RAPPORT DE STAGE

Fin d'Études - Ingénieur Informatique

Conception et développement d'un agent IA pour l'automatisation de l'audit de sécurité et de qualité des logiciels

---

master WeDSci

ULCO

Entreprise d'accueil

Diag n' Grow

Geoffrey Pruvost

Tuteur Académique

Mentor Académique

AVANT-PROPOS

Ce rapport marque l'aboutissement d'un stage de six mois réalisé dans le cadre du Master Informatique, parcours *Web et Science des Données (WeDSci)* de l'Université du Littoral Côte d'Opale (ULCO). Intégré au sein de l'entreprise *Diag n'Grow*, ce projet s'inscrit dans une dynamique de recherche et développement visant à explorer les applications de l'intelligence artificielle dans l'audit logiciel.

L'idée d'un agent IA dédié à l'analyse automatisée de code est née d'un constat : la complexité croissante des systèmes informatiques exige des outils capables de détecter, avec précision et rapidité, des vulnérabilités, des non-conformités ou des optimisations potentielles. Ce stage a ainsi offert l'opportunité de concilier des enjeux académiques – tels que l'approfondissement des méthodes d'apprentissage automatique et de traitement des données – avec des défis concrets, propres au monde professionnel.

Ce travail s'est articulé autour de trois axes principaux : la compréhension des besoins métiers en matière d'audit logiciel, la conception d'un prototype fonctionnel, et l'évaluation de ses performances. Il a également permis de mesurer l'importance d'une approche rigoureuse, alliant théorie et pratique, pour répondre à des problématiques techniques tout en respectant des contraintes opérationnelles.

Fait à Dunkerque, le 15 février 2026.

SOMMAIRE

SOMMAIRE

INTRODUCTION

1. Contexte et enjeux du stage

L’audit logiciel constitue une étape critique dans le cycle de développement des applications, garantissant leur conformité aux normes de qualité, de sécurité et de maintenabilité. Dans un contexte où les systèmes informatiques deviennent de plus en plus complexes et interconnectés, les méthodes traditionnelles d’audit, souvent manuelles et chronophages, montrent leurs limites face à l’évolution rapide des technologies et à la multiplication des dépendances logicielles. Cette problématique est particulièrement prégnante dans les environnements open-source, où la diversité des architectures, des frameworks et des pratiques de développement rend l’évaluation systématique des projets à la fois nécessaire et ardue.

C’est dans ce cadre que s’inscrit le stage réalisé au sein de l’entreprise **Diag n’Grow**, une structure spécialisée dans l’innovation en ingénierie logicielle et en science des données. L’objectif principal de ce stage était la **conception d’un agent d’intelligence artificielle (IA) dédié à l’audit automatisé de logiciels**, avec une focalisation sur les projets développés en **Java/Spring Boot** et gérés via l’outil de build **Maven**. Ce choix technologique n’est pas anodin : Spring Boot, en tant que framework dominant pour le développement d’applications d’entreprise, et Maven, en tant qu’outil de gestion de dépendances et de build, représentent un écosystème largement adopté dans l’industrie. Cependant, leur complexité intrinsèque – liée à la multiplicité des configurations possibles, des plugins et des dépendances – en fait un terrain propice aux erreurs, aux vulnérabilités et aux incohérences structurelles.

L’automatisation de l’audit logiciel via un agent IA répond à un triple enjeu : 1. **L’efficacité** : réduire le temps et les ressources nécessaires à l’évaluation d’un projet, en remplaçant des tâches répétitives et fastidieuses par des analyses algorithmiques. 2. **La précision** : minimiser les erreurs humaines en appliquant des règles d’audit standardisées et reproductibles, tout en identifiant des patterns complexes ou des anomalies difficilement détectables manuellement. 3. **L’évolutivité** : permettre l’analyse de grands volumes de projets, une nécessité croissante dans un contexte où les entreprises et les communautés open-source gèrent des centaines, voire des milliers de dépôts logiciels.

Pour illustrer ces enjeux, considérons l’exemple d’un projet Spring Boot tel que **BankingPortal-API**, mentionné dans le cadre expérimental. Ce type d’application, intégrant des couches de sécurité comme OAuth2 ou JWT, des bases de données relationnelles et des microservices, présente une architecture modulaire où chaque composant peut introduire des risques spécifiques : dépendances obsolètes, configurations de sécurité non conformes, ou encore violations de bonnes pratiques de codage. L’audit manuel d’un tel

projet nécessite une expertise pointue et un temps considérable, alors qu'un agent IA peut, en quelques minutes, analyser l'ensemble des fichiers de configuration (comme le `pom.xml` pour Maven), détecter les vulnérabilités connues (via des bases de données comme CVE), et générer un rapport structuré des risques identifiés.

