RAPPORT DE STAGE

Fin d'Études - Ingénieur Informatique

Conception et développement d'un agent IA pour l'audit automatisé de logiciels : amélioration de la sécurité et de la qualité du code

Yvain Tellier

yvain.tellier@gmail.com

master WeDSci
ULCO

01/03/2025

Entreprise d'accueil

Diag n' Grow

Geoffrey Pruvost

Tuteur Académique

Mentor Académique Tuteur Pédagogique Responsable d’Accompagnement Coach Études Advisor Master Guide Formation Expert Intégration Conseiller Scientifique Superviseur Pédagogique Tuteur IA

Février 2026

AVANT-PROPOS

Ce stage de fin d'études marque l'aboutissement de deux années d'apprentissage intensif au sein du Master WeDSci (Web, Data Science et Intelligence Artificielle) de l'Université du Littoral Côte d'Opale (ULCO). Conçu comme une passerelle entre le monde académique et le milieu professionnel, ce parcours m'a permis d'acquérir des compétences techniques pointues en développement logiciel, en traitement des données et en intelligence artificielle, tout en cultivant une vision stratégique des enjeux numériques contemporains.

Le choix de ce stage s'inscrit dans une volonté de confronter mes connaissances théoriques à des problématiques concrètes, tout en contribuant à l'innovation dans un domaine en pleine mutation. Intégrer Diag n'Grow, une entreprise spécialisée dans l'édition de logiciels et l'audit numérique, représentait une opportunité idéale pour appliquer ces compétences dans un environnement exigeant, où la qualité, la sécurité et l'automatisation des processus sont des priorités absolues.

La mission qui m'a été confiée – concevoir un agent IA dédié à l'audit automatisé de logiciels – reflète parfaitement les défis actuels du secteur. Elle exigeait non seulement une maîtrise des outils et des frameworks modernes, mais aussi une capacité à appréhender les enjeux métiers de l'entreprise, tels que l'optimisation des workflows, la traçabilité des codes sources et la détection proactive des vulnérabilités. Ce projet, à la croisée de l'ingénierie logicielle et de l'intelligence artificielle, m'a permis de mesurer l'importance d'une approche rigoureuse, alliant méthodologie scientifique et pragmatisme opérationnel.

Par ailleurs, ce stage a été l'occasion de développer des compétences transversales essentielles, telles que la gestion de projet, la collaboration en équipe pluridisciplinaire et la communication technique. L'encadrement bienveillant de mon maître de stage, Geoffrey Pruvost, ainsi que les échanges réguliers avec mon tuteur académique, ont été déterminants pour structurer ma réflexion et affiner mes choix techniques. Leur expertise m'a offert un cadre propice à l'expérimentation et à l'apprentissage, tout en m'incitant à questionner en permanence la pertinence de mes solutions.

Enfin, ce stage a renforcé ma conviction quant à l'importance de l'innovation dans le domaine du numérique. À l'heure où les entreprises doivent sans cesse s'adapter aux évolutions technologiques, les solutions automatisées, comme l'agent IA que j'ai contribué à développer, jouent un rôle clé dans l'amélioration de la productivité et de la sécurité des systèmes informatiques. Cette expérience a ainsi confirmé mon désir de m'orienter vers des métiers à la frontière de la recherche appliquée et du développement logiciel, où l'intelligence artificielle et la data science ouvrent des perspectives inédites.

Fait à Calais, le 15 February 2026

REMERCIEMENTS

Ce stage de fin d'études, réalisé dans le cadre du **Master WeDSci (Web, Data Science et Intelligence Artificielle)** de l'**Université du Littoral Côte d'Opale (ULCO)**, a été une expérience professionnelle et humaine enrichissante. Il m'a permis de concrétiser les connaissances acquises durant ma formation tout en découvrant les enjeux concrets de la **conception d'agents IA pour l'audit logiciel**. À l'issue de cette immersion au sein de **Diag n'Grow**, je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué à la réussite de ce projet.

Je remercie tout particulièrement **Geoffrey Pruvost**, mon maître de stage chez Diag n'Grow, pour son **encadrement technique rigoureux** et sa **disponibilité constante**. Ses conseils avisés, son expertise en **développement logiciel et en intégration d'agents IA**, ainsi que sa confiance dans la réalisation de ce projet, ont été déterminants pour mener à bien cette mission. Son accompagnement m'a permis de **comprendre les défis de l'automatisation des audits** et de m'adapter efficacement aux exigences d'un environnement professionnel innovant.

Ma reconnaissance s'adresse également à mon **tuteur académique**, désigné par l'ULCO, pour son **suivi pédagogique attentif** et ses **orientations constructives**. Son expertise en **intelligence artificielle, génie logiciel et gestion de projets** a été précieuse pour structurer ce stage et en garantir la cohérence avec les objectifs du Master WeDSci. Ses retours réguliers et ses recommandations ont grandement contribué à la **qualité de ce rapport** et à la pertinence des solutions proposées.

Je souhaite aussi remercier l'**équipe enseignante du Master WeDSci** pour la **qualité de la formation** dispensée, alliant **théorie et pratique** pour préparer les étudiants aux enjeux du numérique. Leurs enseignements en **Big Data, architectures logicielles et IA** ont été des piliers essentiels pour aborder ce stage avec confiance et compétence.

Enfin, je tiens à exprimer ma gratitude à l'ensemble des **collaborateurs de Diag n'Grow** pour leur **accueil chaleureux** et leur **soutien tout au long de ce stage**. Leur expertise et leur dynamisme ont créé un environnement propice à l'**innovation et à l'apprentissage**, me permettant de m'intégrer rapidement et de contribuer activement au projet.

Ce stage a été une étape marquante dans mon parcours académique et professionnel, et je mesure pleinement la chance que j'ai eue de travailler aux côtés de professionnels aussi engagés. Qu'ils trouvent ici l'expression de ma sincère reconnaissance.

SOMMAIRE

---

1. Cadre institutionnel et professionnel du stage

Le présent rapport de stage s'inscrit dans le cadre du **Master WeDSci (Web, Data Science et Intelligence Artificielle)** dispensé par l'**Université du Littoral Côte d'Opale (ULCO)**, une formation de niveau Bac+5 conçue pour former des **ingénieurs et chercheurs en informatique** capables de concevoir, déployer et optimiser des **systèmes complexes** à l'intersection du **Web, du Big Data et de l'Intelligence Artificielle (IA)**. Ce diplôme, reconnu pour son **approche professionnalisante**, combine des enseignements théoriques avancés avec des **projets appliqués** et des **stages en entreprise**, permettant aux étudiants d'acquérir une **double compétence** : une **maîtrise technique** des outils modernes (frameworks, langages, architectures cloud) et une **capacité à innover** dans des domaines en pleine mutation, tels que l'**automatisation des processus**, la **sécurité logicielle** ou l'**intégration de l'IA dans les workflows métiers**.

C'est dans ce contexte que **Yvain Tellier**, étudiant en dernière année du Master WeDSci, a effectué un stage de six mois (du **1er mars 2025 au 30 août 2025**) au sein de l'entreprise **Diag n'Grow**, spécialisée dans le **développement, l'audit et l'optimisation de solutions**

logicielles. Ce stage, encadré conjointement par un **tuteur académique** de l'ULCO et un **maître de stage** au sein de l'entreprise, avait pour objectif principal la **conception et le développement d'un agent IA dédié à l'audit automatisé de logiciels**, avec une focalisation particulière sur l'**amélioration de la sécurité et de la qualité du code**. Ce projet, à la croisée de l'**ingénierie logicielle** et de l'**intelligence artificielle**, reflète les **enjeux stratégiques** auxquels sont confrontées les entreprises du secteur numérique, où la **complexité croissante des systèmes**, la **multiplication des dépendances logicielles** et les **exigences accrues en matière de cybersécurité** rendent indispensable l'automatisation des processus d'audit et de validation.

---

2. Contexte entrepreneurial et enjeux technologiques

2.1. Présentation de Diag n'Grow : un acteur clé de la transformation numérique

Fondée sur une expertise reconnue en **conception et édition de logiciels**, **Diag n'Grow** se positionne comme un partenaire stratégique pour les entreprises souhaitant **optimiser, sécuriser et valoriser leurs actifs numériques**. L'entreprise intervient sur l'ensemble du **cycle de vie des logiciels**, depuis leur **conception** jusqu'à leur **déploiement**, en passant par des **prestations d'audit, de refactoring et d'optimisation financière**. Dans un écosystème où les **risques liés aux vulnérabilités logicielles** (failles de sécurité, non-conformité aux standards, dette technique) peuvent avoir des **conséquences critiques** (pertes financières, atteintes à la réputation, sanctions réglementaires), Diag n'Grow propose des **solutions innovantes** pour **automatiser et industrialiser** les processus de contrôle qualité.

Le stage de Yvain Tellier s'inscrit directement dans cette dynamique. En effet, l'entreprise fait face à un **double défi** : 1. **L'augmentation exponentielle de la complexité des projets logiciels** : avec l'adoption massive de **microservices, de conteneurs (Docker, Kubernetes) et de dépendances tierces**, les équipes de développement peinent à maintenir une **vision globale et cohérente** de leurs applications. Les outils traditionnels d'audit (comme **SonarQube, Checkstyle ou PMD**) montrent leurs limites face à des **architectures distribuées** ou à des **projets multi-modules** (ex. : projets Maven complexes). 2. **La nécessité d'intégrer l'IA dans les workflows métiers** : pour répondre aux **exigences de rapidité et de précision** des audits, Diag n'Grow explore des solutions basées sur l'**intelligence artificielle**, capables d'**automatiser l'analyse des codes sources**, de **détecter des patterns de vulnérabilités** et de **proposer des corrections adaptées**. C'est dans ce cadre que s'inscrit le projet confié à Yvain Tellier : **développer un agent IA capable d'auditer automatiquement des projets logiciels**, en combinant **analyse statique, dynamique et raisonnement automatisé**.

2.2. Le projet de stage : un agent IA pour l'audit automatisé de logiciels

Le projet mené par Yvain Tellier chez Diag n'Grow vise à **concevoir et implémenter un agent IA** capable d'effectuer des **audits automatisés** de projets logiciels, avec pour objectifs principaux : - **L'identification des vulnérabilités de sécurité** (injections SQL, failles XSS, expositions de données sensibles). - **L'évaluation de la qualité du code** (respect des bonnes pratiques, complexité cyclomatique, dette technique). - **La détection des non-conformités** aux standards industriels (OWASP, CIS, normes internes). - **La génération de rapports d'audit** détaillés, incluant des **recommandations correctives** et des **métriques de risque**.

