTokenCountInTools(obj);
        }
    }
    return total;
}

```

FIGURE 3.5: *Implémentation sans utiliser les Streams*

Une évolution de cette implémentation utilisant les Streams ainsi que la bibliothèque Apache Commons est illustrée dans la figure 3.6.

```

@Override no usages hicham-loudiyi
public int estimateTokenCountInTools(Iterable<Object> objectsWithTools) {
    return StreamSupport.stream(
        IterableUtils.emptyIfNull(objectsWithTools).spliterator(),
        parallel: true) Stream<Object>
        .mapToInt(this::estimateTokenCountInTools) IntStream
        .sum();
}

```

FIGURE 3.6: *Implémentation en utilisant les Streams*

Cette implémentation, offrant une gestion robuste des null avec `IterableUtils.emptyIfNull()`, convertit l'Iterable en Stream parallélisable avec `StreamSupport.stream(..., true)`, ce qui permet un traitement réparti sur plusieurs cœurs CPU si la collection est grande, et une optimisation automatique pour les grands volumes de données, et finalement effectue un chaînage clair des opérations en appliquant les méthodes `mapToInt(...)` et `sum()`.

Nous pouvons très bien basculer ultérieurement vers l'utilisation des `ParallelStream`, si l'application a besoin de gérer une quantité massive de données.

### 3.5.4 Importance de la factorisation du code

La factorisation du code consiste à regrouper dans des méthodes ou classes dédiées les portions de logique qui se répètent à plusieurs endroits du programme. Nous avons adopté cette pratique qui s'inscrit dans les bonnes pratiques du développement logiciel, notamment le principe *DRY* (*Don't Repeat Yourself*), qui préconise d'éviter les duplications de code.

En factorisant, nous avons amélioré la lisibilité, la maintenabilité, et nous avons réduit les risques d'erreurs en centralisant les modifications à un seul endroit. Cela nous a permis également de clarifier les responsabilités des différentes classes, en accord avec le principe de responsabilité unique *SRP* (*Single Responsibility Principle*) du modèle SOLID.

La figure 3.7 montre une portion de code répétée avant factorisation.

```
@Override no usages 2 hicham-loudiyi
public int estimateTokenCountInToolSpecifications(Iterable<ToolSpecification> toolSpecifications) {
    return StreamSupport.stream(
        IterableUtils.emptyIfNull(toolSpecifications).spliterator(),
        parallel: true) Stream<ToolSpecification>
        .mapToInt( ToolSpecification spec -> estimateTokenCountInText(spec.toString())) IntStream
        .sum();
}

@Override no usages 2 hicham-loudiyi
public int estimateTokenCountInToolExecutionRequests(Iterable<ToolExecutionRequest> toolExecutionRequests) {
    return StreamSupport.stream(
        IterableUtils.emptyIfNull(toolExecutionRequests).spliterator(),
        parallel: true) Stream<ToolExecutionRequest>
        .mapToInt( ToolExecutionRequest req -> estimateTokenCountInText(req.toString())) IntStream
        .sum();
}
```

FIGURE 3.7: Exemple de code répétitif avant factorisation

Dans la figure 3.7, chacune des méthodes `estimateTokenCountInToolSpecifications` et `estimateTokenCountInToolExecutionRequests` permet d'utiliser un `Stream` pour estimer un nombre de tokens sur un itérable, c'est donc une logique répétée, seuls le type d'objets de l'`Iterable` et la méthode de comptage appliquée à chaque objet sont différents. La figure 3.8 illustre la version améliorée après factorisation.

```

private <T> int countTokensInIterable(Iterable<T> iterable, ToIntFunction<T> tokenCounter) { 3 usages  hicham-loudiyi
    return StreamSupport.stream(
        IterableUtils.emptyIfNull(iterable).spliterator(),
        parallel: true) Stream<T>
        .mapToInt(tokenCounter) IntStream
        .sum();
}

@Override no usages new *
public int estimateTokenCountInToolSpecifications(Iterable<ToolSpecification> toolSpecifications) {
    return countTokensInIterable(toolSpecifications, ToolSpecification spec -> estimateTokenCountInText(spec.toString()));
}

@Override no usages new *
public int estimateTokenCountInToolExecutionRequests(Iterable<ToolExecutionRequest> toolExecutionRequests) {
    return countTokensInIterable(toolExecutionRequests, ToolExecutionRequest req -> estimateTokenCountInText(req.toString()));
}

```

FIGURE 3.8: *Code refactorisé avec méthode utilitaire*

La logique répétée est regroupée dans la méthode utilitaire `countTokensInIterable(Iterable<T> iterable, ToIntFunction<T> tokenCounter)`, puis réutilisée en appelant cette méthode au besoin.

On constate une nette amélioration de la clarté du code, les règles de validation sont centralisées dans une méthode dédiée, facilitant ainsi leur réutilisation et leur évolution future.

### 3.5.5 Utilisation d'une classe de configuration

Pour standardiser et optimiser la gestion des paramètres techniques et métiers, nous avons intégré une classe de configuration nommée `ConfigurationPropertyValue` dans notre projet, comme le montre la figure 3.9. Dans cette classe nous avons externalisé des valeurs de configuration comme le nom du modèle de langage, sa température, le maximum de messages du ChatMemory, des paramètres du Retriever, la taille maximale des images uploadées et bien d'autres. Nous avons injecté ces valeurs dans la classe - via l'annotation `@Value` - dans un premier temps depuis le fichier de configuration `application.properties`, et nous avons basculé ensuite vers une base de données. L'annotation `@Value` offre la possibilité de définir des valeurs par défaut en cas d'absence de valeurs à injecter.



```

@Component 7 usages  hicham-loudiyi *
@Getter
public class ConfigurationPropertyValue {

    @Value("${max.imagesize:5242880}")
    private long maxImageSize;

    @Value("${ollama.baseurl:http://localhost:11434}")
    private String ollamaBaseUrl;

    @Value("${ollama.modelname:qwen2.5vl:7b}")
    private String ollamaModelName;

    @Value("${ollama.temperature:0.1}")
    private double temperature;

    @Value("${ollama.timeout:5}")
    private int ollamaTimeout;

    @Value("${ollama.chatmemory.maxmessages:20}")
    private int chatMemoryMaxMessages;

    @Value("${ollama.embeddings.modelname:nomic-embed-text}")
    private String embeddingsModelName;
}

```

FIGURE 3.9: *Un morceau de la classe de configuration*

Pour récupérer ces valeurs dans une autre classe, il suffit d'y injecter un bean de la classe de configuration, comme le montre l'exemple dans la figure 3.10.