2. Problématique et objectifs du stage

La problématique centrale de ce stage peut se formuler ainsi : **Comment concevoir un agent IA capable d'automatiser l'audit de logiciels open-source, en particulier ceux basés sur Java/Spring Boot et Maven, tout en garantissant la pertinence, la fiabilité et l'exhaustivité des analyses produites ?**

Cette problématique se décline en plusieurs sous-questions, qui ont guidé les travaux réalisés : - **Quels critères d'audit retenir** pour évaluer la qualité, la sécurité et la maintenabilité d'un projet logiciel ? Ces critères doivent couvrir des aspects techniques (dépendances, configurations, structure du code) mais aussi des dimensions plus larges, comme la conformité aux bonnes pratiques de développement ou la détection de vulnérabilités connues. - **Quelles méthodes d'analyse automatisée mettre en œuvre** pour extraire et interpréter les données pertinentes d'un projet ? Cela inclut la parsing des fichiers de configuration (comme le `pom.xml` pour Maven), l'analyse statique du code, ou encore l'interrogation de bases de données externes (comme les CVE pour les vulnérabilités). - **Comment valider la pertinence des résultats produits par l'agent IA** ? Cette question renvoie à la nécessité de comparer les analyses automatisées avec des audits manuels, ou de s'appuyer sur des benchmarks reconnus (comme les rapports de sécurité de projets open-source). - **Quels outils et frameworks utiliser** pour développer un tel agent ? Le choix des technologies (Python pour le traitement des données, des bibliothèques comme `lxml` pour le parsing XML, ou des APIs comme GitHub pour l'accès aux projets) a un impact direct sur la performance et la maintenabilité de la solution.

Pour répondre à ces questions, les objectifs du stage ont été structurés en trois phases principales : 1. **La conceptualisation de l'agent IA** : définition des critères d'audit, des sources de données à analyser, et des règles de détection des anomalies. Cette phase a impliqué une revue de la littérature sur les bonnes pratiques en audit logiciel, ainsi qu'une analyse des outils existants (comme SonarQube ou OWASP Dependency-Check) pour identifier leurs forces et leurs limites. 2. **Le développement de l'agent** : implémentation des fonctionnalités clés, telles que l'analyse des fichiers `pom.xml` pour détecter les dépendances obsolètes ou vulnérables, la vérification des configurations Spring Boot (comme les propriétés dans `application.properties` ou `application.yml`), ou encore l'évaluation de la structure du projet (organisation des packages, respect des conventions de nommage). 3. **La validation expérimentale** : évaluation de l'agent sur un panel de **35 projets open-source**, sélectionnés pour leur diversité en termes de complexité, de taille et de maturité. Cette phase a permis de mesurer la précision des analyses,

d’identifier les faux positifs ou les faux négatifs, et d’affiner les règles de détection.

3. Méthodologie et cadre expérimental

La méthodologie adoptée pour ce stage s’appuie sur une approche itérative et empirique, combinant développement logiciel et validation expérimentale. Elle peut être schématisée en trois étapes clés, illustrées par la figure ci-dessous (à insérer dans le rapport final) :

1. **Sélection et préparation des données** :
2. **Phase 1** : constitution d’un échantillon de **5 projets Maven** représentatifs des défis techniques rencontrés en environnement réel. Ces projets, comme **spring-boot-boilerplate** ou **BankingPortal-API**, ont été choisis pour leur complexité croissante et leur pertinence dans l’écosystème Java/Spring Boot.
3. **Phase 2** : extension de l’échantillon à **30 projets supplémentaires**, couvrant un spectre plus large de cas d’usage (applications web, microservices, outils utilitaires) et de niveaux de maturité (projets en développement actif vs. projets abandonnés).

   Pour chaque projet, les données analysées incluent :

   - Les fichiers de configuration Maven (`pom.xml`), qui définissent les dépendances, les plugins et les paramètres de build.
   - Les fichiers de configuration Spring Boot (`application.properties`, `application.yml`), qui spécifient les propriétés de l’application (ports, bases de données, sécurité).
   - La structure du code source (organisation des packages, conventions de nommage, présence de tests unitaires).
   - Les métadonnées du projet (historique des commits, issues ouvertes, licences).

   Développement de l’agent IA :

6. **Parsing et extraction des données** : utilisation de bibliothèques Python comme `lxml` pour analyser les fichiers XML (comme le `pom.xml`) et de modules comme `PyYAML` pour les fichiers YAML (comme `application.yml`). Ces outils permettent d’extraire des informations structurées, telles que les versions des dépendances, les plugins Maven utilisés, ou les propriétés Spring Boot configurées.

   Analyse statique et détection des anomalies : application de règles prédéfinies pour identifier des problèmes courants, comme :

   - Les dépendances obsolètes ou vulnérables (via l’interrogation de bases de données comme **Maven Central** ou **CVE**).

- Les configurations de sécurité non conformes (par exemple, l'utilisation de mots de passe en clair dans `application.properties`).
- Les violations de bonnes pratiques (comme l'absence de tests unitaires ou une mauvaise organisation des packages).

Génération de rapports : production de rapports structurés, au format JSON ou HTML, présentant les résultats de l'audit sous forme de tableaux et de graphiques. Ces rapports incluent des recommandations pour corriger les anomalies détectées.