Pour ce faire, l'agent IA s'appuie sur une **architecture modulaire**, intégrant : 1. **Un module d'analyse statique** : pour parcourir le code source et identifier des **patterns de vulnérabilités** ou des **écarts par rapport aux bonnes pratiques**. 2. **Un module d'analyse dynamique** : pour exécuter des **tests unitaires et d'intégration**, et détecter des **comportements anormaux** (fuites mémoire, erreurs d'exécution). 3. **Un moteur de raisonnement automatisé** : basé sur des **modèles de langage (LLM)** et des **frameworks agentiques** (comme **LangChain ou CrewAI**), ce module permet à l'agent de **comprendre le contexte du projet**, de **corréler les résultats des analyses** et de **proposer des solutions adaptées**. 4. **Un système de génération de rapports** : pour produire des **documents structurés** (PDF, Markdown) incluant des **visualisations** (graphiques, heatmaps) et des **recommandations priorisées**.

Ce projet illustre les **défis technologiques** auxquels Diag n'Grow est confrontée, notamment : - **L'intégration de l'IA dans des workflows existants** : comment faire coexister des outils traditionnels (comme **Maven, Jenkins ou GitLab CI**) avec des **solutions basées sur l'IA** ? - **La gestion des projets complexes** : comment auditer efficacement des **projets multi-modules** (ex. : *opengrok*, *manimgl*) ou des **dépendances hybrides** (système + Maven) ? - **La scalabilité et la performance** : comment garantir que l'agent IA puisse **traiter des projets volumineux** sans impacter les performances des pipelines d'intégration continue (CI/CD) ?

---

3. Problématique et enjeux académiques

3.1. Une problématique à l'intersection de l'IA et du génie logiciel

Le stage de Yvain Tellier soulève une **problématique centrale** pour les entreprises du secteur numérique : **comment automatiser et industrialiser l'audit de logiciels à grande échelle, tout en garantissant une détection précise des vulnérabilités et une amélioration continue de la qualité du code ?** Cette question s'articule autour de plusieurs **enjeux clés** : 1. **L'automatisation des audits** : les outils traditionnels (comme **SonarQube ou Fortify**) nécessitent souvent une **configuration manuelle** et une **interprétation humaine** des résultats. Comment les **agents IA** peuvent-ils **automatiser ces tâches** tout en maintenant un **niveau de précision élevé** ? 2. **L'adaptation aux projets**

complexes : les architectures logicielles modernes (microservices, conteneurs, multi-modules) rendent les audits **plus difficiles à réaliser**. Comment concevoir un agent IA capable de **comprendre la structure d'un projet** et d'**identifier les dépendances critiques** ? 3. **L'intégration dans les pipelines CI/CD** : pour être efficace, un outil d'audit doit s'intégrer **nativement** dans les **workflows de développement** (GitLab CI, Jenkins, GitHub Actions). Comment garantir que l'agent IA puisse **s'exécuter en temps réel** sans ralentir les builds ? 4. **La génération de recommandations actionnables** : au-delà de la détection des vulnérabilités, l'agent IA doit **proposer des corrections adaptées** au contexte du projet. Comment entraîner des **modèles de langage** à générer des **solutions pertinentes et sécurisées** ?

3.2. Un projet ancré dans les défis du Master WeDSci

Ce stage s'inscrit pleinement dans les **objectifs pédagogiques** du Master WeDSci, qui vise à former des professionnels capables de : - **Concevoir des architectures logicielles complexes** : le projet d'agent IA nécessite une **compréhension approfondie** des **systèmes distribués**, des **APIs** et des **frameworks d'IA**. - **Maîtriser les outils d'automatisation** : l'intégration de l'agent dans des **pipelines CI/CD** implique une **expertise en DevOps** (Docker, Kubernetes, scripts Bash/Python). - **Appliquer des méthodes d'IA avancées** : l'utilisation de **modèles de langage** et de **frameworks agentiques** (LangChain, CrewAI) permet à Yvain Tellier de **mettre en pratique** les enseignements reçus en **machine learning, traitement du langage naturel (NLP) et raisonnement automatisé**. - **Développer une réflexion critique sur les enjeux éthiques et sécuritaires** : l'audit de logiciels soulève des **questions sensibles** (protection des données, biais algorithmiques, conformité RGPD). Ce stage permet à l'étudiant d'aborder ces **défis sous un angle professionnel**.

---

4. Annonce du plan

Pour répondre à cette problématique et rendre compte des **travaux réalisés** au cours de ce stage, le présent rapport s'articule autour de **quatre parties principales** :

> **État de l'art des outils et méthodologies d'audit de logiciels** : Cette première partie propose une **analyse critique** des solutions existantes (outils d'analyse statique/dynamique, frameworks d'IA) en les confrontant aux **limites observées** dans des contextes réels (projets Maven multi-modules, dépendances hybrides). Elle met en lumière les **lacunes des approches traditionnelles** et identifie les **bonnes pratiques** issues de retours d'expérience (ex. : projets *opengrok*, *manimgl*).
>
> **Conception et architecture de l'agent IA** : Cette section détaille la **méthodologie de conception** de l'agent IA, depuis l'**analyse des besoins** jusqu'à la **modélisation de l'architecture**. Elle aborde les **choix technologiques** (frameworks, modèles de

langage, outils d'analyse) et les **défis rencontrés** (intégration des modules, gestion des dépendances, performance).

Implémentation et validation expérimentale : Ici, sont présentés les **étapes de développement** de l'agent IA, ainsi que les **tests réalisés** sur des projets réels. Cette partie inclut une **évaluation des performances** (précision, temps d'exécution, scalabilité) et une **analyse des résultats** obtenus (détection de vulnérabilités, génération de recommandations).

Bilan et perspectives : Enfin, cette dernière partie propose une **synthèse des apprentissages** et des **compétences acquises** par Yvain Tellier, ainsi qu'une **réflexion sur les limites** du projet et les **pistes d'amélioration** (intégration de nouveaux modèles d'IA, extension à d'autres langages, industrialisation du produit).

---

5. Conclusion de l'introduction

Ce stage, réalisé dans le cadre du **Master WeDSci de l'ULCO** et encadré par **Diag n'Grow**, représente une **opportunité unique** pour Yvain Tellier de **concilier expertise académique et application professionnelle**. En développant un **agent IA pour l'audit automatisé de logiciels**, il contribue à répondre à un **enjeu majeur** pour les entreprises du secteur numérique : **automatiser et optimiser les processus de contrôle qualité**, tout en intégrant les **dernières avancées en intelligence artificielle**.

Au-delà de sa dimension technique, ce projet soulève des **questions fondamentales** sur l'**avenir du génie logiciel** : comment l'IA peut-elle **transformer les pratiques de développement** ? Quels sont les **risques et les opportunités** liés à l'automatisation des audits ? Comment garantir que ces outils **restent accessibles, transparents et éthiques** ?

Ces interrogations, au cœur des **débats actuels** sur la **transformation numérique**, seront explorées tout au long de ce rapport, qui se veut à la fois un **compte-rendu technique** et une **réflexion sur les défis de demain**.

État de l'art des outils et méthodologies d'analyse de projets Maven

---

L’analyse des projets Maven, en particulier dans des contextes industriels ou académiques, repose sur un écosystème d’outils et de méthodologies dont la maturité et les limites influencent directement la qualité des builds, la détection des vulnérabilités et l’automatisation des pipelines. Cet état de l’art explore les solutions existantes en les confrontant aux défis posés par des projets complexes, notamment ceux structurés en multi-modules ou intégrant des dépendances hybrides (système et Maven). L’objectif est de mettre en lumière les lacunes des approches actuelles, tout en identifiant les bonnes pratiques qui émergent des retours d’expérience concrets, comme ceux observés lors des tests sur des projets tels qu’*opengrok* ou *manimgl*.

Les outils d’analyse statique et dynamique : forces et limites dans un contexte Maven

L’analyse des projets Maven s’articule principalement autour de deux familles d’outils : ceux dédiés à la gestion des dépendances et ceux axés sur la qualité du code ou la sécurité. Parmi les premiers, le **Maven Dependency Plugin** occupe une place centrale en permettant d’identifier les dépendances transitives, de générer des graphes de dépendances ou de détecter les conflits de versions. Son intégration native avec Maven en fait un outil incontournable pour les développeurs, mais ses capacités restent limitées à une analyse statique et déclarative. Par exemple, il ne permet pas de résoudre dynamiquement les dépendances système, comme l’a illustré l’échec partiel sur *manimgl*, où des bibliothèques comme *libpango1.0-dev* étaient requises sans être explicitement déclarées dans le *pom.xml*. Cette lacune révèle une limite fondamentale des outils basés sur Maven : leur incapacité à interagir avec l’environnement d’exécution, ce qui les rend inefficaces face à des projets hybrides combinant dépendances Java et dépendances système.

En parallèle, des outils comme **SonarQube** ou **OWASP Dependency-Check** étendent l’analyse à des dimensions de sécurité et de qualité. SonarQube, par exemple, offre une couverture exhaustive des métriques de code (complexité cyclomatique, duplication, couverture de tests), mais son efficacité dépend fortement de la configuration des règles et de l’intégration dans un pipeline CI/CD. Dans le cas de projets multi-modules comme

opengrok, SonarQube peut rencontrer des difficultés à agréger les résultats de manière cohérente, surtout si les modules ne sont pas analysés individuellement. OWASP Dependency-Check, quant à lui, se concentre sur la détection des vulnérabilités connues dans les dépendances, en s'appuyant sur des bases de données comme la *National Vulnerability Database* (NVD). Cependant, son approche réactive – basée sur des CVE déjà publiées – ne permet pas d'anticiper les risques liés à des dépendances obsolètes ou mal configurées. De plus, ces outils génèrent souvent des faux positifs ou des alertes peu prioritaires, ce qui peut noyer les équipes dans un bruit informationnel, comme l'a montré l'analyse des logs des tests sur *BankingPortal-API*, où plusieurs alertes mineures ont masqué un problème critique de conflit de versions.

Scan statique vs dynamique : une complémentarité nécessaire mais difficile à orchestrer

La distinction entre analyse statique et dynamique est cruciale pour comprendre les limites des outils actuels. Les scans statiques, comme ceux réalisés par le Maven Dependency Plugin ou SonarQube, analysent le code et les fichiers de configuration sans exécuter le projet. Cette approche présente l'avantage d'être rapide et reproductible, mais elle se heurte à des projets dont la complexité dépasse le cadre déclaratif. Par exemple, dans le cas d'*opengrok*, un projet multi-modules avec des dépendances conditionnelles, l'analyse statique ne parvient pas à résoudre les dépendances dynamiques ou les plugins Maven configurés pour des profils spécifiques. Les tests ont révélé que le scan global échouait systématiquement, nécessitant une intervention manuelle pour analyser chaque module séparément. Cette contrainte limite considérablement l'automatisation, surtout dans des pipelines où le temps d'exécution est un facteur critique.