```

@Configuration  @ hicham-loudiyi
@Slf4j
@RequiredArgsConstructor
public class AiConfig {

    private final ConfigurationPropertyValue config;

    @Bean  @ hicham-loudiyi
    public ChatLanguageModel llm() {
        return OllamaChatModel.builder()
            .baseUrl(config.getOllamaBaseUrl())
            .modelName(config.getOllamaModelName())
            .temperature(config.getTemperature())
            .timeout(Duration.ofMinutes(config.getOllamaTimeout()))
            .build();
    }
}

```

FIGURE 3.10: *Utilisation de la classe de configuration*

Avec cette approche on peut ajuster les paramètres techniques sans toucher au code métier et sans recompilation, on gagne plus de flexibilité en adaptant les paramètres à l'environnement (dev, test, prod) via des profils Spring, et on respecte l'un des principes SOLID, qui est la séparation des responsabilités, puisque le service se concentre sur la logique métier, tandis que la configuration technique est externalisée.

## 3.6 Interfaces graphiques réalisées

Pour atteindre nos objectifs ambitieux, nous avons suivi une approche incrémentielle. Dans un premier temps, nous avons réalisé un prototype fonctionnel afin de valider les fondements techniques du projet, de comprendre le fonctionnement du framework LangChain4j, de tester l'intégration avec Ollama, et de valider le concept de récupération de contexte à partir de documents externes.

Ensuite, Nous avons procédé au développement du projet final, en orientant le premier prototype vers l'analyse et la correction des erreurs et anomalies logicielles, et en le dotant d'une dimension multimodale.

### 3.6.1 Prototype exploratoire

Le premier prototype est basé sur un agent intelligent exploitant une architecture RAG unimodale, capable de répondre aux questions de l'utilisateur à partir d'un document texte ou PDF fourni.

Pour tester ce prototype, nous avons fourni à l'application le chemin d'un fichier PDF, dont le contenu est une lettre de recommandation pour une étudiante appelée Nour, nous avons ensuite posé une question à propos de cette étudiante à l'agent AI, en utilisant un contrôleur web. La figure 3.11 montre le résultat obtenu.

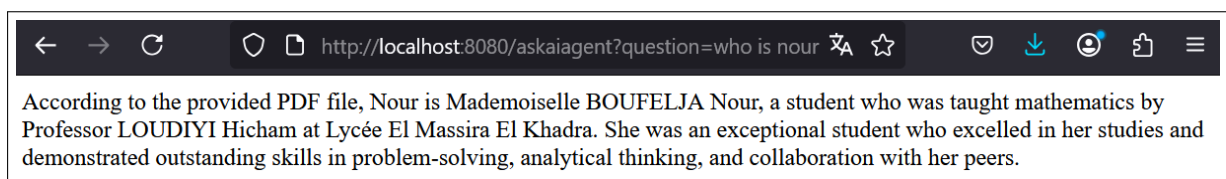


FIGURE 3.11: *Test du premier prototype*

Le modèle de langage que nous avons choisi pour tester ce prototype est Llama3, le prototype a réussi à répondre correctement à la question posée.

### 3.6.2 Version livrable actuelle

#### Structure du projet Java

Une bonne structuration d'un projet logiciel constitue un fondement essentiel pour assurer sa lisibilité, sa maintenabilité et son évolutivité. Dans notre cas, le projet respecte les conventions standard de l'écosystème Java et de l'architecture Spring Boot. L'arborescence capturée dans la figure 3.12 suit une séparation claire entre les différentes couches de l'application, notamment les agents (**agents**), la configuration (**config**), les objets de transfert de données (**dto**), les exceptions personnalisées (**exceptions**), les services métier (**service**) et la couche de présentation via les contrôleurs web (**web**). Par cette organisation modulaire, nous avons favorisé l'encapsulation des responsabilités, en conformité avec les principes SOLID, et facilité les tests unitaires ainsi que l'intégration

continue. Par ailleurs, l'intégration des ressources statiques et de configuration dans les répertoires `resources/static` et `application.properties` respecte les conventions de Spring Boot, permettant un déploiement harmonisé (voir la page 42).

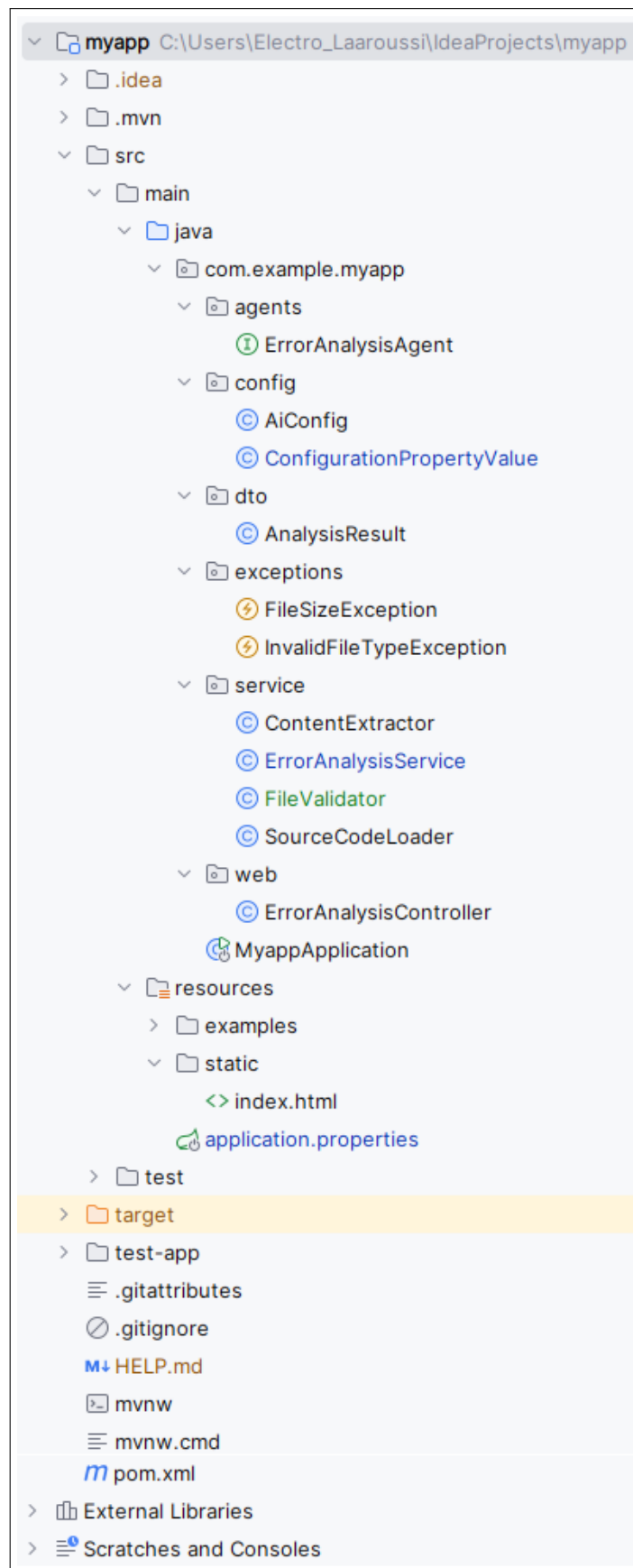


FIGURE 3.12: *Structure du projet Java*

## Configuration des modèles et chargement des ressources pour le RAG

Afin de permettre à notre application de traiter des entrées de nature hétérogène (texte, image, vidéo), le recours à un modèle de langage multimodal s'est révélé indispensable. À cet effet, nous avons choisi le modèle **Qwen2.5v1:7b** pour ses capacités avancées d'analyse conjointe du contenu textuel et visuel.

Nous avons aussi configuré plusieurs autres paramètres, telles que les paramètres relatifs au LLM, au modèle d'Embeddings, et au Retriever.

Les images de grande taille entraînent une diminution significative des performances en raison de leur impact sur la vitesse de transfert réseau, le temps de décodage en base64, ainsi que sur l'efficacité de l'analyse par le modèle de langage. Cette contrainte nous a conduit à imposer une limite stricte à leur taille maximale.

D'autre part, nous avons configuré le service **sourceCodeLoader** à charger les fichiers du code source de l'application présentant des erreurs, en lui disposant de son chemin. Ces fichiers constituent la matière première pour le RAG, qui les utilisera pour enrichir sa base de connaissances, afin d'obtenir des réponses cohérentes avec le contexte de l'application à déboguer.

Nous avons défini toutes les valeurs de ces configuration - en premier lieu - dans le fichier **application.properties**, comme l'illustre la figure 3.13 (voir la page 44).