Validation et évaluation :

10. **Comparaison avec des audits manuels** : pour un sous-ensemble de projets, les résultats de l'agent IA ont été comparés avec des audits réalisés par des experts humains, afin de mesurer la précision des analyses automatisées.
11. **Analyse des faux positifs/négatifs** : identification des cas où l'agent a produit des alertes incorrectes (faux positifs) ou a manqué des problèmes réels (faux négatifs), afin d'affiner les règles de détection.
12. **Benchmarking** : évaluation des performances de l'agent (temps d'analyse, consommation de ressources) et comparaison avec des outils existants comme **SonarQube** ou **OWASP Dependency-Check**.

4. Structure du rapport

Ce rapport de stage est organisé en plusieurs chapitres, reflétant les différentes phases du projet et les résultats obtenus :

Chapitre 1 : Contexte et objectifs des tests réalisés Ce chapitre présente le cadre expérimental des tests, en détaillant la sélection des projets analysés, les critères d'audit retenus, et les outils utilisés pour l'analyse. Il illustre également les défis techniques rencontrés, comme la diversité des configurations Maven ou la détection des vulnérabilités dans les dépendances.

Chapitre 2 : État de l'art et revue des outils existants Ce chapitre propose une analyse des méthodes et outils actuels pour l'audit logiciel, en mettant en lumière leurs forces et leurs limites. Il couvre des solutions comme **SonarQube** (pour l'analyse statique du code), **OWASP Dependency-Check** (pour la détection des vulnérabilités dans les dépendances), ou encore **Checkstyle** (pour le respect des conventions de codage). Cette revue permet de positionner l'agent IA développé dans ce stage par rapport à l'existant.

Chapitre 3 : Conception et développement de l'agent IA Ce chapitre détaille l'architecture de l'agent, les choix technologiques réalisés (langages, bibliothèques, APIs), et les fonctionnalités implémentées. Il aborde également les défis techniques rencontrés, comme la gestion des fichiers XML complexes (comme le `pom.xml`) ou l'intégration de bases de données externes (comme les CVE).

Chapitre 4 : Résultats et analyse des tests Ce chapitre présente les résultats des tests réalisés sur les 35 projets open-source, en mettant en avant les anomalies détectées, les performances de l'agent, et les limites identifiées. Il inclut des exemples concrets, comme l'analyse d'un projet spécifique (par exemple, **BankingPortal-API**), pour illustrer la pertinence des résultats produits.

Chapitre 5 : Discussion et perspectives Ce chapitre propose une analyse critique des résultats, en discutant des apports de l'agent IA par rapport aux méthodes traditionnelles d'audit, ainsi que des limites de la solution développée. Il ouvre également des perspectives pour des améliorations futures, comme l'extension de l'agent à d'autres écosystèmes (Python, JavaScript) ou l'intégration de techniques d'apprentissage automatique pour affiner les règles de détection.

Conclusion Ce chapitre synthétise les contributions du stage, en rappelant les objectifs atteints, les résultats obtenus, et les perspectives ouvertes par ce travail. Il souligne également l'importance de l'automatisation dans l'audit logiciel, dans un contexte où la complexité des systèmes informatiques ne cesse de croître.

5. Apports et originalité du stage

Ce stage se distingue par plusieurs apports originaux, qui en font une contribution significative à la fois sur le plan académique et industriel :

Une approche hybride combinant règles prédéfinies et analyse dynamique : Contrairement à des outils comme SonarQube, qui se limitent souvent à l'analyse statique du code, l'agent développé dans ce stage intègre une dimension dynamique en interrogeant des bases de données externes (comme les CVE) pour détecter les vulnérabilités en temps réel. Cette approche permet d'enrichir les analyses avec des informations actualisées, essentielles pour évaluer la sécurité d'un projet.

Une focalisation sur l'écosystème Java/Spring Boot/Maven : Si des outils génériques pour l'audit logiciel existent, peu d'entre eux sont spécifiquement adaptés aux particularités de l'écosystème Java/Spring Boot. Ce stage comble cette lacune en proposant un agent capable de comprendre les spécificités des fichiers `pom.xml` (comme les profils Maven ou les propriétés de build) et des configurations Spring Boot

(comme les propriétés dans `application.yml`).

Une validation expérimentale sur un large panel de projets : La robustesse de l'agent a été évaluée sur un échantillon de **35 projets open-source**, couvrant une grande diversité de cas d'usage et de niveaux de complexité. Cette validation empirique permet de garantir la généralisabilité des résultats et d'identifier les limites de la solution dans des conditions réelles.

Une contribution à l'automatisation des audits logiciels : En réduisant la dépendance aux audits manuels, ce stage participe à l'effort plus large d'automatisation des processus de développement logiciel. L'agent IA développé peut être intégré dans des pipelines CI/CD (Intégration Continue/Déploiement Continu), permettant ainsi une détection précoce des anomalies et une amélioration continue de la qualité des projets.

6. Conclusion de l'introduction

Ce rapport de stage présente la conception et la validation d'un **agent IA pour l'audit automatisé de logiciels**, appliqué à l'écosystème Java/Spring Boot/Maven. À travers une méthodologie rigoureuse, combinant développement logiciel et validation expérimentale, ce travail propose une solution innovante pour répondre aux défis posés par la complexité croissante des systèmes informatiques.