À l'inverse, les approches dynamiques, comme celles utilisées par des outils de test d'intégration ou des frameworks de *fuzzing*, exécutent le code pour détecter des comportements anormaux. Cependant, leur intégration dans des pipelines Maven reste marginale en raison de leur coût en ressources et de leur complexité de configuration. Par exemple, les tests sur *TelegramBots* ont échoué en raison de permissions insuffisantes sur le script *mvnw*, un problème qui n'aurait pas été détecté par une analyse statique. Les frameworks de test comme **JUnit** ou **TestNG** jouent ici un rôle clé, car ils permettent de valider la robustesse des builds en simulant des scénarios d'exécution. Pourtant, leur efficacité dépend de la qualité des cas de test écrits par les développeurs, ce qui introduit une variabilité difficile à maîtriser dans des projets open source où les contributions sont hétérogènes.

Une solution émergente consiste à combiner les deux approches dans des pipelines hybrides, où une première phase statique identifie les risques potentiels, suivie d'une phase

dynamique pour valider les hypothèses. Cependant, cette orchestration reste complexe à mettre en œuvre, notamment en raison de l'absence de standards pour l'échange de données entre outils. Par exemple, les résultats de SonarQube ne sont pas directement exploitables par OWASP Dependency-Check, ce qui oblige les équipes à développer des scripts ad hoc pour corréler les alertes. Cette fragmentation des outils explique en partie pourquoi les tests sur *manimgl* ont échoué : l'agent n'a pas pu croiser les informations issues de l'analyse statique (dépendances Maven) avec celles de l'environnement système (dépendances *libpango*), faute d'un cadre unifié.

L'intégration des frameworks de test dans les pipelines CI/CD : un enjeu de robustesse

L'automatisation des tests dans les pipelines CI/CD est un pilier de la qualité logicielle, mais son efficacité dépend étroitement de l'intégration des frameworks de test avec Maven. **JUnit** et **TestNG** dominent ce paysage, chacun offrant des fonctionnalités adaptées à des besoins spécifiques. JUnit, avec son approche légère et son intégration native avec Maven via le *Surefire Plugin*, est souvent privilégié pour les tests unitaires. Cependant, sa simplicité devient un handicap pour des projets complexes comme *opengrok*, où les tests d'intégration nécessitent des configurations avancées (mocks, bases de données embarquées). TestNG, en revanche, offre une granularité supérieure grâce à ses annotations (*@BeforeGroups*, *@AfterGroups*) et sa gestion native des dépendances entre tests, ce qui en fait un choix pertinent pour les projets multi-modules. Pourtant, son adoption reste limitée en raison de sa courbe d'apprentissage plus raide et de sa compatibilité partielle avec certains plugins Maven.

L'un des défis majeurs réside dans la gestion des dépendances de test, notamment lorsque celles-ci entrent en conflit avec les dépendances de production. Par exemple, lors des tests sur *BankingPortal-API*, un conflit entre une version de *Mockito* utilisée pour les tests et une version requise par une dépendance de production a provoqué des échecs intermittents. Les solutions classiques, comme l'utilisation de l'élément *test* dans le *pom.xml*, ne suffisent pas toujours, car certaines dépendances transitives peuvent échapper à ce mécanisme. Une bonne pratique consiste à isoler les dépendances de test dans un profil Maven dédié, mais cette approche alourdit la configuration et peut introduire des incohérences si les profils ne sont pas maintenus rigoureusement.

Un autre enjeu est la couverture des tests dans des projets multi-modules. Les frameworks comme JUnit ou TestNG ne fournissent pas de mécanismes natifs pour agréger les résultats de tests exécutés sur plusieurs modules, ce qui oblige les équipes à recourir à des outils externes comme **Jacoco** pour la couverture de code ou **Allure** pour la génération de rapports. Cependant, ces outils ajoutent une couche de complexité supplémentaire, comme

l'ont montré les tests sur *opengrok*, où l'agrégation des rapports de couverture a échoué en raison d'incompatibilités entre les versions des plugins Maven utilisés. Cette fragmentation des outils souligne la nécessité d'une approche plus unifiée, où les résultats des tests seraient centralisés et analysés de manière cohérente, indépendamment du nombre de modules ou de la technologie sous-jacente.

Gestion des dépendances et droits d'exécution : des bonnes pratiques encore trop peu appliquées

La gestion des dépendances dans les projets Maven repose sur un ensemble de bonnes pratiques dont l'application rigoureuse permet d'éviter des erreurs courantes, comme les conflits de versions ou les dépendances manquantes. Parmi ces pratiques, le **versioning sémantique** (SemVer) est largement adopté, mais son application reste inégale, notamment dans les projets open source où les contributeurs peuvent ignorer les règles de compatibilité ascendante. Par exemple, lors des tests sur *TelegramBots*, un conflit entre deux versions majeures d'une même dépendance a provoqué un échec de build, alors qu'une analyse statique aurait pu le détecter en amont. Pour atténuer ce risque, des outils comme **Maven Enforcer Plugin** permettent de définir des règles strictes pour les versions de dépendances, mais leur configuration est souvent perçue comme complexe et est donc sous-utilisée.

Un autre défi est la résolution des dépendances transitives, qui peuvent introduire des vulnérabilités ou des incompatibilités non détectées par les outils statiques. Des solutions comme **Maven Dependency Tree** ou **Gradle's Dependency Insight** offrent une visibilité sur l'arbre des dépendances, mais leur utilisation reste réactive plutôt que proactive. Par exemple, lors des tests sur *manimgl*, l'absence de *libpango1.0-dev* n'a pas été détectée avant l'exécution, car cette dépendance système n'était pas référencée dans le *pom.xml*. Une bonne pratique consiste à documenter explicitement les dépendances système dans un fichier *README* ou un script d'installation, mais cette approche repose sur la discipline des contributeurs et n'est pas automatisable.

La gestion des droits d'exécution, en particulier pour les scripts comme *mvnw*, est un autre point critique souvent négligé. Lors des tests sur *TelegramBots*, l'échec initial était dû à des permissions insuffisantes sur *mvnw*, un problème résolu par l'ajout d'une commande *chmod +x* dans le pipeline. Cependant, cette solution ad hoc n'est pas scalable, car elle suppose que tous les projets utilisent le même mécanisme d'exécution. Une approche plus robuste consiste à encapsuler les builds dans des conteneurs Docker, où les permissions sont gérées de manière isolée. Cette méthode a été testée avec succès sur *spring-boot-boilerplate* et *java-spring-boot-boilerplate*, mais elle introduit une complexité supplémentaire pour les projets hybrides comme *manimgl*, où les dépendances système doivent être installées dans le conteneur.

Enfin, la gestion des dépendances dans des projets multi-modules comme *opengrok* nécessite une approche spécifique, où chaque module doit être analysé individuellement avant d'agréger les résultats. Les outils actuels, comme le Maven Dependency Plugin, ne prennent pas en charge cette granularité de manière native, ce qui oblige les équipes à développer des scripts personnalisés. Cette fragmentation des outils et des méthodes explique en partie pourquoi les tests sur *opengrok* ont nécessité cinq tentatives avant d'échouer définitivement : l'absence de planification explicite et la variabilité des approches ont conduit à des solutions partielles, sans jamais adresser le problème de fond lié à la structure multi-modules.

Les limites des solutions actuelles : vers une approche proactive et unifiée

Les tests réalisés sur des projets comme *opengrok*, *manimgl* ou *TelegramBots* ont mis en lumière les lacunes des outils actuels, qui peinent à adresser des scénarios complexes où les dépendances, les permissions et les structures de projet interagissent de manière non linéaire. L'une des limites majeures est l'**absence de planification explicite** dans les outils d'analyse, qui adoptent une approche réactive plutôt que proactive. Par exemple, lors des tests sur *manimgl*, l'agent a tenté des solutions aléatoires (comme l'installation de *pip install manimgl*) sans consulter la documentation officielle, ce qui a conduit à un bouclage sur des erreurs évitables. Cette observation suggère que les outils actuels manquent d'un mécanisme de *reasoning* capable de hiérarchiser les actions en fonction du contexte.

Une autre limite est la **fragmentation des outils**, qui oblige les équipes à développer des scripts ad hoc pour corréler les résultats de différents scans. Par exemple, les alertes générées par OWASP Dependency-Check ne sont pas directement exploitables par SonarQube, ce qui complique la priorisation des correctifs. Cette fragmentation est particulièrement problématique dans des pipelines CI/CD, où le temps d'exécution est un facteur critique. Les tests ont montré que le temps moyen d'analyse pour un projet Maven était de 12 minutes, une durée qui peut devenir prohibitive pour des projets avec des dizaines de modules. Des optimisations, comme le parallélisme des scans ou la mise en cache des résultats, sont possibles, mais elles nécessitent une intégration fine avec les outils existants, ce qui n'est pas toujours supporté.

Enfin, les outils actuels sont **peu adaptés aux projets hybrides**, combinant dépendances Maven et dépendances système. Le cas de *manimgl* est emblématique : l'absence de *libpango1.0-dev* n'a pas été détectée par les outils statiques, car ceux-ci ne scannent que les fichiers *pom.xml*. Une solution potentielle consisterait à intégrer des outils comme **Ansible** ou **Chef** pour gérer les dépendances système, mais cette approche introduit une complexité supplémentaire et n'est pas standardisée. De plus, les outils de scan dynamique,

comme les frameworks de test d'intégration, ne sont pas conçus pour détecter ce type de dépendances, ce qui crée un angle mort dans l'analyse.

Pour surmonter ces limites, une piste prometteuse est l'adoption d'une **architecture multi-agent**, où un *manager* centralisé orchestrerait les actions des différents outils en fonction d'un plan préétabli. Cette approche, inspirée des systèmes de *planning* en intelligence artificielle, permettrait de séparer la phase de diagnostic (identification des problèmes) de la phase d'exécution (application des correctifs). Par exemple, dans le cas d'*opengrok*, un agent pourrait d'abord analyser la structure multi-modules, identifier les modules critiques, puis appliquer des correctifs ciblés, plutôt que de modifier aveuglément le *pom.xml*. Cette séparation des responsabilités réduirait la variabilité des approches et améliorerait la robustesse des pipelines. Cependant, sa mise en œuvre nécessite des avancées significatives dans l'interopérabilité des outils et la standardisation des formats de données, des défis qui restent ouverts à ce jour.

Analyse des échecs et diagnostic des problèmes récurrents

L'analyse des échecs rencontrés au cours de ce stage révèle une complexité sous-jacente dans l'automatisation des builds pour des projets logiciels hétérogènes, notamment lorsque ceux-ci s'appuient sur des architectures multi-modules, des dépendances système externes ou des configurations de permissions non triviales. Cette section propose une dissection méthodique des problèmes récurrents, en s'appuyant sur une approche à la fois quantitative et qualitative pour en dégager les causes profondes, les impacts mesurables et les pistes d'amélioration. L'étude s'articule autour de trois axes principaux : la classification des échecs par nature, l'analyse des logs comme outil de diagnostic, et l'établissement de corrélations entre la complexité des projets et leur propension à l'échec.