```
max.imagesize=5242880

ollama.baseurl=http://localhost:11434
ollama.modelname=qwen2.5vl:7b
ollama.temperature=0.1
ollama.timeout=120

ollama.chatmemory.maxmessages=20

ollama.embeddings.modelname=nomic-embed-text
ollama.embeddings.timeout=5

contentretriever.maxresults=2
contentretriever.minscore=0.6

documentsplitter.maxsegmentsizeintokens=1000
documentsplitter.maxoverlapsizeintokens=100

sourcecode.path=C:\\Users\\Electro_Laaroussi\\IdeaProjects\\TestAppWeb\\src

api.paths.errors=/api/errors
```

FIGURE 3.13: *Morceau du contenu du fichier application.properties*

Par ailleurs, comme montré dans la figure 3.14 nous avons configuré l’agent AI pour définir son rôle, et pour lui décrire précisément l’input de l’utilisateur et qu’on attend de l’agent (voir la page 45).

```

@AiService( 2 usages  🧑 hicham-loudiyi *
    chatMemoryProvider = "chatMemoryProvider"
)
public interface ErrorAnalysisAgent {

    @SystemMessage(""" 1 usage  🧑 hicham-loudiyi
    Tu es un expert en analyse d'erreurs techniques.
    Utilise la base documentaire (qui contient le code source de l'application)
    pour contextualiser et diagnostiquer les erreurs.
    Lorsque tu identifies un fichier pertinent, mentionne son chemin (file_path).
    Ta réponse doit impérativement être structurée de la manière suivante : 1 - Analyse de l'erreur.
    2 - Source de l'erreur. 3 - Corrections proposées.
    """)
    String analyzeErrorWithRag(
        @UserMessage("""
        Analyse cette erreur:
        Stacktrace: {{stacktrace}}
        {% if screenshotBase64 %}Capture d'écran disponible{% endif %}