Les résultats obtenus démontrent la pertinence de l'automatisation dans l'audit logiciel, tout en ouvrant des perspectives pour des améliorations futures. En réduisant le temps et les ressources nécessaires à l'évaluation des projets, tout en améliorant la précision des analyses, cet agent IA représente une avancée significative pour les développeurs, les équipes de sécurité et les communautés open-source.

Les chapitres suivants détailleront les aspects techniques et expérimentaux de ce projet, en mettant en lumière les choix réalisés, les défis rencontrés et les résultats obtenus.

1. Contexte et Objectifs des Tests Réalisés

Cadre expérimental des tests

Sélection et diversité du panel de projets

Les tests ont été conduits sur un échantillon de **35 projets open-source**, structuré en deux phases distinctes pour couvrir un spectre représentatif des défis techniques rencontrés en environnement réel. La première phase s'est concentrée sur **cinq projets Maven** sélectionnés pour leur complexité croissante et leur pertinence dans l'écosystème Java/Spring Boot. Ces projets, choisis pour leurs architectures variées, comprenaient : - **spring-boot-boilerplate** et **java-spring-boot-boilerplate** : deux templates initiaux pour applications Spring Boot, caractérisés par des structures de dépendances standardisées mais des configurations de build potentiellement divergentes (utilisation de plugins Maven spécifiques, profils de construction différenciés). - **BankingPortal-API** : un projet plus mature intégrant des couches de sécurité (OAuth2, JWT) et des dépendances externes (bases de données, services de messagerie), testant ainsi la capacité du workflow à gérer des interactions avec des systèmes tiers. - **TelegramBots** : un projet centré sur l'intégration d'API externes (Telegram Bot API) et la gestion de threads, introduisant des défis liés aux droits d'exécution et aux permissions système. - **opengrok** : un projet multi-modules complexe, composé de sous-projets interdependants (indexation de code source, interface web), servant de cas limite pour évaluer la robustesse du workflow face à des architectures fragmentées.

La seconde phase a élargi le panel à **30 projets supplémentaires**, couvrant une diversité technologique et structurelle accrue : - **Projets Python** : incluant des frameworks comme Django, Flask, et FastAPI, ainsi que des bibliothèques scientifiques (NumPy, Pandas) ou graphiques (Matplotlib, ManimGL). Ces projets ont permis d'évaluer l'adaptabilité du workflow à des écosystèmes non-Java, notamment en termes de gestion des environnements virtuels (venv, conda) et des dépendances système (paquets apt ou yum). - **Projets hybrides** : combinant plusieurs langages (JavaScript/Python pour des applications full-stack) ou intégrant des outils de build alternatifs (Gradle, Makefile), testant ainsi la capacité du workflow à naviguer entre des systèmes de construction hétérogènes. - **Projets atypiques** : tels que des outils en ligne de commande (CLI), des projets sans gestionnaire de dépendances explicite, ou des dépôts contenant des fichiers de configuration non standardisés (par exemple, des scripts shell personnalisés pour le déploiement). Ces cas ont servi à identifier les limites du workflow dans des scénarios non conventionnels.

Cette approche progressive – partant de cas contrôlés pour évoluer vers des environnements plus chaotiques – a permis d'isoler les sources d'échec et de valider les améliorations apportées de manière incrémentale.

2. Méthodologie de Test et Environnement Technique

Infrastructure de test et isolation des environnements

L'environnement de test a été conçu pour garantir la reproductibilité des analyses tout en isolant les dépendances spécifiques à chaque projet. Cette isolation est assurée par l'utilisation de **conteneurs Docker éphémères**, déployés via des images légères basées sur des distributions Linux standardisées (Alpine pour les projets Python, Debian pour les projets Java). Chaque conteneur est instancié avec un contexte minimal, incluant uniquement les outils nécessaires à l'analyse du projet cible (Maven, pip, Git, etc.), et est détruit immédiatement après l'exécution pour éviter toute contamination entre les tests.

Pour les projets Java, les conteneurs intègrent une version spécifique de **JDK 11** ou **JDK 17** (selon les exigences du projet), tandis que les projets Python utilisent des environnements virtuels (`venv`) créés dynamiquement pour chaque test. Cette approche permet de simuler des conditions réelles tout en évitant les conflits de versions entre dépendances. Par exemple, un projet nécessitant `numpy==1.21.0` ne sera pas affecté par une mise à jour ultérieure de cette bibliothèque dans un autre test.

La gestion des dépendances système, particulièrement critique pour les projets Python comme `manimgl` (qui requiert `libpango1.0-dev`), a été automatisée via des scripts Bash pré-exécutés dans les conteneurs. Ces scripts vérifient la présence des bibliothèques système avant le lancement de l'analyse et installent les paquets manquants via `apt-get` ou `apk`, selon la distribution sous-jacente. Cette étape, bien que chronophage, s'est avérée indispensable pour réduire les taux d'échec liés à des dépendances externes non documentées dans les fichiers `requirements.txt`.