Classification des échecs par nature et quantification de leur impact

Les échecs observés lors des phases de test se répartissent en trois catégories distinctes, chacune présentant des caractéristiques et des défis spécifiques. La première, et la plus fréquente dans les premières itérations, concerne les problèmes de permissions, qui ont représenté jusqu'à 20 % des échecs initiaux. Le cas emblématique de ce type d'erreur est survenu lors du build du projet *TelegramBots*, où l'exécution du script `mvnw` a systématiquement échoué en raison de droits d'exécution insuffisants. Cette situation illustre une problématique récurrente dans les environnements automatisés, où les scripts shell inclus dans les dépôts Git ne sont pas toujours configurés avec les permissions adéquates pour une exécution en contexte conteneurisé. L'analyse des logs a révélé que l'erreur `Permission denied` était immédiatement suivie de tentatives infructueuses de contournement, telles que l'invocation directe de `mvn` sans passer par le wrapper, ou des modifications inutiles du fichier `pom.xml`. Ces actions, bien que logiques en apparence, témoignent d'une absence de planification préalable et d'une méconnaissance des bonnes pratiques Maven, où le wrapper `mvnw` est précisément conçu pour garantir une exécution cohérente indépendamment de l'environnement. La solution retenue, consistant à appliquer systématiquement un `chmod +x mvnw` avant toute tentative de build, a permis d'éradiquer ce type d'échec, mais elle soulève une question plus large sur la nécessité d'intégrer des vérifications préventives dans le workflow d'automatisation, plutôt que de réagir a posteriori aux erreurs.

La deuxième catégorie d'échecs, liée aux dépendances, s'est avérée particulièrement problématique pour les projets multi-modules et ceux nécessitant des bibliothèques système externes. Le projet *opengrok*, par exemple, a nécessité cinq tentatives avant d'aboutir à un build réussi, en raison de sa structure complexe composée de plusieurs sous-modules interdépendants. Les logs ont montré que l'agent automatisé tentait initialement d'exécuter un build global via `mvn clean install`, une approche qui échoue systématiquement lorsque les modules ne sont pas correctement configurés pour une compilation séquentielle. Cette erreur révèle une méconnaissance des spécificités des projets Maven multi-modules, où chaque module doit souvent être compilé individuellement avant d'être intégré dans un build global. La solution temporaire, consistant à scanner et builder chaque module séparément, a permis de résoudre le problème, mais elle n'est pas scalable pour des projets comportant des dizaines de modules. Par ailleurs, le projet *manimgl* a mis en lumière une autre facette des problèmes de dépendances, à savoir l'absence de détection des prérequis système. Dans ce cas, l'agent a correctement identifié la nécessité d'installer le package Python `manimgl`, mais il a omis de vérifier les dépendances système sous-jacentes, telles que `libpango1.0-dev`, pourtant documentées dans les fichiers de configuration du projet. Cette omission a conduit à des boucles d'erreurs répétitives, où l'agent tentait des solutions aléatoires sans consulter les ressources disponibles, comme la documentation officielle ou les fichiers `README`. Ces échecs soulignent l'importance d'une phase de planification explicite, où les dépendances système et les structures de projet sont analysées avant toute tentative de build.

Enfin, les échecs liés à la planification constituent une catégorie plus subtile, mais tout aussi critique. Ils se manifestent lorsque l'agent s'écarte de la solution optimale pour s'engager dans des modifications inutiles ou contre-productives. Un exemple frappant est survenu lors du build d'*opengrok*, où l'agent a tenté de modifier le fichier `pom.xml` en ajoutant des dépendances ou en ajustant des versions, alors que le problème initial était lié à la structure multi-modules. Ces actions, bien que motivées par une volonté de résoudre l'erreur, ont en réalité complexifié le diagnostic en introduisant des variables supplémentaires. L'analyse des logs a révélé que ces écarts étaient souvent précédés d'une accumulation d'erreurs dans l'historique, suggérant que l'agent, submergé par des informations contradictoires, perdait de vue l'objectif initial. Cette observation a conduit à l'hypothèse selon laquelle une séparation claire entre la phase de planification et la phase d'exécution pourrait améliorer significativement la robustesse du processus. En effet, une planification préalable permettrait d'identifier les étapes critiques, les dépendances à vérifier et les outils à utiliser, réduisant ainsi le risque de s'engager dans des voies sans issue.

Méthodologie d’analyse des logs et interprétation des erreurs récurrentes

L’analyse des logs s’est imposée comme un outil indispensable pour diagnostiquer les échecs et comprendre les mécanismes sous-jacents aux erreurs observées. Contrairement à une approche intuitive, où les erreurs seraient traitées de manière isolée, l’examen systématique des logs permet de reconstituer la séquence d’actions menant à l’échec, d’identifier les patterns récurrents et d’évaluer l’efficacité des solutions proposées. Dans le cadre de ce stage, une méthodologie rigoureuse a été mise en place pour extraire et interpréter ces logs, en s’appuyant sur trois étapes clés : la collecte structurée des données, l’identification des motifs d’erreur et la corrélation avec les actions de l’agent.

La collecte des logs a été standardisée pour garantir une comparabilité entre les différents projets et les tentatives de build. Chaque exécution a généré un fichier de log complet, incluant les commandes exécutées, les sorties standard et d’erreur, ainsi que les timestamps pour chaque étape. Cette granularité temporelle s’est avérée cruciale pour distinguer les erreurs transitoires des problèmes persistants. Par exemple, dans le cas de *TelegramBots*, les logs ont révélé que l’erreur `Permission denied` survenait systématiquement dans les premières secondes de l’exécution, avant même que l’agent n’ait tenté une quelconque solution. Cette observation a permis de cibler immédiatement le problème de permissions, plutôt que de s’engager dans des diagnostics plus complexes. À l’inverse, pour *opengrok*, les logs ont montré une accumulation progressive d’erreurs, avec des tentatives de build global suivies de modifications du `pom.xml`, puis de nouvelles tentatives infructueuses. Cette séquence a mis en lumière un phénomène de boucle infinie, où l’agent, incapable de résoudre le problème initial, s’enfermait dans des actions répétitives et inefficaces.

L’identification des motifs d’erreur a reposé sur une analyse qualitative des logs, complétée par des outils de traitement automatique pour détecter les patterns récurrents. Parmi les motifs les plus fréquents, on retrouve les boucles d’erreurs, où l’agent répète les mêmes actions sans adaptation, et les tentatives aléatoires, où des solutions non pertinentes sont essayées sans logique apparente. Les boucles d’erreurs sont particulièrement problématiques, car elles consomment des ressources sans apporter de progrès. Dans le cas de *manimgl*, par exemple, l’agent a tenté à plusieurs reprises d’installer `manimgl` via `pip`, alors que l’erreur persistante indiquait clairement un problème de dépendances système. Cette incapacité à sortir de la boucle suggère un défaut de mémoire à court terme, où l’agent ne conserve pas une trace claire des actions déjà tentées et de leurs résultats. Les tentatives aléatoires, quant à elles, se manifestent par des modifications non justifiées des fichiers de configuration ou des commandes exécutées sans lien avec l’erreur initiale. Ces comportements révèlent une absence de planification et une tendance à l’improvisation, qui peut s’avérer contre-productive dans des environnements complexes.

La corrélation entre les logs et les actions de l'agent a permis de dégager des enseignements clés pour l'amélioration du processus. Par exemple, l'analyse des logs de *opengrok* a montré que les modifications du `pom.xml` étaient systématiquement précédées d'une accumulation d'erreurs liées aux modules. Cette observation a conduit à l'hypothèse selon laquelle l'agent, face à une erreur complexe, tente des solutions de plus en plus invasives, sans évaluer leur pertinence. Pour remédier à ce problème, une approche basée sur une hiérarchie des solutions a été envisagée, où les actions les plus simples et les moins risquées (comme la vérification des permissions ou des dépendances) sont tentées en premier, avant d'envisager des modifications plus profondes. Par ailleurs, l'analyse des logs a révélé que les échecs étaient souvent précédés d'une absence de vérification des prérequis, comme la présence de dépendances système ou la structure du projet. Cette constatation a motivé l'introduction d'une phase de pré-analyse, où les caractéristiques du projet sont évaluées avant toute tentative de build, afin d'orienter l'agent vers les solutions les plus adaptées.

Métriques de performance et facteurs d'amélioration

L'évaluation des performances du processus d'automatisation s'est appuyée sur une série de métriques quantitatives, permettant de mesurer l'évolution du taux de réussite et d'identifier les facteurs ayant contribué à son amélioration. Les résultats initiaux, obtenus lors des premiers tests sur cinq projets Maven, ont révélé un taux de réussite de 60 %, avec des échecs concentrés sur *TelegramBots* et *opengrok*. Cette performance, bien que modeste, a servi de référence pour évaluer l'impact des corrections apportées. Après l'introduction de solutions ciblées, telles que la vérification systématique des permissions et le scan individuel des modules, le taux de réussite a progressé à 80 %, avec un succès sur quatre des cinq projets initiaux. Cette amélioration, bien que significative, a mis en lumière la persistance de défis pour les projets les plus complexes, comme *opengrok*, dont le build n'a abouti qu'après cinq tentatives et des ajustements manuels.

Les tests ultérieurs, menés sur un échantillon élargi de trente projets, ont permis d'atteindre un taux de réussite de 90 %, avec seulement trois échecs sur des projets qualifiés d'atypiques. Cette progression s'explique par plusieurs facteurs clés, dont l'introduction de vérifications préventives et l'optimisation des stratégies de résolution. Par exemple, l'ajout d'une étape de pré-analyse pour détecter les dépendances système manquantes a permis de réduire les échecs liés à des bibliothèques externes, comme dans le cas de *manimgl*. De même, la séparation des phases de planification et d'exécution a contribué à limiter les modifications inutiles des fichiers de configuration, en garantissant que chaque action est précédée d'une évaluation de sa pertinence. Ces optimisations ont non seulement amélioré le taux de réussite, mais elles ont également réduit le temps moyen d'exécution, passant de 15 minutes pour les premiers tests à 12 minutes pour les projets Maven, et de 7 à 5 minutes pour les projets Python.

L’analyse des métriques a également révélé une corrélation forte entre la complexité des projets et leur propension à l’échec. Les projets comportant plus de cinq modules ou nécessitant des dépendances système externes ont présenté un taux d’échec trois fois supérieur à celui des projets plus simples. Cette observation souligne l’importance d’adapter le processus d’automatisation à la complexité du projet, en intégrant des mécanismes de détection précoce des défis potentiels. Par exemple, les projets multi-modules pourraient bénéficier d’une phase de pré-build, où chaque module est analysé individuellement avant d’être intégré dans un build global. De même, les projets nécessitant des dépendances système pourraient être identifiés dès la phase de pré-analyse, afin de déclencher des vérifications spécifiques avant toute tentative d’installation. Ces adaptations, bien que nécessitant des développements supplémentaires, permettraient de réduire encore le taux d’échec et d’améliorer la robustesse du processus.