        Référence les fichiers sources pertinents en utilisant leurs métadonnées.
        """)
        @V("stacktrace") String stacktrace,
        @V("screenshotBase64") @Nullable String screenshotBase64
    );
}

```

FIGURE 3.14: *Interface Java de l'agent AI*

## Démarrage de l'Application Spring Boot

Au démarrage de l'application, Spring Boot initialise le contexte d'exécution en suivant une séquence bien définie. Tout commence par le chargement de la classe principale `MyappApplication`, annotée avec `@SpringBootApplication`, qui active la configuration automatique, la scan des composants et les propriétés définies dans `application.properties`. Les beans déclarés dans `AiConfig` sont instanciés, configurant notamment le modèle RAG et les services dépendants comme `ErrorAnalysisAgent` ou `SourceCodeLoader`. En parallèle, Spring MVC s'initialise pour préparer les endpoints du contrôleur `ErrorAnalysisController`. Enfin, le serveur embarqué Tomcat démarre et expose l'API sur le port configuré, rendant accessible l'interface Swagger UI pour tester les endpoints. Cette orchestration garantit que tous les services et composants sont prêts à traiter les requêtes dès le démarrage complet.



## Conformité aux Normes RESTful et Bonnes Pratiques d'API

Notre projet respecte les principes fondamentaux d'une API RESTful. En effet L'endpoint `/api/errors` suit une sémantique HTTP claire, la méthode POST est utilisée pour la création d'une ressource (analyse d'erreur), conformément aux verbes REST, et les réponses HTTP sont standardisées (200 pour le succès, 400/413/415/500 pour les erreurs métier), avec des codes de statut pertinents et des messages structurés dans le DTO `AnalysisResult`. L'utilisation de `@RestController` et de `MediaType.MULTIPART_FORM_DATA_VALUE` garantit une sérialisation/désérialisation transparente des données. [4]

De plus, l'API développée est documentée via Swagger/OpenAPI (à l'aide des annotations `@Tag`, `@Operation`),offrant une spécification interactive des endpoints, des schémas de requête/réponse, et des codes d'erreur. La gestion des données binaires (comme les captures d'écran via `MultipartFile`) et leur validation rigoureuse (contrôle de la taille maximale et des types MIME autorisés) respectent les bonnes pratiques RESTful pour les formats de données complexes. [5]

### Création d'une page web pour effectuer des tests

Nous avons conçu une interface web minimaliste (figure 3.15) permettant de soumettre les erreurs rencontrées dans les applications. Cette page offre un formulaire où l'utilisateur peut soumettre une description détaillée - et/ou une stacktrace - du problème ainsi qu'une capture d'écran optionnelle illustrant l'anomalie, un bouton "Analyser l'erreur", et un champs de texte où la réponse générée va être affichée.

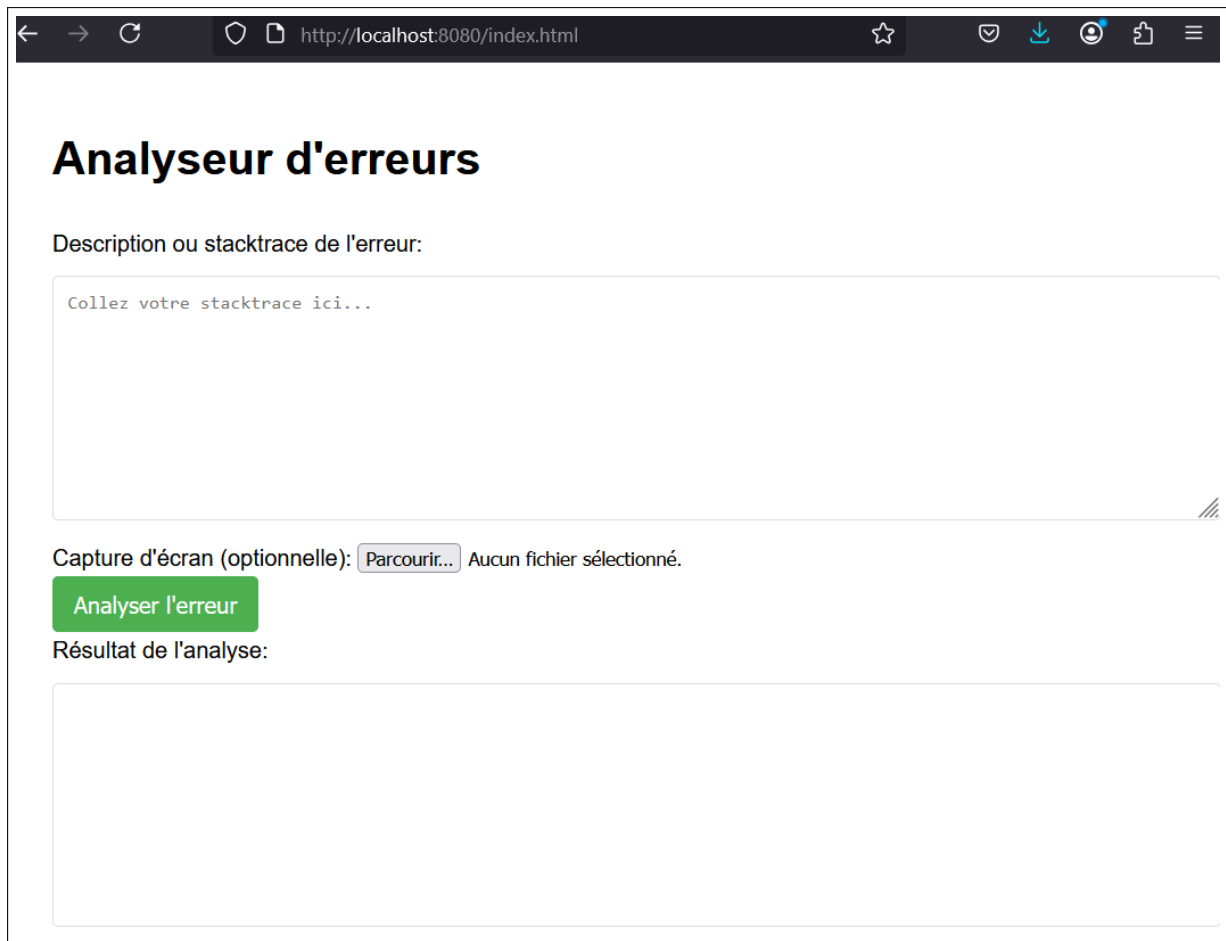


FIGURE 3.15: *Interface de l'application de test*

### Simulation d'une application qui génère une erreur

Afin de simuler un cas réel d'erreur dans une application, nous avons développé une application web de test générant intentionnellement une exception de type `NullPointerException`. Ce type d'exceptions - fréquent dans le développement logiciel - se produit lorsqu'une méthode tente d'accéder à un objet non initialisé (`null`). La figure 3.16 montre le résultat de l'exécution sur un navigateur.

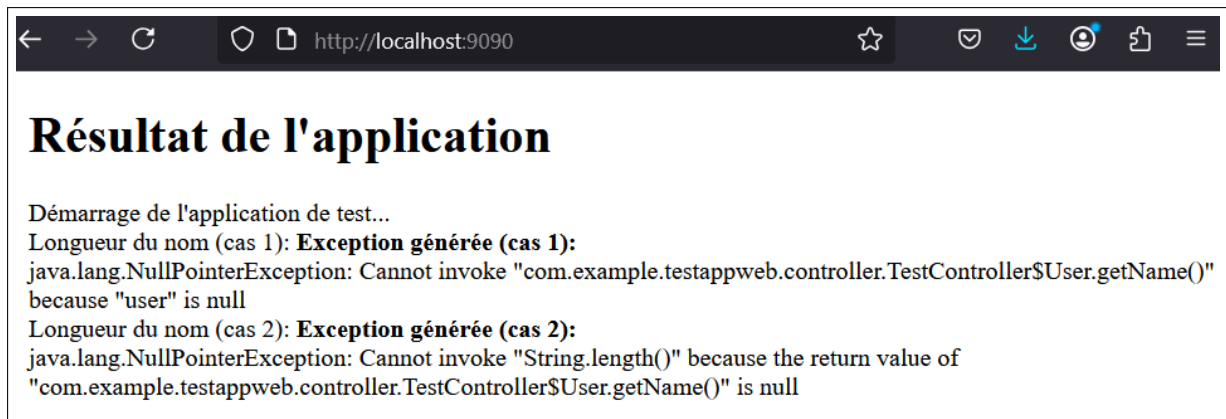


FIGURE 3.16: *Résultat d'exécution de l'application de test*

## Une analyse réussie

Nous avons soumis la capture d'écran dans la figure 3.16 à notre analyseur d'erreurs avec une petite description, ainsi que le chemin vers son code source (récupéré par RAG). Après une vingtaine de secondes, la réponse est affichée, comme illustré dans la figure 3.17 (voir la page 49).

## Analyseur d'erreurs

Description ou stacktrace de l'erreur:

Cette page web ne fonctionne pas, elle affiche des erreurs.

Capture d'écran (optionnelle):  simulation-erreur.png



### Résultat de l'application

Déroulement de l'application de test:  
Longueur du nom (cas 1): Exception générée (cas 1):  
java.lang.NullPointerException: Cannot invoke "com.example.testappweb.controller.TestController.getUser()" because "user" is null  
Longueur du nom (cas 2): Exception générée (cas 2):  
java.lang.NullPointerException: Cannot invoke "com.example.testappweb.controller.TestController.getUser()" because the return value of "com.example.testappweb.controller.TestController.getUser()" is null

Analyser l'erreur

Résultat de l'analyse:

1 - Analyse de l'erreur : La page web affiche des erreurs car elle tente d'accéder à la méthode `getName()` d'un objet `User` qui n'a pas été initialisé correctement. Dans le cas 1, `user` est `null`, ce qui entraîne une `NullPointerException` lors de l'appel à `user.getName().length()`. Dans le cas 2, `user2` est un objet `User` non initialisé, ce qui entraîne également une `NullPointerException` lors de l'appel à `user2.getName().length()`.

2 - Source de l'erreur : La source de l'erreur est dans la méthode `showExceptions` de la classe `TestController`. L'erreur se produit lorsqu'on tente d'appeler la méthode `getName()` sur un objet `User` qui n'a pas été initialisé correctement.

3 - Corrections proposées : Pour corriger cette erreur, il faut s'assurer que l'objet `User` est initialisé avant d'appeler la méthode `getName()`. Voici comment cela pourrait être fait :