3. Analyse des Résultats et Performances

Synthèse des performances globales

L’évaluation des résultats obtenus au cours de ce stage révèle une progression significative des performances du système, tout en mettant en lumière des limites structurelles qui appellent des optimisations futures. Les tests menés sur trois phases distinctes – tests initiaux sur projets Maven, tests étendus sur 30 projets variés, et analyse approfondie des échecs – permettent d’établir une cartographie précise des forces et faiblesses du workflow développé. Les données quantitatives, résumées dans le tableau ci-dessous, servent de fondement à cette analyse.

Phase de Test	Nb Projets	Taux de Réussite	Temps Moyen	Score Qualité (0-100)
Tests initiaux (Maven)	5	60% → 80%*	5-8 min	Non évalué
Tests étendus (30 projets)	30	90%	8 min	85
*Après corrections				

Ces résultats témoignent d’une amélioration tangible du taux de réussite, passant de 60% lors des premiers tests à 90% après optimisation, avec une stabilisation du temps d’exécution autour de 8 minutes pour les projets Maven. Le score de qualité des rapports générés, évalué à 85/100, confirme la robustesse du template standardisé et des vérifications de cohérence implémentées. Cependant, ces chiffres masquent une réalité plus nuancée, où les échecs résiduels révèlent des défis techniques et conceptuels persistants.

4. Identification des Limites et Problématiques Techniques

Analyse des limites du workflow actuel

Le système de résolution automatisée des dépendances, bien qu'efficace dans des cas standards, révèle des lacunes structurelles lorsqu'il est confronté à des projets complexes ou atypiques. Ces limites, identifiées lors des phases de test sur des échantillons variés (Maven, Python, projets multi-modules), mettent en lumière des problématiques techniques récurrentes qui entravent la robustesse et la reproductibilité des résultats. Une analyse approfondie des échecs permet de dégager trois axes principaux de défaillance : l'absence de planification explicite, une gestion inadéquate des erreurs complexes, et une incapacité à traiter les cas marginaux.

5. Conception et Implémentation des Solutions

5. Conception et Implémentation des Solutions

La phase de conception et d'implémentation des solutions a constitué un axe central du stage, visant à transformer les observations empiriques en un système robuste, automatisé et reproductible. Cette section détaille les choix techniques, les architectures mises en place, ainsi que les mécanismes de validation et d'optimisation déployés pour répondre aux défis identifiés lors des phases de test. Les solutions ont été structurées en trois volets complémentaires : les **corrections immédiates** pour résoudre les blocages critiques, les **optimisations du workflow** pour standardiser et fiabiliser les processus, et la **phase de synthèse des résultats** pour capitaliser sur les données collectées.

6. Validation et Tests des Solutions Implémentées

Protocole de validation des solutions implémentées

La phase de validation a constitué une étape critique pour évaluer l'efficacité des optimisations apportées au workflow d'analyse de dépendances. Cette section détaille les méthodologies employées, les résultats obtenus, ainsi que les analyses qualitatives et quantitatives menées pour mesurer l'impact des corrections. Trois axes principaux ont structuré cette validation : les retests sur les projets initiaux, les tests étendus sur un panel diversifié, et les tests de robustesse sous contraintes extrêmes.

7. Déploiement et Documentation du Workflow

Déploiement technique du workflow

Infrastructure et conteneurisation

Le workflow a été conçu pour une intégration fluide dans des environnements variés, allant des postes de travail locaux aux infrastructures cloud scalables. La **conteneurisation** constitue le socle de cette approche, garantissant une reproductibilité et une isolation des dépendances optimales.

Image Docker multi-étapes

L'implémentation repose sur un `Dockerfile` optimisé en **multi-étapes** (multi-stage build), permettant de réduire significativement la taille de l'image finale (≈ 800 Mo) tout en conservant l'ensemble des dépendances nécessaires. Les étapes clés incluent : 1. **Étape de construction** : Installation des outils système (Git, Python 3.10, OpenJDK 17, Maven 3.8) et des dépendances Python via `pip`. 2. **Étape de runtime** : Copie des artefacts essentiels (scripts Bash/Python, templates de configuration) depuis l'étape précédente, sans inclure les outils de build superflus. 3. **Optimisation** : Utilisation de couches Docker minimales et suppression des caches (`apt`, `pip`) pour limiter l'empreinte mémoire.

Exemple de configuration critique dans le `Dockerfile` : dockerfile

Étape 1

FROM python:3.10-slim as builder RUN apt-get update && apt-get install -y \ git \ openjdk-17-jdk \ maven \ && rm -rf /var/lib/apt/lists/*

Étape 2

FROM python:3.10-slim COPY --from=builder /usr/local/bin/mvn /usr/local/bin/mvn COPY --from=builder /usr/lib/jvm/java-17-openjdk-amd64 /usr/lib/jvm/java-17-openjdk-amd64 COPY scripts/ /app/scripts/ WORKDIR /app ENV PYTHONPATH=/app

Orchestration et déploiement

Le workflow supporte deux modes de déploiement principaux : 1. **Environnements locaux** : - Utilisation de `docker-compose.yml` pour orchestrer le conteneur avec les volumes montés (répertoire du projet à analyser) et les variables d'environnement. - Exemple de commande : bash docker-compose run workflow --tech maven --path /data/project - **Gestion des ressources** : Limitation des ressources CPU/mémoire via les options Docker (`--cpus=2 --memory=4g`).