Enfin, l’évaluation de la qualité des rapports générés a fourni un indicateur complémentaire de la performance du système. Avec un score moyen de 85 sur 100, les rapports ont été jugés complets et cohérents, grâce à l’introduction de vérifications automatiques pour valider la présence de toutes les sections requises et la cohérence des données. Cette métrique, bien que secondaire par rapport au taux de réussite, est cruciale pour garantir l’utilisabilité des résultats et la confiance des utilisateurs dans le processus d’automatisation. Les améliorations futures pourraient inclure une analyse plus fine des échecs résiduels, notamment pour les projets atypiques, afin d’identifier des patterns spécifiques et d’adapter le workflow en conséquence. Par ailleurs, l’introduction d’un système de feedback continu, où les échecs sont analysés et intégrés dans une base de connaissances, pourrait permettre d’améliorer progressivement la performance du système, en capitalisant sur les enseignements tirés de chaque exécution.

Conception de l'architecture technique du workflow d'analyse

La conception de l’architecture technique du workflow d’analyse a constitué une étape charnière dans l’évolution du projet, marquée par une transition progressive d’une approche linéaire et réactive vers une structure modulaire, scalable et intelligente. Cette refonte a été motivée par les limites intrinsèques du modèle initial, dont les performances se sont révélées insuffisantes face à la complexité croissante des projets Maven et Python analysés. L’objectif sous-jacent était double : d’une part, améliorer la robustesse du système en réduisant les échecs liés à des erreurs contextuelles ou à des dépendances mal gérées, et d’autre part, optimiser l’efficacité globale en introduisant des mécanismes de planification et de validation systématiques. Cette section détaille les choix architecturaux, les justifications techniques, ainsi que les compromis adoptés pour répondre à ces enjeux.

Des limites du modèle initial à la nécessité d’une refonte architecturale

Le modèle initial, fondé sur une boucle simple de détection-correction-réessai, reposait sur une logique itérative où chaque erreur identifiée déclenchait une tentative de résolution immédiate, suivie d’un nouveau cycle d’analyse. Cette approche, bien que fonctionnelle pour des cas d’usage triviaux, s’est rapidement heurtée à des obstacles majeurs lors des tests sur des projets réels. Les résultats obtenus sur les cinq projets Maven sélectionnés – *spring-boot-boilerplate*, *java-spring-boot-boilerplate*, *BankingPortal-API*, *TelegramBots* et *opengrok* – ont révélé un taux de réussite de seulement 60 % lors des premières exécutions, avec des échecs particulièrement marqués sur les projets multi-modules ou nécessitant des permissions spécifiques.

L’analyse des logs a mis en lumière un problème fondamental : l’absence de contextualisation des erreurs. Le système, confronté à une erreur de dépendance ou à un problème de droits d’exécution, tentait des corrections de manière aléatoire, sans évaluer au préalable la nature exacte du problème ni les dépendances potentielles entre les composants. Par exemple, dans le cas du projet *opengrok*, le système a persisté à modifier le fichier *pom.xml* sans tenir compte de la structure multi-modules du projet, ce qui a conduit à des boucles infinies de tentatives infructueuses. De même, pour *TelegramBots*, l’échec initial était lié à un simple problème de permissions sur le script *mvnw*, mais l’absence de vérification préalable des droits d’exécution a retardé la résolution.

Ces observations ont conduit à une remise en question profonde de l'architecture. Une boucle linéaire, même enrichie de mécanismes de rétroaction, ne permettait pas de gérer la complexité inhérente aux projets modernes, où les erreurs sont souvent interdépendantes et nécessitent une approche méthodique. La variabilité des temps d'exécution, allant de 4 minutes pour *java-spring-boot-boilerplate* à plus de 15 minutes pour *BankingPortal-API*, a également souligné la nécessité d'optimiser la gestion des ressources, notamment en environnement conteneurisé. Ces constats ont servi de catalyseur pour repenser l'architecture en profondeur, en introduisant des phases distinctes de planification, d'exécution et de validation, ainsi qu'une modularité accrue pour isoler les responsabilités.

Introduction d'une architecture modulaire : planification, exécution et validation

La refonte architecturale s'est articulée autour de trois phases clés, chacune encapsulée dans un module indépendant : la planification, l'exécution et la validation. Cette décomposition a été guidée par le principe de séparation des préoccupations, un paradigme fondamental en génie logiciel visant à isoler les responsabilités pour améliorer la maintenabilité, la testabilité et l'évolutivité du système. L'hypothèse sous-jacente était qu'une approche structurée, où chaque phase est optimisée pour une tâche spécifique, permettrait de réduire les échecs liés à des décisions prises dans l'urgence ou sans contexte suffisant.

La phase de planification : vers une approche méthodique

La phase de planification a été conçue comme un prérequis à toute action corrective, avec pour objectif de générer un plan d'action structuré avant toute tentative de résolution. Ce module repose sur une analyse préliminaire du projet, incluant la détection de sa structure (mono-module ou multi-modules), l'identification des dépendances déclarées dans les fichiers de configuration (*pom.xml* pour Maven, *requirements.txt* ou *pyproject.toml* pour Python), ainsi que la vérification des permissions et des prérequis système. Par exemple, pour un projet Maven multi-modules comme *opengrok*, le planificateur génère une checklist dynamique indiquant les étapes à suivre : scanner chaque module individuellement, vérifier la cohérence des versions de dépendances entre modules, et s'assurer que les outils de build (*mvnw* ou *mvn*) sont exécutables.

L'introduction de ce module a été motivée par les échecs observés lors des tests initiaux, où le système, en l'absence de planification, adoptait des stratégies de résolution inefficaces. Dans le cas de *manimgl*, par exemple, l'agent initial a tenté d'installer la dépendance Python via *pip* sans vérifier au préalable les dépendances système requises (*libpango1.0-dev*), ce qui a conduit à des erreurs persistantes. Le planificateur, en revanche, intègre désormais

une étape de consultation de la documentation officielle du projet (via des requêtes ciblées ou des fichiers *README*) pour identifier les prérequis non déclarés dans les fichiers de configuration. Cette approche a permis de réduire significativement les boucles d'erreurs, en évitant les tentatives de résolution basées sur des hypothèses erronées.

Un autre avantage majeur de la planification réside dans sa capacité à prioriser les actions en fonction de leur criticité. Par exemple, un problème de permissions sur un script *mvnw* est identifié comme bloquant et traité en priorité, tandis qu'une dépendance optionnelle manquante peut être reportée à une étape ultérieure. Cette granularité dans la planification a permis d'améliorer le taux de réussite global, comme en témoignent les résultats obtenus après refonte : 80 % de succès sur les cinq projets Maven initiaux, puis 90 % sur un échantillon élargi de trente projets.

La phase d'exécution : modularité et spécialisation des tâches

La phase d'exécution a été repensée pour refléter la complexité des tâches à accomplir, en décomposant le processus en sous-modules spécialisés, chacun responsable d'un aspect spécifique du workflow. Cette approche modulaire présente plusieurs avantages. D'abord, elle permet d'isoler les sources d'erreurs : un échec dans la résolution des dépendances, par exemple, n'impacte pas directement la gestion des permissions ou le processus de build. Ensuite, elle facilite l'évolution du système, en permettant d'ajouter ou de modifier des sous-modules sans remettre en cause l'ensemble de l'architecture. Enfin, elle offre une meilleure traçabilité, chaque sous-module générant des logs détaillés qui peuvent être analysés indépendamment.

Parmi les sous-modules implémentés, on peut citer : - **Gestion des permissions** : ce module vérifie et corrige les droits d'exécution sur les scripts critiques (*mvnw*, *setup.py*, etc.), en appliquant des commandes comme *chmod +x* lorsque nécessaire. Son introduction a permis de résoudre systématiquement les échecs liés à *TelegramBots*, où le problème initial était purement lié à une permission manquante. - **Résolution des dépendances** : ce sous-module analyse les fichiers de configuration pour identifier les dépendances manquantes ou obsolètes, puis tente de les installer via les gestionnaires de paquets appropriés (*mvn* pour Maven, *pip* ou *conda* pour Python). Pour les dépendances système, comme dans le cas de *manimgl*, il consulte une base de connaissances interne ou la documentation officielle pour déterminer les paquets à installer via *apt* ou *yum*. - **Build et compilation** : ce module exécute les commandes de build (*mvn clean install*, *python setup.py install*, etc.) en tenant compte des spécificités du projet. Pour les projets multi-modules, il applique une stratégie de build séquentielle, en commençant par les modules racines avant de traiter les sous-modules dépendants. - **Gestion des conteneurs** : ce sous-module supervise le cycle de vie des conteneurs Docker utilisés pour l'analyse, en veillant à leur nettoyage après chaque exécution pour éviter la pollution des environnements. Il optimise également l'allocation des ressources, en ajustant

dynamiquement la mémoire allouée en fonction de la taille du projet (une nécessité pour *BankingPortal-API*, dont le build consomme des ressources importantes).

L'un des défis majeurs de cette phase a été la coordination entre les sous-modules, notamment pour gérer les dépendances entre les tâches. Par exemple, la résolution des dépendances doit impérativement précéder le build, tandis que la gestion des permissions peut être effectuée en parallèle. Pour répondre à ce besoin, un orchestrateur léger a été introduit, chargé de définir l'ordre d'exécution des sous-modules en fonction des dépendances identifiées lors de la phase de planification. Cet orchestrateur utilise un graphe de dépendances pour déterminer les séquences optimales, évitant ainsi les blocages liés à des tâches mal ordonnées.

La phase de validation : garantie de qualité et robustesse

La phase de validation constitue le dernier rempart avant la génération du rapport final, avec pour mission de s'assurer que les corrections apportées sont cohérentes, complètes et conformes aux attentes. Cette phase repose sur deux piliers : des vérifications automatiques et des tests unitaires, chacun jouant un rôle complémentaire dans la garantie de la qualité des résultats.

Les vérifications automatiques couvrent plusieurs aspects critiques du rapport généré. D'abord, une vérification de la cohérence structurelle s'assure que toutes les sections attendues (dépendances, permissions, résultats de build, recommandations) sont présentes et correctement formatées. Ensuite, une vérification de la cohérence sémantique valide la logique interne du rapport, par exemple en s'assurant que les totaux calculés (nombre de dépendances manquantes, temps d'exécution) sont corrects et que les recommandations proposées sont alignées avec les erreurs identifiées. Enfin, une vérification du format vérifie que le rapport respecte les standards Markdown, avec une attention particulière portée aux liens, aux tableaux et aux blocs de code.

Pour quantifier la qualité globale du rapport, un score sur 100 a été introduit, calculé à partir de critères pondérés tels que la complétude des sections (40 %), la cohérence des données (30 %), la pertinence des recommandations (20 %) et la conformité au format (10 %). Ce score, bien que subjectif dans sa pondération, offre une métrique objective pour évaluer les améliorations apportées au fil des itérations. Lors des tests finaux sur trente projets, le score moyen obtenu était de 85/100, avec des pics à 95/100 pour les projets bien documentés et des scores plus faibles (70-75/100) pour les projets atypiques ou mal structurés.