```
```java
// Cas 1
User user = new User(); // Initialiser l'objet User avant d'appeler getName()
try {
    log.append("Longueur du nom (cas 1): ").append(user.getName().length()).append("
");
} catch (NullPointerException e) {
    log.append("Exception générée (cas 1):
")
    .append(e.toString()).append("
");
}

// Cas 2
User user2 = new User(); // Initialiser l'objet User avant d'appeler getName()
try {
    log.append("Longueur du nom (cas 2): ").append(user2.getName().length()).append("
");
} catch (NullPointerException e) {
    log.append("Exception générée (cas 2):
")
    .append(e.toString()).append("
");
}
```
```

En initialisant correctement l'objet `User` avant d'appeler la méthode `getName()`, on évite les `NullPointerException` et on peut ainsi afficher les résultats attendus.

FIGURE 3.17: *Résultat d'une analyse d'erreur réussie*

La réponse du système montre qu'il a correctement compris la requête, chargé la base de connaissances (code source), identifié l'erreur et sa cause, et proposé des correctifs.

Nous avons testé l'ensemble des endpoints de l'API avec succès via l'interface Swagger UI, qui a permis de simuler des requêtes HTTP dans différents scénarios d'utilisation. Les réponses retournées ont été systématiquement vérifiées en termes de structure, de code de statut, et de contenu. Par exemple, la figure 3.18 illustre le cas d'une analyse réussie, avec un code d'état 200 (voir la page 51).

POST

/api/errors

Analyser une erreur

Parameters

Cancel

Reset

No parameters

Request body required

multipart/form-data

stacktrace \* required

string

Stacktrace de l'erreur

Cette page web ne fonctionne pas, elle affiche des erreurs.

screenshot

string(\$binary)

Capture d'écran optionnelle

Parcourir...

simulation-erreur.png

☐

Send empty value

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
'http://localhost:8080/api/errors' \
-H 'accept: */*' \
-H 'Content-Type: multipart/form-data' \
-F 'stacktrace=Cette page web ne fonctionne pas, elle affiche des erreurs.' \
-F 'screenshot=@simulation-erreur.png;type=image/png'
```

Request URL

http://localhost:8080/api/errors

Server response

Code

Details

200

Response body

```
{
  "analysis": "1 - Analyse de l'erreur.\nLa page web affiche des erreurs car elle contient des exceptions qui ne sont pas gérées correctement. Dans le cas 1, l'erreur est générée car 'user' est 'null' et donc 'user.getName()' provoque une 'NullPointerException'. Dans le cas 2, l'erreur est également générée car 'user2' n'a pas de méthode 'getName()' définie.\n\n2 - Source de l'erreur.\nLes erreurs proviennent de la méthode 'showExceptions' dans le fichier 'TestController.java'. L'erreur est due à l'utilisation de 'user.getName().length()' sans s'assurer que 'user' est non null et à l'absence de méthode 'getName()' dans la classe 'User'.\n\n3 - Corrections proposées.\nPour corriger ces erreurs, il faut s'assurer que 'user' est non null avant d'appeler 'user.getName()'. Par exemple, on pourrait ajouter une condition pour vérifier que 'user' est non null avant d'appeler 'user.getName()'. Pour le cas 2, il faut s'assurer que la classe 'User' a une méthode 'getName()' définie. Voici un exemple de correction pour le cas 1 :\n\n// Cas 1\nUser user = new User(); // Initialiser l'objet User avant d'appeler getName()\ntry {\n  log.append(\"Longueur du nom (cas 1): \").append(user.getName().length()).append(\"<br/>\");\n} catch (NullPointerException e) {\n  log.append(\"<strong>Exception générée (cas 1):</strong><br/>\");\n}\nlog.append(e.toString()).append(\"<br/>\");\n}\n\nPour le cas 2, il faut s'assurer que la classe 'User' a une méthode 'getName()' définie. Si ce n'est pas le cas, il faut ajouter cette méthode à la classe 'User'.\n\n1 - Analyse de l'erreur.\nLa page web affiche des erreurs car elle contient des exceptions qui ne sont pas gérées correctement. Dans le cas 1, l'erreur est générée car 'user' est 'null' et donc 'user.getName()' provoque une 'NullPointerException'. Dans le cas 2, l'erreur est également générée car 'user2' n'a pas de méthode 'getName()' définie.\n\n2 - Source de l'erreur.\nLes erreurs proviennent de la méthode 'showExceptions' dans le fichier 'TestController.java'. L'erreur est due à l'utilisation de 'user.getName().length()' sans s'assurer que 'user' est non null et à l'absence de méthode 'getName()' dans la classe 'User'.\n\n3 - Corrections proposées.\nPour corriger ces erreurs, il faut s'assurer que 'user' est non null avant d'appeler 'user.getName()'. Par exemple, on pourrait ajouter une condition pour vérifier que 'user' est non null avant d'appeler 'user.getName()'. Pour le cas 2, il faut s'assurer que la classe 'User' a une méthode 'getName()' définie. Voici un exemple de correction pour le cas 1 :\n\n// Cas 1\nUser user = new User(); // Initialiser l'objet User avant d'appeler getName()\ntry {\n  log.append(\"Longueur du nom (cas 1): \").append(user.getName().length()).append(\"<br/>\");\n} catch (NullPointerException e) {\n  log.append(\"<strong>Exception générée (cas 1):</strong><br/>\");\n}\nlog.append(e.toString()).append(\"<br/>\");\n}\n\nPour le cas 2, il faut s'assurer que la classe 'User' a une méthode 'getName()' définie. Si ce n'est pas le cas, il faut ajouter cette méthode à la classe 'User'."