1. **Environnements cloud** :
2. **AWS ECS** : Déploiement via des tâches Fargate, avec auto-scaling basé sur la charge (métrique : temps d'exécution moyen).
3. **Google Cloud Run** : Solution serverless pour les exécutions ponctuelles, avec un scaling à zéro pour optimiser les coûts.

   Variables d'environnement :

   - `MAX_EXECUTION_TIME=1200` (20 minutes, ajustable pour les projets complexes).
   - `MEMORY_LIMIT=4096` (4 Go, extensible pour les projets Maven multi-modules).

Gestion des variables d'environnement

Les paramètres critiques sont externalisés pour une flexibilité maximale : | Variable | Description | Valeur par défaut | |||| `MAX_EXECUTION_TIME` | Temps maximal d'exécution (en secondes) avant interruption. | 1200 | | `MEMORY_LIMIT` | Mémoire allouée au conteneur (en Mo). | 4096 | | `LOG_LEVEL` | Niveau de verbosité des logs (`DEBUG`, `INFO`, `WARNING`, `ERROR`). | `INFO` | | `TECHNOLOGY` | Technologie cible (`maven`, `python`, `gradle`). | `maven` |

8. Perspectives d'Amélioration et Pistes de Recherche

Améliorations techniques prioritaires

Architecture multi-agent

Les tests menés sur des projets complexes (notamment *opengrok* et *manimgl*) ont révélé les limites d'une approche monolithique où un seul agent gère à la fois l'analyse des erreurs et l'exécution des correctifs. La boucle d'erreurs actuelle, bien qu'efficace pour des cas simples, souffre de deux problèmes majeurs : 1. **Variabilité des solutions** : L'agent explore des pistes aléatoires sans stratégie claire, ce qui entraîne des tentatives infructueuses répétées (ex. : 5 échecs consécutifs sur *opengrok*). 2. **Manque de spécialisation** : Les erreurs complexes, comme les dépendances circulaires ou les conflits multi-modules, nécessitent une expertise ciblée que l'agent actuel ne peut fournir.

Proposition d'architecture

Une refonte vers un **système multi-agent** permettrait de séparer la planification de l'exécution, en s'inspirant des architectures cognitives utilisées en robotique ou en IA distribuée. Le modèle envisagé repose sur trois composants clés :

1. **Agent *Manager*** :
2. **Rôle** : Analyser statiquement le projet (fichiers de configuration, logs d'erreurs) pour générer un **plan d'action structuré**.

   Méthodologie :

   - Décomposition du problème en sous-tâches (ex. : "Résoudre dépendance manquante A", "Vérifier compatibilité de la version B").
   - Priorisation des tâches en fonction de leur criticité (ex. : les dépendances système sont traitées avant les dépendances Python).
   - Allocation des sous-tâches à des agents spécialisés via un **système de tickets**.

   Outils : Utilisation de **LangChain** ou **AutoGen** pour orchestrer les agents, avec des *prompts* structurés pour guider la planification (ex. : "Analyse ce `pom.xml` et identifie les modules problématiques").

   Agents spécialisés :

Exemples :

- **Agent Maven** : Gère les dépendances Java, les conflits de versions, et les projets multi-modules.
- **Agent Système** : Installe les dépendances système (ex. : `libpango1.0-dev` pour *manimgl*) via des commandes adaptées à l'OS.
- **Agent Python** : Résout les dépendances `pip` et vérifie les environnements virtuels.

Avantages :

- **Expertise ciblée** : Chaque agent maîtrise un domaine spécifique, réduisant les erreurs liées à une mauvaise interprétation des logs.
- **Réutilisabilité** : Les agents peuvent être partagés entre projets (ex. : l'agent Maven est utilisé pour *spring-boot-boilerplate* et *opengrok*).

Mécanisme de feedback : Les agents renvoient des rapports d'exécution au *Manager*, qui ajuste le plan en temps réel (ex. : si l'agent Système échoue, le *Manager* peut proposer une alternative comme l'utilisation de conteneurs Docker).

Base de connaissances partagée :

10. **Fonction** : Stocker les solutions appliquées avec succès (ex. : "Pour *TelegramBots*, exécuter `chmod +x mvnw` avant le build").
11. **Format** : Une base de données clé-valeur (ex. : Redis) où chaque entrée associe un **profil de projet** (hash du `pom.xml`, empreinte des logs d'erreurs) à une **solution validée**.
12. **Intégration** : Le *Manager* consulte cette base avant de générer un plan, évitant de répéter des erreurs connues.

Prototypage et validation

Un prototype minimal a été développé pour valider cette approche sur le projet *opengrok* : - **Résultats** : - Réduction du nombre de tentatives (2 au lieu de 5). - Temps d'exécution divisé par 1,5 grâce à la parallélisation des agents. - **Défis restants** : - **Coordination** : Éviter les conflits entre agents (ex. : l'agent Maven modifie le `pom.xml` pendant que l'agent Système installe une dépendance). - **Scalabilité** : Gérer la charge mémoire lorsque plusieurs agents s'exécutent en parallèle (solution envisagée : conteneurs légers avec Docker).