En complément des vérifications automatiques, des tests unitaires ont été implémentés pour valider le bon fonctionnement des scripts critiques, notamment ceux responsables de la résolution des dépendances et de la gestion des permissions. Ces tests, exécutés dans des environnements isolés, simulent des scénarios d'échec courants (dépendances manquantes, permissions insuffisantes) et vérifient que les corrections apportées sont

conformes aux attentes. Par exemple, un test unitaire pour le module de gestion des permissions simule un script *mvnw* sans droits d'exécution et vérifie que la commande *chmod +x* est bien appliquée. Ces tests, bien que chronophages à mettre en place, ont permis de réduire significativement les régressions lors des évolutions du système.

Intégration des conteneurs Docker : isolation, optimisation et nettoyage

L'utilisation de conteneurs Docker pour exécuter les analyses a constitué un choix architectural majeur, motivé par la nécessité d'isoler les environnements d'exécution et de garantir la reproductibilité des résultats. Cependant, cette approche a également introduit des défis spécifiques, notamment en termes de gestion des ressources et de nettoyage post-exécution, qui ont nécessité des optimisations ciblées.

Stratégie de nettoyage et gestion des ressources

L'un des risques majeurs liés à l'utilisation de conteneurs est la pollution des environnements, où des artefacts résiduels (fichiers temporaires, dépendances installées, caches de build) peuvent fausser les analyses ultérieures ou saturer les ressources disponibles. Pour atténuer ce risque, une stratégie de nettoyage systématique a été mise en place, articulée autour de trois axes : 1. **Nettoyage post-exécution** : chaque conteneur est détruit immédiatement après l'analyse, en veillant à supprimer tous les volumes et réseaux associés. Cette approche, bien que radicale, garantit que chaque analyse commence dans un environnement vierge, éliminant ainsi les risques de contamination croisée entre projets. 2. **Gestion des caches** : pour les projets Maven, les caches locaux (*~/.m2/repository*) sont montés dans des volumes éphémères, qui sont supprimés après chaque exécution. Cette mesure évite l'accumulation de dépendances obsolètes ou corrompues, tout en accélérant les builds ultérieurs pour des projets similaires. 3. **Optimisation des ressources** : pour les projets volumineux comme *BankingPortal-API*, une allocation dynamique de la mémoire a été implémentée, en fonction de la taille du projet et de la complexité du build. Cette optimisation repose sur une analyse préliminaire du fichier *pom.xml* pour estimer les besoins en mémoire, suivie d'un ajustement des paramètres Docker (*--memory*, *--cpus*) avant le lancement du conteneur.

Ces mesures ont permis de réduire significativement les échecs liés à des problèmes de ressources, tout en maintenant des temps d'exécution raisonnables. Par exemple, le temps moyen d'analyse pour les projets Maven est passé de 15 minutes à 12 minutes après optimisation, avec une variabilité réduite entre les exécutions.

Gestion des dépendances système et des prérequis

Un autre défi lié à l'utilisation de conteneurs concerne la gestion des dépendances système, qui ne sont pas toujours déclarées dans les fichiers de configuration des projets. Par exemple, le projet *manimgl* nécessite l'installation de *libpango1.0-dev*, une dépendance système non mentionnée dans *requirements.txt*. Pour résoudre ce problème, une base de connaissances interne a été constituée, associant des projets ou des bibliothèques à leurs dépendances système courantes. Cette base est consultée lors de la phase de planification, et les dépendances identifiées sont installées via *apt* ou *yum* avant le lancement des commandes de build.

Dans les cas où la base de connaissances ne contient pas d'information pertinente, le système consulte la documentation officielle du projet, en analysant les fichiers *README* ou les pages de documentation en ligne pour identifier les prérequis. Cette approche, bien que plus lente, a permis de résoudre des cas complexes comme celui de *manimgl*, où l'absence de dépendance système bloquait systématiquement le build.

Vers une architecture multi-agent : hypothèses et perspectives

Les limites persistantes du système modulaire, notamment face à des projets très atypiques ou mal documentés, ont conduit à explorer une piste plus ambitieuse : une architecture multi-agent, où chaque agent serait spécialisé dans une tâche spécifique et coordonné par un agent "Manager". Cette hypothèse, bien que non implémentée durant le stage, a fait l'objet d'une étude approfondie et d'une modélisation préliminaire, dont les résultats sont présentés ci-dessous.

Principes de l'architecture multi-agent

L'architecture multi-agent repose sur une division des responsabilités entre plusieurs agents autonomes, chacun doté d'une expertise spécifique et capable de communiquer avec les autres pour atteindre un objectif commun. Dans le contexte du workflow d'analyse, cette approche permettrait de décomposer le processus en tâches hautement spécialisées, tout en introduisant une couche de coordination pour gérer les dépendances entre les agents.

Les rôles envisagés pour les agents incluent : - **Agent Manager** : responsable de la planification globale et de la coordination entre les agents. Cet agent génère un plan d'action initial, le met à jour en fonction des retours des autres agents, et prend les décisions critiques en cas de blocage. - **Agent Dépendances** : spécialisé dans l'identification et la résolution des dépendances manquantes, qu'elles soient déclarées dans les fichiers de

configuration ou requises au niveau système. - **Agent Permissions** : chargé de vérifier et de corriger les droits d'exécution sur les scripts et binaires critiques. - **Agent Build** : responsable de l'exécution des commandes de build et de la collecte des logs. - **Agent Documentation** : dédié à la consultation de la documentation officielle des projets pour identifier des solutions non évidentes (comme les dépendances système manquantes). - **Agent Validation** : en charge des vérifications automatiques et du calcul du score de qualité du rapport.

Cette décomposition présente plusieurs avantages théoriques. D'abord, elle permet une parallélisation accrue des tâches, où plusieurs agents peuvent travailler simultanément sur des aspects indépendants du projet (par exemple, l'agent Permissions et l'agent Dépendances). Ensuite, elle améliore la résilience du système, car un échec d'un agent n'impacte pas nécessairement les autres. Enfin, elle offre une meilleure évolutivité, en permettant d'ajouter de nouveaux agents sans modifier l'architecture globale.

Diagramme de séquence et interactions entre agents

Pour illustrer le fonctionnement de cette architecture, un diagramme de séquence a été modélisé, décrivant les interactions entre les agents lors de la détection et de la résolution d'une dépendance manquante. Prenons l'exemple d'un projet où l'agent Build échoue en raison d'une dépendance non résolue :

1. **Détection de l'erreur** : l'agent Build exécute la commande *mvn clean install* et détecte une erreur liée à une dépendance manquante (*org.springframework:spring-core:5.3.0*).
2. **Notification au Manager** : l'agent Build notifie l'agent Manager de l'échec, en transmettant le log d'erreur.
3. **Planification de la résolution** : l'agent Manager consulte l'agent Dépendances pour déterminer si la dépendance est déclarée dans le *pom.xml*. Dans ce cas, la dépendance est présente mais n'a pas pu être téléchargée.
4. **Recherche de solutions** : l'agent Manager sollicite l'agent Documentation pour consulter la documentation officielle de *spring-core* et identifier des solutions potentielles (par exemple, un miroir Maven alternatif ou une version compatible).
5. **Tentative de résolution** : l'agent Dépendances tente d'installer la dépendance en utilisant les informations fournies par l'agent Documentation.
6. **Validation** : l'agent Build réexécute la commande de build et notifie l'agent Manager du succès ou de l'échec. En cas de succès, l'agent Validation génère un rapport et calcule le score de qualité.

Ce scénario illustre la richesse des interactions possibles dans une architecture multi-agent, où chaque agent contribue à la résolution du problème en apportant son expertise spécifique. Par rapport à l'architecture modulaire actuelle, cette approche introduirait une flexibilité accrue, en permettant aux agents de s'adapter dynamiquement aux situations imprévues (par exemple, en consultant la documentation en temps réel pour résoudre une erreur complexe).

Défis et limites de l'approche multi-agent

Bien que prometteuse, l'architecture multi-agent soulève plusieurs défis techniques et conceptuels. D'abord, la coordination entre les agents nécessite un protocole de communication robuste, capable de gérer les dépendances entre les tâches et d'éviter les blocages. Par exemple, si deux agents tentent de modifier simultanément le même fichier (*pom.xml*), des conflits peuvent survenir. Pour résoudre ce problème, une solution envisagée consiste à introduire un mécanisme de verrouillage, où l'agent Manager attribue des jetons d'accès exclusifs aux agents pour les ressources critiques.

Ensuite, la complexité de l'architecture multi-agent peut entraîner une augmentation des temps d'exécution, en raison des overheads liés à la communication entre agents. Pour atténuer cet effet, des optimisations sont possibles, comme la mise en cache des résultats des agents (par exemple, l'agent Documentation pourrait stocker les solutions trouvées pour des erreurs récurrentes).

Enfin, la mise en œuvre d'une telle architecture nécessite des outils adaptés, comme des frameworks de développement multi-agent (par exemple, JADE pour Java ou SPADE pour Python). Ces outils, bien que puissants, introduisent une courbe d'apprentissage et une complexité supplémentaire, qui doivent être pesées face aux bénéfices attendus.

Bilan et perspectives d'évolution

La refonte architecturale du workflow d'analyse a permis de franchir un cap significatif en termes de robustesse et d'efficacité. Le passage d'une boucle linéaire à une architecture modulaire, articulée autour des phases de planification, d'exécution et de validation, a permis d'améliorer le taux de réussite de 60 % à 90 % sur un échantillon élargi de projets. Cette progression a été rendue possible par une approche méthodique, où chaque module a été conçu pour répondre à des besoins spécifiques, tout en étant intégré dans un ensemble cohérent.

Les optimisations apportées à la gestion des conteneurs Docker ont également joué un rôle clé, en garantissant des environnements d'exécution isolés et reproductibles, tout en

optimisant l’utilisation des ressources. La stratégie de nettoyage systématique et l’allocation dynamique de la mémoire ont permis de réduire les échecs liés à des problèmes de ressources, tout en maintenant des temps d’exécution acceptables.

Enfin, l’exploration d’une architecture multi-agent a ouvert des perspectives prometteuses pour les évolutions futures du système. Bien que non implémentée durant le stage, cette approche offre un cadre théorique solide pour améliorer encore la flexibilité et la résilience du workflow, notamment face à des projets complexes ou mal documentés. Les défis identifiés (coordination entre agents, overheads de communication) devront être adressés dans le cadre de travaux ultérieurs, en s’appuyant sur des outils et des frameworks adaptés.

À plus court terme, les améliorations prioritaires pourraient inclure l’enrichissement de la base de connaissances pour la résolution des dépendances système, ainsi que l’introduction de mécanismes d’apprentissage automatique pour affiner les stratégies de résolution en fonction des échecs passés. Ces évolutions, combinées à une architecture déjà robuste, permettraient de tendre vers un système capable de s’adapter dynamiquement à une grande variété de projets, tout en garantissant des résultats fiables et reproductibles.