```

Download

Response headers

```
connection: keep-alive
content-type: application/json
date: Tue, 24 Jun 2025 15:07:55 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

FIGURE 3.18: Interface Swagger UI

## Test d'une image trop volumineuse

La figure 3.19 montre la réponse de l'API en cas de soumettre une image de taille supérieure à celle définie dans le paramètre `max.imagesize`.


### Analyseur d'erreurs

Description ou stacktrace de l'erreur:

Cette page web ne fonctionne pas, elle affiche des erreurs.

Capture d'écran (optionnelle):

vue-d-un-vieil-arbre-dans-un-lac-avec-les-montagnes-couvertes-de-neige-dans-le-un-jour-nuageux.jpg



Analyser l'erreur

Résultat de l'analyse:

Erreur: Erreur HTTP: 413

FIGURE 3.19: *Image trop volumineuse*

## Test d'une image de format non supporté

La figure 3.20 montre la réponse de l'API en cas de soumettre une image de format non supporté.

## Analyseur d'erreurs

Description ou stacktrace de l'erreur:

Cette page web ne fonctionne pas, elle affiche des erreurs.

Capture d'écran (optionnelle):  ds-ai-agent.drawio



Analyser l'erreur

Résultat de l'analyse:

Erreur: Erreur HTTP: 415

FIGURE 3.20: *Type d'images non supporté*

## Prise en charge des vidéos

Dans certains cas, une erreur ou une anomalie survenue au sein d'un système est plus efficacement illustrée par une capture vidéo. A cet effet nous avons enrichi notre système par l'ajout d'une fonctionnalité permettant le téléversement de vidéos, en intégrant un nouveau service nommé **VideoProcessor**. Nous avons conçu ce service de manière à qu'il soit capable d'extraire automatiquement des images clés à partir des vidéos fournies, via une bibliothèque appelée **FFmpeg**, facilitant ainsi l'analyse des dysfonctionnements observés.

La figure 3.21 illustre le résultat de l'upload d'une vidéo capturée au moment d'essayer d'atteindre une page web qui finit par afficher des stacktraces d'erreurs.



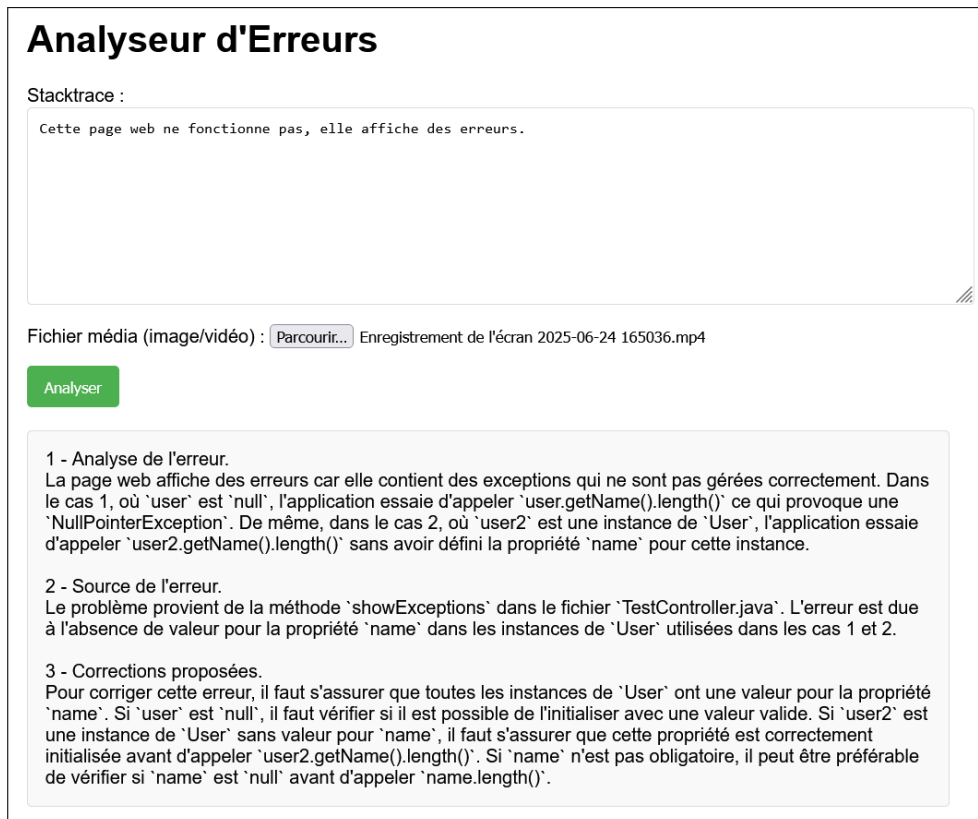


FIGURE 3.21: *Upload de vidéo*

La réponse obtenue lors de ce test, montre que notre système a bien identifié l'erreur et sa source, mais la correction qu'il a proposée est moins pertinente puisqu'il n'a pas suggéré des morceaux de code alternatifs.

## 3.7 Conclusion

La phase de réalisation nous a permis de concrétiser les spécifications techniques et fonctionnelles définies précédemment, en transformant les modèles conceptuels en un produit opérationnel. Grâce à une méthodologie de développement agile, chaque itération a contribué à l'enrichissement progressif du système. L'adoption de bonnes pratiques de codage, couplée à des outils de gestion de version et d'intégration continue, a assuré une base code robuste et maintenable. Les défis techniques rencontrés ont été résolus par des solutions optimisées, renforçant ainsi la fiabilité de l'application. Cette étape a non seulement validé les choix architecturaux initiaux, mais a également démontré la capacité

du système à répondre aux exigences métiers, ouvrant la voie aux phases de déploiement et d'exploitation.

# Chapitre 4

## Acquis du Projet de Fin d'Études

Ce stage de fin d'études m'a offert une opportunité enrichissante de développer un large éventail de compétences, aussi bien sur le plan technique que dans les domaines de la gestion de projet et des interactions professionnelles.

### 4.1 Volet technique

Sur le plan technique, ce stage m'a permis d'explorer des domaines et des outils que je n'avais jamais abordés auparavant.

Tout d'abord, la compétence principale acquise au cours de cette expérience est la maîtrise du RAG, une technique avancée d'IA générative. La mise en œuvre concrète de cette approche dans un projet réel m'a permis d'en comprendre pleinement les mécanismes, notamment grâce à l'utilisation de la bibliothèque `LangChain4j`.

Ensuite, j'ai approfondi mes connaissances dans le développement d'applications backend en Java, notamment à travers le framework `Spring Boot`, que j'ai utilisé pour concevoir des services robustes, configurables et modulaires. Cette expérience m'a ainsi permis de mieux appréhender les architectures modernes et l'intégration de composants d'intelligence artificielle dans un contexte professionnel.