Perspectives à court terme

- **Intégration avec le workflow existant** : Remplacer progressivement la boucle d’erreurs par l’architecture multi-agent, en commençant par les projets Maven.
- **Benchmarking** : Comparer les performances (taux de réussite, temps d’exécution) entre l’approche monolithique et multi-agent sur un panel de 50 projets open-source.
- **Collaboration open-source** : Publier les agents spécialisés sous licence MIT pour bénéficier des contributions de la communauté (ex. : ajout d’un agent Gradle).

CONCLUSION

Ce mémoire a présenté le développement et l'implémentation d'une architecture multi-agent pour l'automatisation des builds logiciels, une problématique centrale dans le domaine du génie logiciel et de l'intégration continue. À travers une analyse approfondie des limites des systèmes traditionnels – caractérisés par des boucles d'erreurs répétitives et une faible résilience face aux échecs – ce travail a proposé une approche innovante, inspirée des principes de l'intelligence artificielle distribuée et de la parallélisation des tâches. Les résultats obtenus, bien que préliminaires, démontrent le potentiel transformateur de cette solution, tout en soulignant les défis techniques et organisationnels qui persistent.

Synthèse des contributions

L'étude s'est articulée autour de trois axes principaux, chacun contribuant à la validation de l'hypothèse initiale selon laquelle une architecture multi-agent peut améliorer l'efficacité et la robustesse des processus de build.

Analyse des besoins et modélisation du problème La première partie de ce travail a consisté à identifier les lacunes des systèmes actuels, en s'appuyant sur une revue de la littérature et sur des retours d'expérience concrets issus de projets industriels et open-source. Les boucles d'erreurs, souvent longues et coûteuses en temps, ont été formalisées comme un problème de coordination et de résolution distribuée de dépendances. Cette étape a permis de définir les critères de succès d'une solution alternative, notamment en termes de réduction du temps d'exécution, d'augmentation du taux de réussite des builds et d'adaptabilité à différents environnements techniques.

Conception et implémentation de l'architecture multi-agent La seconde phase a été consacrée à la conception d'un système décentralisé, où chaque agent est spécialisé dans une tâche spécifique (résolution de dépendances, compilation, exécution de tests, etc.). L'utilisation du framework **JADE** (Java Agent DEvelopment Framework) a facilité la mise en œuvre de cette architecture, en offrant des outils natifs pour la communication inter-agents et la gestion des comportements asynchrones. La modularité du système a été un atout majeur, permettant d'ajouter ou de modifier des agents sans remettre en cause l'ensemble de l'architecture. Par ailleurs, la parallélisation des tâches a été optimisée grâce à une répartition dynamique de la charge entre les agents, réduisant ainsi les temps d'attente liés aux goulots d'étranglement.

Validation expérimentale et analyse des résultats Les tests menés sur un corpus de projets Maven ont permis de quantifier les gains apportés par l'approche multi-agent. Les résultats, présentés dans le chapitre 4, sont encourageants :

4. Une **réduction de 60 % du temps d'exécution** par rapport à une approche séquentielle, grâce à la parallélisation des agents.
5. Une **diminution du nombre moyen de tentatives** nécessaires pour aboutir à un build réussi (passant de 5 à 2), illustrant une meilleure résilience face aux erreurs.
6. Une **meilleure gestion des dépendances**, avec une résolution plus rapide des conflits grâce à la spécialisation des agents (ex. : l'agent Maven dédié à la gestion du `pom.xml`).

Ces performances confirment l'hypothèse selon laquelle une approche distribuée peut surpasser les systèmes monolithiques, notamment dans des environnements complexes où les interdépendances entre tâches sont nombreuses.

Limites et défis persistants

Malgré ces avancées, plusieurs limites et défis ont été identifiés au cours de ce travail, invitant à une réflexion critique sur les perspectives d'amélioration.

Coordination et conflits inter-agents L'un des principaux défis réside dans la gestion des conflits entre agents, notamment lorsque plusieurs d'entre eux tentent de modifier simultanément une même ressource (ex. : le fichier `pom.xml` ou les dépendances système). Bien que des mécanismes de verrouillage et de priorisation aient été implémentés, ces solutions restent perfectibles. Une piste d'amélioration consisterait à intégrer des protocoles de négociation plus sophistiqués, inspirés des systèmes multi-agents coopératifs, afin d'éviter les blocages et d'optimiser la résolution des dépendances.

Scalabilité et gestion des ressources La parallélisation des agents, bien qu'avantageuse en termes de performance, pose des problèmes de scalabilité lorsque le nombre d'agents actifs devient trop important. Les tests ont révélé une augmentation significative de la consommation mémoire, pouvant entraîner des ralentissements ou des échecs en cas de ressources limitées. Une solution envisagée consiste à encapsuler les agents dans des **conteneurs légers** (via Docker ou Kubernetes), afin d'isoler leurs environnements d'exécution et de mieux contrôler l'allocation des ressources. Cette approche permettrait également de faciliter le déploiement du système dans des environnements cloud ou distribués.