Implémentation détaillée des solutions techniques

L’implémentation des solutions techniques a constitué une phase critique du stage, marquée par une approche systématique visant à résoudre les problèmes identifiés tout en optimisant les performances et la robustesse du système. Cette section détaille les choix architecturaux, les algorithmes développés, et les mécanismes de validation mis en place, en insistant sur leur justification théorique et leur impact pratique. Les développements ont été guidés par une analyse rigoureuse des échecs initiaux, révélant des lacunes dans la gestion des permissions, la détection des structures de projet complexes, et la cohérence des rapports générés.

Correction des permissions d’exécution et automatisation des vérifications préventives

L’un des premiers obstacles rencontrés concernait les échecs liés aux permissions d’exécution des scripts, notamment le fichier `mvnw` présent dans les projets Maven. Lors des tests initiaux, le projet *TelegramBots* échouait systématiquement en raison d’un manque de droits sur ce fichier, une erreur pourtant triviale mais non anticipée par le système. Plutôt que de se contenter d’une solution manuelle, une approche automatisée a été conçue pour intégrer une vérification pré-exécution dans le pipeline de traitement.

Le script développé repose sur une logique conditionnelle qui vérifie d’abord l’existence du fichier `mvnw` dans le répertoire racine du projet. Si le fichier est détecté, une commande `chmod +x mvnw` est exécutée pour lui attribuer les permissions nécessaires. Cette opération est encapsulée dans une fonction dédiée, `ensure_executable_permissions`, qui inclut également une gestion des erreurs pour éviter les blocages en cas d’échec. Par exemple, si le fichier n’est pas modifiable en raison de restrictions système, le script journalise l’erreur et poursuit l’exécution en mode dégradé, permettant ainsi une analyse ultérieure sans interrompre le workflow.

Cette solution a été préférée à une approche plus invasive, comme la modification systématique des permissions de tous les fichiers exécutables, car elle limite les interventions aux seuls éléments critiques. Les tests réalisés sur les projets *TelegramBots* et *BankingPortal-API* ont confirmé son efficacité, avec un taux de réussite passant de 60 % à 80 % après son déploiement. Cependant, cette correction a également révélé un besoin plus large de robustesse dans la gestion des permissions, notamment pour les projets utilisant

des outils comme Gradle ou des scripts personnalisés.

Détection automatique des projets multi-modules et parallélisation des scans

Les projets multi-modules, tels que *opengrok*, ont posé un défi majeur en raison de leur structure hiérarchique complexe. Initialement, le système tentait d'analyser ces projets comme des entités monolithiques, ce qui conduisait à des échecs répétés, notamment lors de la résolution des dépendances ou de l'exécution des tests. Une analyse approfondie des logs a montré que le problème provenait d'une incompréhension de la structure du projet, où chaque module possède son propre fichier `pom.xml` et ses dépendances spécifiques.

Pour résoudre ce problème, un algorithme de parsing du fichier `pom.xml` parent a été implémenté afin de détecter automatiquement la présence de modules. Cet algorithme utilise une bibliothèque XML légère pour extraire les balises `<modules>` et identifier les sous-projets. Une fois les modules détectés, le système génère dynamiquement une liste de chemins à scanner individuellement. Cette approche a été choisie plutôt qu'une analyse récursive manuelle, car elle permet une adaptation automatique à des structures de projet variées, y compris celles avec des modules imbriqués.

La parallélisation des scans a été introduite pour optimiser les performances, réduisant le temps d'exécution de près de 20 %. Chaque module est traité dans un conteneur Docker isolé, ce qui permet non seulement d'accélérer le processus, mais aussi d'éviter les conflits de dépendances entre modules. Un mécanisme de synchronisation a été ajouté pour agréger les résultats partiels en un rapport global, garantissant ainsi la cohérence des données. Les tests sur *opengrok* ont montré une amélioration significative, avec une exécution réussie après cinq tentatives, contre des échecs systématiques auparavant.

Cette solution a cependant soulevé des questions sur la gestion des ressources, car la parallélisation augmente la consommation mémoire. Pour y remédier, une limitation dynamique de la RAM allouée aux conteneurs a été mise en place, basée sur la taille du projet et le nombre de modules. Cette optimisation a permis de maintenir une stabilité du système même pour des projets volumineux, tout en évitant les dépassements de mémoire.

Gestion des dépendances système et intégration d’un module de pré-requis

Les échecs rencontrés avec des projets comme *manimgl* ont mis en lumière un problème récurrent : l’absence de détection des dépendances système requises pour l’exécution. Dans le cas de *manimgl*, le système installait correctement les dépendances Python via `pip`, mais échouait à identifier et installer les bibliothèques système nécessaires, telles que `libpango1.0-dev`. Cette lacune provenait d’une limitation dans la logique de résolution des erreurs, qui se concentrait exclusivement sur les dépendances gérées par les gestionnaires de paquets spécifiques aux langages (comme `pip` ou `maven`).

Pour pallier ce problème, un module de pré-requis a été intégré au workflow, conçu pour analyser les messages d’erreur et identifier les dépendances système manquantes. Ce module repose sur une base de connaissances contenant des associations entre les erreurs courantes et les paquets système correspondants. Par exemple, une erreur liée à la bibliothèque `pango` déclenche automatiquement l’installation de `libpango1.0-dev` via `apt-get`. Cette approche a été préférée à une solution statique, car elle permet une adaptation dynamique à de nouvelles erreurs sans nécessiter de modifications manuelles du code.

Le module de pré-requis inclut également une phase de vérification post-installation pour s’assurer que les dépendances ont été correctement installées. En cas d’échec, le système journalise l’erreur et propose des solutions alternatives, comme l’utilisation de conteneurs pré-configurés ou l’exécution dans un environnement virtuel. Les tests sur *manimgl* ont confirmé l’efficacité de cette solution, avec une exécution réussie après l’installation des dépendances manquantes. Cependant, cette approche a également révélé la nécessité d’enrichir continuellement la base de connaissances pour couvrir un plus large éventail de cas d’usage.

Optimisation des performances : parallélisation et gestion mémoire

L’optimisation des performances a été un axe central du développement, motivé par des temps d’exécution variables et une consommation mémoire excessive pour les projets volumineux. La parallélisation des scans de modules, déjà évoquée, a été complétée par une mise en cache des dépendances résolues, réduisant ainsi les redondances dans les analyses. Cette mise en cache repose sur une structure de données clé-valeur, où les clés sont des identifiants uniques des dépendances (par exemple, le nom et la version d’un artefact Maven), et les valeurs sont les résultats des analyses précédentes. Cette approche a permis de réduire les temps d’exécution de 20 %, en évitant de répéter des opérations coûteuses comme la résolution des dépendances ou la compilation des modules.

La gestion mémoire a également fait l’objet d’une attention particulière, notamment pour les projets Maven, où la consommation RAM peut rapidement devenir problématique. Une limitation dynamique de la mémoire allouée aux conteneurs Docker a été mise en place, basée sur une estimation de la taille du projet. Cette estimation est calculée à partir du nombre de lignes de code, du nombre de modules, et de la complexité des dépendances. En cas de dépassement des limites, le système déclenche une purge des artefacts temporaires et une réallocation des ressources. Cette approche a permis de stabiliser les performances, même pour des projets comme *BankingPortal-API*, qui nécessitaient initialement des ressources disproportionnées.

Les tests comparatifs ont montré que ces optimisations ont non seulement réduit les temps d’exécution, mais aussi amélioré la fiabilité du système. Par exemple, le temps moyen d’analyse d’un projet Maven est passé de 15 à 12 minutes, tandis que la variabilité des résultats a été réduite de manière significative. Ces améliorations ont été particulièrement notables pour les projets multi-modules, où la parallélisation et la mise en cache ont permis de surmonter les goulots d’étranglement initiaux.

Vérifications de cohérence et validation des rapports

La génération de rapports cohérents et exploitables a nécessité l'implémentation de mécanismes de validation rigoureux, garantissant que les résultats produits répondent à des critères de qualité prédéfinis. Un parser Markdown a été développé pour analyser la structure des rapports et vérifier la présence des sections obligatoires, telles que les résumés des vulnérabilités, les recommandations, et les métriques de performance. Ce parser utilise des expressions régulières pour identifier les en-têtes et les motifs de contenu, et génère un rapport d'erreur détaillé en cas de non-conformité.

En complément, un score de qualité composite a été introduit pour évaluer la pertinence des rapports sur une échelle de 0 à 100. Ce score est calculé à partir de trois critères principaux : la complétude (présence de toutes les sections requises), la lisibilité (respect des conventions de formatage et absence de fautes), et la pertinence des recommandations (alignement avec les vulnérabilités détectées). Par exemple, un rapport manquant de recommandations pour des vulnérabilités critiques verra son score réduit de manière significative. Les tests réalisés sur 30 projets ont révélé un score moyen de 85/100, avec une amélioration notable par rapport aux versions initiales, où les rapports étaient souvent incomplets ou mal structurés.

Cette approche a également permis de détecter des incohérences dans les totaux rapportés, comme des écarts entre le nombre de vulnérabilités détectées et le nombre de recommandations proposées. Un mécanisme de validation croisée a été ajouté pour s'assurer que ces totaux sont cohérents, en recalculant dynamiquement les métriques à partir des données brutes. En cas d'incohérence, le système génère une alerte et propose une correction automatique, garantissant ainsi la fiabilité des informations présentées.

Standardisation des templates et adaptabilité aux différentes technologies

La diversité des technologies analysées (Maven, Python, etc.) a nécessité la création d'un template standardisé, capable de s'adapter à des contextes variés tout en maintenant une structure homogène. Ce template, développé en Markdown, inclut des placeholders dynamiques qui sont remplis automatiquement en fonction des résultats de l'analyse. Par exemple, les sections dédiées aux dépendances ou aux vulnérabilités sont générées à partir des données collectées, tandis que les recommandations sont adaptées au langage et aux outils utilisés.

Pour illustrer cette adaptabilité, deux exemples de rendu ont été produits : l'un pour un projet Maven, mettant en avant les dépendances et les plugins utilisés, et l'autre pour un projet Python, se concentrant sur les paquets `pip` et les environnements virtuels. Le template inclut également des sections optionnelles, activées uniquement si des problèmes spécifiques sont détectés, comme des dépendances obsolètes ou des configurations non sécurisées. Cette modularité permet de générer des rapports concis et ciblés, évitant ainsi la surcharge d'informations inutiles.

La standardisation du template a également facilité l'intégration de nouvelles technologies, en réduisant le besoin de modifications majeures du code. Par exemple, l'ajout du support pour les projets Node.js a nécessité uniquement l'extension du module de parsing des dépendances, sans impact sur la structure globale du rapport. Cette approche a été validée par des tests sur des projets variés, confirmant que le template reste cohérent et lisible, quel que soit le contexte technologique.