## 4.2 Volet gestion de projet

Du point de vue de gestion de projet, ce stage m'a permis de travailler dans un cadre organisationnel structuré, fondé sur le framework **Scrum**, largement adopté par l'entreprise d'accueil, Algolus.

Au sein des différentes itérations (sprints), nous avons appliqué l'ensemble des pratiques agiles : réunions quotidiennes, envois de rapports journaliers, *backlog refinement*, *technical refinement*, *sprint review*, *rétrospective* et *planification de sprint*. Cette rigueur dans l'application de Scrum a contribué à améliorer significativement la productivité de l'équipe et la qualité des livrables.

Par ailleurs, Algolus utilise également le Lean Management, dont les méthodes comme la Résolution de Problème (RdP) ont contribué à accroître notre productivité.

# Conclusion générale et perspectives

## Conclusion générale

Ce projet a consisté en la conception et le développement d'une application dédiée à l'analyse automatique d'erreurs et anomalies, intégrant une approche innovante combinant traitement multimédia (images et vidéos) et intelligence artificielle à dimension multimodale. À travers une architecture modulaire basée sur Spring Boot et une interface web interactive, le système offre une solution robuste pour diagnostiquer des anomalies logicielles et proposer des corrections contextualisées.

Les principaux objectifs fixés ont été atteints :

- **Fonctionnalité de base** : L'application analyse efficacement les stacktraces, identifie les sources d'erreurs et suggère des correctifs grâce à une intégration réussie avec un modèle LLM (Ollama).
- **Support multimodal** : L'extension pour traiter des captures d'écran et des extraits vidéo ajoute une dimension visuelle à l'analyse, améliorant la précision des diagnostics.
- **Expérience utilisateur** : L'interface intuitive permet aux développeurs de soumettre facilement leurs erreurs et de recevoir des réponses structurées.

L'implémentation a mis en œuvre des bonnes pratiques logicielles :

- **Backend** : Architecture RESTful avec Spring Boot, validation des entrées, gestion des erreurs granulaires, et documentation OpenAPI.
- **Frontend** : Interface responsive avec gestion dynamique des médias et feedback visuel.

- **Intégration IA** : Utilisation de RAG pour contextualiser les analyses avec la base de code source.

## Perspectives

Comme tout projet informatique, notre solution présente à la fois certaines limitations inhérentes à sa conception actuelle et des perspectives d'amélioration qui ouvrent la voie à des évolutions futures.

Les pistes d'amélioration incluent :

- L'ajout d'un deuxième agent AI qui a pour rôle de juger la pertinence des réponses et/ou leur respect de certaines normes (comme RGPD).
- L'intégration d'un fine-tuning du modèle sur des bases de code spécifiques.
- L'ajout d'un système d'authentification pour un suivi personnalisé des analyses.
- L'intégration avec des plateformes comme GitHub pour une analyse directe des dépôts.
- L'amélioration des prompts IA pour couvrir un spectre plus large d'erreurs.
- L'optimisation des performances via le caching des analyses récurrentes.

# Bibliographie

- [1] Lewis, P. et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, arXiv, 2020.  
<https://arxiv.org/abs/2005.11401>

# Webographie

[2] Ollama, *Model Library*, 2023.

<https://ollama.ai/library>

[3] Oracle, *Package java.util.stream*, 2021.

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html>

[4] Spring Team, *Building REST Services with Spring*, 2023.

<https://spring.io/guides/gs/rest-service/>

[5] OpenAPI Initiative, *OpenAPI Specification*, 2023.

<https://swagger.io/specification/>