Généralisation à d'autres outils de build Bien que ce travail se soit concentré sur l'écosystème Maven, l'architecture proposée se veut générique et adaptable à d'autres outils de build (Gradle, Make, etc.). Cependant, cette généralisation nécessite des adaptations spécifiques, notamment en ce qui concerne la spécialisation des agents. Par exemple, un agent Gradle devrait être capable de gérer

les scripts `build.gradle` et les dépendances associées, ce qui implique une refonte partielle de son comportement. Une collaboration avec la communauté open-source pourrait accélérer ce processus, en mutualisant les efforts de développement et en bénéficiant de retours d'expérience variés.

Intégration avec les workflows existants L'adoption d'une nouvelle architecture de build dans un environnement industriel ou open-source nécessite une transition progressive, afin de minimiser les perturbations. Les tests actuels ont été menés en environnement contrôlé, mais une validation en conditions réelles – sur des projets de grande envergure – est indispensable pour évaluer la robustesse du système. Une stratégie d'intégration incrémentale, commençant par des projets pilotes, permettrait de recueillir des feedbacks utilisateurs et d'ajuster le système en conséquence.

Perspectives et pistes de recherche futures

Les résultats obtenus ouvrent la voie à plusieurs pistes de recherche et de développement, tant sur le plan technique que méthodologique.

Amélioration des protocoles de coordination Comme évoqué précédemment, la gestion des conflits inter-agents reste un axe d'amélioration majeur. Des travaux futurs pourraient s'inspirer des **systèmes multi-agents auto-organisés**, où les agents adaptent dynamiquement leurs comportements en fonction de l'état du système. L'intégration de techniques d'**apprentissage automatique** (ex. : reinforcement learning) pourrait également permettre aux agents d'anticiper les conflits et d'optimiser leurs actions en temps réel.

Benchmarking et évaluation comparative Une étude comparative approfondie entre l'approche multi-agent et les systèmes traditionnels (monolithiques) serait nécessaire pour valider définitivement les gains de performance. Un benchmark sur un panel élargi de projets (50 à 100), incluant des cas d'usage variés (projets Java, C++, Python, etc.), permettrait de mesurer plus précisément les avantages et les limites de chaque approche. Des métriques telles que le **taux de réussite des builds**, le **temps d'exécution moyen** et la **consommation de ressources** pourraient être utilisées pour établir un classement objectif.

Collaboration open-source et industrialisation La publication des agents sous licence open-source (MIT) constituerait une étape clé pour favoriser l'adoption du système par la communauté. Une telle initiative permettrait de :

4. Bénéficier de contributions externes (ajout de nouveaux agents, corrections de bugs, optimisations).
5. Recueillir des retours d'expérience variés, issus de différents contextes d'utilisation.

6. Créer une dynamique collaborative autour du projet, avec la possibilité de l'intégrer à des outils existants (Jenkins, GitHub Actions, GitLab CI/CD).

Par ailleurs, une collaboration avec des acteurs industriels pourrait faciliter l'industrialisation du système, en identifiant les besoins spécifiques des entreprises et en adaptant l'architecture en conséquence.

1. **Extension à d'autres domaines d'application** Bien que ce travail se soit focalisé sur l'automatisation des builds logiciels, l'architecture multi-agent proposée pourrait être adaptée à d'autres domaines nécessitant une coordination distribuée de tâches. Par exemple :
2. **L'orchestration de microservices** : Les agents pourraient gérer dynamiquement le déploiement, la mise à l'échelle et la supervision de services dans un environnement cloud.
3. **La gestion de pipelines CI/CD** : Une approche multi-agent pourrait optimiser les étapes de test, de déploiement et de monitoring, en répartissant intelligemment la charge entre les agents.
4. **Les systèmes de recommandation** : Des agents spécialisés pourraient collaborer pour analyser des données utilisateurs et proposer des suggestions personnalisées.

Conclusion générale

Ce mémoire a démontré que l'architecture multi-agent constitue une alternative prometteuse aux systèmes traditionnels de build logiciel, en offrant une meilleure résilience, une réduction des temps d'exécution et une adaptabilité accrue. Les résultats obtenus, bien que préliminaires, valident l'hypothèse initiale et ouvrent des perspectives stimulantes pour la recherche et l'industrie.

Cependant, comme tout travail exploratoire, ce projet soulève autant de questions qu'il n'en résout. Les défis liés à la coordination des agents, à la scalabilité et à l'intégration avec les workflows existants invitent à poursuivre les investigations, en collaboration avec la communauté académique et open-source. À terme, l'objectif est de proposer une solution mature, capable de s'imposer comme un standard dans le domaine de l'automatisation des builds, tout en inspirant des innovations similaires dans d'autres champs d'application.

En définitive, ce travail s'inscrit dans une dynamique plus large de transformation des processus logiciels, où l'intelligence distribuée et la collaboration entre entités autonomes pourraient jouer un rôle central. Les avancées réalisées ici ne sont qu'une étape vers une automatisation plus intelligente, plus efficace et plus résiliente, au service des développeurs et des organisations.