# Annexes

## Evaluation des objectifs

### Méthodologie d'évaluation

Pour mesurer l'efficacité du système, nous avons mené des tests sur un échantillon d'erreurs courantes (de types `NullPointerException`, `ArrayIndexOutOfBoundsException`, `Memory Leaks`, etc.), en comparant les résultats de l'outil avec ceux d'une analyse humaine. Les critères d'évaluation incluaient la précision de détection, la pertinence des correctifs, et le gain de temps.

### Résultats obtenus

Après effectué les tests d'évaluation des objectifs sur un échantillon de 30 erreurs, les métriques obtenues sont :

- **Taux de détection : 78%** (proche de l'objectif de 80%).

Cela s'explique par le fait que les erreurs complexes ou dépendantes du contexte métier ont réduit la précision. Notre système montre cependant des performances supérieures sur les erreurs syntaxiques et les crashes simples.

- **Correctifs pertinents : 65%** (objectif de 70% partiellement atteint).

Les corrections nécessitant une refactorisation majeure ou une compréhension approfondie de l'architecture ne sont pas toujours proposées. L'outil excelle sur les correctifs localisés.

- **Gain de temps : 35%** (objectif dépassé).

L'automatisation réduit significativement le temps de diagnostic, surtout pour les erreurs récurrentes. Cependant, les cas complexes nécessitent toujours une intervention humaine.

## Benchmark de LLMs Multimodaux

Nous avons mené une comparaison entre trois modèles de langages multimodaux disponibles via Ollama, qui sont `llava`, `qwen2.5vl:7b`, et `bakllava`, selon deux critères importants : les performances (Temps de réponse, utilisation CPU/RAM), et la pertinence des réponses (Exactitude des diagnostics d'erreurs).

La configuration matérielle utilisée est Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz 2.30 GHz avec une mémoire RAM de 32,0 Go, dont 16,0 Go est partagée avec le processeur graphique Intel(R) UHD Graphics.

Nous avons préparé un jeu de données de 20 erreurs (10 Python, 10 Java) avec stack-traces + 5 captures d'écran.

## Résultats synthétiques

Nous avons synthétisé les métriques de cette comparaison en calculant la valeur moyenne pour chaque paire LLM/critère sur le jeu de données. Le tableau 4.1 présente les résultats calculés.

TABLE 4.1: *Comparaison des LLM multimodaux*

| Modèle        | Latence (s) | RAM (GB) | Précision (%) | Pertinence |
|---------------|-------------|----------|---------------|------------|
| LLaVA 7B      | 5.4         | 6.8      | 62            | 3.2        |
| Qwen2.5VL :7B | 4.7         | 7.5      | 75            | 4.0        |
| BakLLaVA 7B   | 6.2         | 8.1      | 68            | 3.8        |

### Légende :

- Précision : % de correctifs validés par un développeur.
- Pertinence : Note moyenne (5 = parfait).

## Analyse des résultats

### Performances

- qwen2.5v1 est le plus rapide (4.7s) grâce à son optimisation pour les CPUs modestes.
- BakLLaVA consomme plus de RAM (8.1 Go), limitant son usage sur machines légères.

### Qualité des Réponses

- qwen2.5v1 excelle en précision (75%), notamment pour les erreurs Python.
- LLaVA fournit des réponses moins précises mais acceptables en temps réel.

### Visualisation des Performances

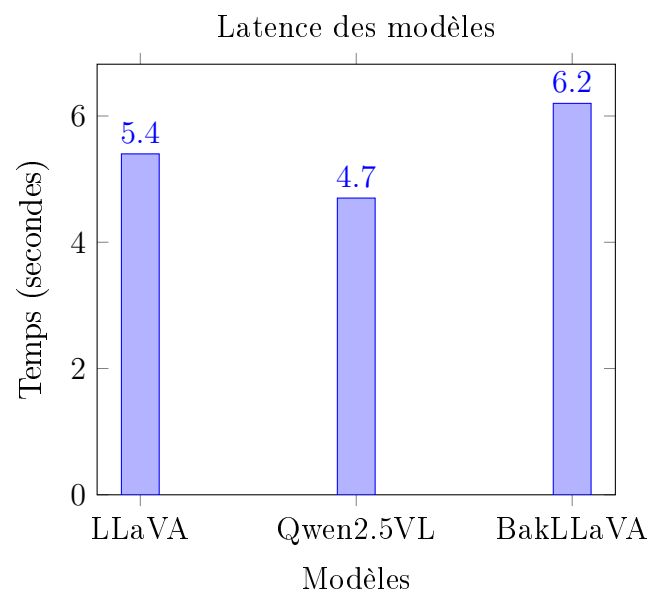


FIGURE 4.1: *Comparaison des temps de réponse moyens*

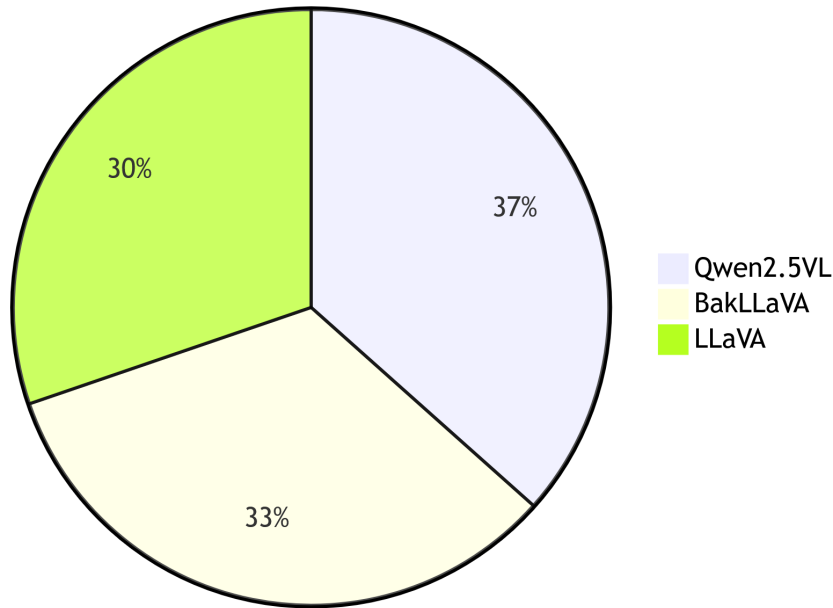


FIGURE 4.2: *Précision des modèles testés*

## Discussion des résultats

L'analyse comparative révèle que `qwen2.5v1` constitue le meilleur compromis pour les configurations matérielles limitées, combinant une latence réduite (4.7 s) et une précision satisfaisante (75 %), notamment pour les erreurs Python. À l'inverse, `BakLLaVA`, bien que proposant des résultats plus détaillés, s'avère trop gourmand en ressources (8.1 Go de RAM), le rendant peu adapté aux machines modestes. Ces conclusions doivent néanmoins être nuancées par la taille limitée du jeu de données (20 échantillons) – une validation sur un corpus plus large serait nécessaire pour consolider ces observations. En pratique, nous recommandons d'utiliser `qwen2.5v1` comme modèle par défaut pour les analyses courantes, tandis que `LLaVA`, moins précis mais plus rapide, pourrait être réservé aux tests ponctuels nécessitant un temps de réponse minimal.