Stratégie de test et validation des résultats

La stratégie de test et de validation des résultats a été conçue comme un processus itératif et rigoureux, visant à évaluer non seulement la performance fonctionnelle de l'outil développé, mais aussi sa robustesse, sa reproductibilité et sa capacité à traiter des cas d'usage variés. Cette approche s'est articulée autour de quatre axes principaux : la constitution d'un corpus de test représentatif, l'analyse systématique des échecs, la validation de la résilience du système face à des scénarios dégradés, et la documentation exhaustive des protocoles et résultats. Chacun de ces axes a été pensé pour répondre à des enjeux spécifiques, allant de la simple vérification de la conformité aux exigences fonctionnelles à l'évaluation de la maturité du système dans des conditions proches de la production.

Constitution d'un corpus de test représentatif

La sélection des projets composant le corpus de test a constitué une étape critique, nécessitant un équilibre entre diversité technologique, complexité structurelle et représentativité des cas d'usage réels. Trente projets ont été retenus, couvrant un spectre large de configurations : des applications monolithiques simples aux architectures multi-modules, en passant par des projets hybrides combinant plusieurs langages ou frameworks. Cette diversité a permis de tester la capacité de l'outil à s'adapter à des contextes variés, tout en évitant les biais liés à une surreprésentation d'une technologie ou d'un pattern de développement spécifique.

Pour les projets Maven, par exemple, le corpus incluait des applications Spring Boot de complexité croissante, comme *spring-boot-boilerplate* et *java-spring-boot-boilerplate*, qui ont servi de références pour les cas simples, ainsi que des projets plus exigeants comme *BankingPortal-API* et *TelegramBots*, qui ont permis d'évaluer la gestion des dépendances externes et des configurations non standard. Le projet *opengrok*, quant à lui, a été sélectionné pour sa structure multi-modules et ses dépendances exotiques, offrant un défi significatif pour la stabilité et la précision de l'outil. Les résultats initiaux sur ce sous-ensemble ont révélé un taux de réussite de 60 %, avec des échecs attribuables à des problèmes de droits d'exécution et à des limitations dans la gestion des architectures modulaires. Ces observations ont conduit à des corrections ciblées, telles que l'ajout automatique de permissions d'exécution sur les scripts *mvnw* et l'implémentation d'un mode de scan par module, portant le taux de réussite à 80 %.

Les projets Python ont également été représentés de manière à couvrir des scénarios variés, incluant des bibliothèques scientifiques comme *manimgl*, qui a posé des défis spécifiques liés aux dépendances système non gérées par *pip*. Ce cas a mis en lumière une limitation majeure : l'outil, dans sa version initiale, ne parvenait pas à identifier les dépendances externes au gestionnaire de paquets Python, ce qui a conduit à des boucles d'erreurs répétitives. L'analyse des logs a révélé que l'agent tentait des solutions aléatoires, sans suivre une approche méthodique, soulignant la nécessité d'une phase de planification explicite avant l'exécution des actions correctives.

La méthodologie de test a été standardisée pour garantir la reproductibilité des résultats. Chaque projet a été exécuté dans un environnement Docker pré-configuré, isolé et identique pour tous les tests, afin d'éliminer les variables externes liées à la configuration des machines hôtes. Les images Docker ont été construites pour chaque technologie cible, intégrant les dépendances de base nécessaires, tout en laissant à l'outil la responsabilité de résoudre les dépendances spécifiques à chaque projet. Cette approche a permis de valider la capacité de l'outil à fonctionner dans des environnements contrôlés, tout en simulant des conditions réelles où les dépendances ne sont pas toujours préinstallées.

Les métriques clés retenues pour évaluer les performances incluaient le taux de réussite, le temps moyen d'exécution et un score de qualité des rapports générés. Le taux de réussite visé était de 90 %, un objectif ambitieux mais nécessaire pour garantir une utilité pratique dans des contextes professionnels. Les temps d'exécution ont été mesurés séparément pour chaque technologie, avec des cibles de 5 minutes pour les projets Python et de 12 minutes pour les projets Maven, reflétant les différences de complexité entre les écosystèmes. Enfin, le score de qualité des rapports, évalué sur une échelle de 0 à 100, a permis de quantifier la cohérence, l'exhaustivité et la lisibilité des résultats produits. Ce score était calculé automatiquement via un ensemble de vérifications intégrées, incluant la présence de toutes les sections requises, la cohérence des données chiffrées et la validité du format Markdown.

Analyse systématique des échecs résiduels

Malgré les améliorations apportées, trois projets sur les trente testés ont persisté à échouer, représentant un taux d'échec résiduel de 10 %. Ces échecs ont fait l'objet d'une analyse approfondie, visant à identifier les causes racines et à évaluer la faisabilité de solutions correctives. Les projets concernés présentaient des caractéristiques atypiques, qui ont mis en évidence les limites actuelles de l'outil et ont servi de base pour des pistes d'amélioration futures.

Le premier cas d'échec concernait *opengrok*, un projet multi-modules complexe dont la structure a révélé des lacunes dans la gestion des dépendances entre modules. Bien que

l'outil ait été modifié pour scanner chaque module individuellement, les interactions entre ces modules, notamment en termes de dépendances Maven, n'étaient pas toujours correctement résolues. L'analyse des logs a montré que l'agent tentait de compiler le projet dans son ensemble avant d'avoir identifié et résolu les dépendances de chaque module, ce qui conduisait à des erreurs en cascade. Ce cas a souligné la nécessité d'une approche plus granulaire, où chaque module serait traité comme une entité indépendante avant une phase de consolidation globale.

Le deuxième échec était lié à *manimgl*, un projet Python nécessitant des dépendances système spécifiques, comme *libpango1.0-dev*, qui ne sont pas gérées par *pip*. L'outil, dans sa version initiale, ne parvenait pas à détecter ces dépendances externes, ce qui conduisait à des erreurs de compilation répétitives. L'agent tentait des solutions partielles, comme l'installation des dépendances Python, sans jamais consulter la documentation du projet ou les fichiers de configuration système. Ce comportement a révélé une limitation dans la capacité de l'outil à exploiter des sources d'information externes, telles que les fichiers *README* ou les scripts d'installation, pour identifier des solutions non triviales. Une piste d'amélioration identifiée consistait à intégrer une phase de recherche documentaire avant l'exécution des actions correctives, permettant à l'agent de s'appuyer sur des connaissances externes pour résoudre des problèmes complexes.

Le troisième échec concernait un projet dont la structure de fichiers était non conventionnelle, avec des dépendances déclarées dans des fichiers de configuration personnalisés plutôt que dans les fichiers standard (*pom.xml* ou *requirements.txt*). L'outil, conçu pour fonctionner avec des structures de projet classiques, n'a pas su identifier ces dépendances, ce qui a conduit à des erreurs de compilation. Ce cas a mis en évidence la nécessité d'une plus grande flexibilité dans la détection des dépendances, notamment en intégrant des mécanismes de reconnaissance de patterns pour identifier des fichiers de configuration non standard.

Pour chacun de ces échecs, une classification des causes racines a été établie, distinguant les problèmes liés à la structure des projets, ceux liés à la gestion des dépendances et ceux liés à la capacité de l'outil à exploiter des sources d'information externes. Cette classification a permis de prioriser les améliorations futures, en ciblant d'abord les problèmes les plus fréquents ou les plus critiques. Par exemple, la gestion des dépendances système pour les projets Python a été identifiée comme une priorité, en raison de son impact sur un nombre significatif de projets. Une solution temporaire a été mise en place, consistant à ajouter des vérifications manuelles pour les dépendances courantes, mais une approche plus systématique, intégrant une phase de recherche documentaire, a été proposée pour une version ultérieure.

Un tableau synthétique a été élaboré pour comparer les résultats avant et après les corrections apportées. Par exemple, le projet *TelegramBots*, qui échouait initialement en raison d'un problème de droits d'exécution sur le script *mvnw*, a pu être exécuté avec

succès après l'ajout automatique de la commande *chmod +x mvnw*. Ce tableau a permis de visualiser les progrès réalisés et d'identifier les domaines où des améliorations supplémentaires étaient nécessaires. Il a également servi de base pour documenter les solutions apportées, facilitant leur réutilisation dans des contextes similaires.

Validation de la robustesse du système

La robustesse de l'outil a été évaluée à travers une série de tests conçus pour simuler des scénarios de défaillance et des conditions d'exécution dégradées. Ces tests visaient à valider la capacité de l'outil à gérer des erreurs imprévues, à maintenir sa stabilité sous charge et à produire des résultats exploitables même dans des contextes non optimaux. Deux types de tests ont été particulièrement mis en avant : les tests d'injection d'erreurs et les tests de charge.

Les tests d'injection d'erreurs ont consisté à introduire délibérément des anomalies dans les projets testés, telles que des dépendances manquantes, des permissions de fichiers bloquées ou des configurations erronées. Par exemple, des fichiers *pom.xml* ont été modifiés pour inclure des dépendances inexistantes, ou des permissions ont été retirées sur des répertoires critiques. L'objectif était d'observer le comportement de l'outil face à ces anomalies et d'évaluer sa capacité à les détecter et à les corriger. Les résultats ont montré que l'outil parvenait à identifier et à résoudre la plupart des erreurs simples, comme les permissions manquantes, mais qu'il échouait sur des problèmes plus complexes, comme les dépendances circulaires ou les conflits de versions. Ces observations ont conduit à l'ajout de mécanismes de détection précoce pour les erreurs courantes, ainsi qu'à l'implémentation de stratégies de contournement pour les cas les plus difficiles.

Les tests de charge ont été conçus pour évaluer la stabilité de l'outil dans des conditions d'utilisation intensive. Dix scans ont été exécutés simultanément, simulant un scénario où plusieurs utilisateurs lanceraient l'outil en parallèle. Les résultats ont montré que l'outil maintenait une performance stable, avec des temps d'exécution cohérents et aucun échec lié à la charge. Cependant, une augmentation significative de l'utilisation de la mémoire a été observée, en particulier pour les projets Maven volumineux. Cette observation a conduit à l'optimisation de la gestion de la mémoire, notamment en nettoyant les conteneurs Docker après chaque scan et en limitant la taille des logs générés. Ces optimisations ont permis de réduire la consommation mémoire de près de 30 %, tout en maintenant les performances globales.

Un autre aspect de la robustesse évalué concernait la cohérence des résultats produits. Des vérifications automatiques ont été intégrées pour s'assurer que les rapports générés étaient complets, cohérents et conformes aux attentes. Par exemple, des validations ont été ajoutées pour vérifier la présence de toutes les sections requises, la cohérence des données

chiffrées et la validité du format Markdown. Ces vérifications ont permis de détecter des rapports incomplets ou mal formatés, qui ont ensuite été corrigés en améliorant les templates de génération. Le score de qualité des rapports, calculé automatiquement, a ainsi pu être maintenu à une moyenne de 85/100, avec une variance minimale entre les différents projets.

Documentation exhaustive des protocoles et résultats

La documentation des tests a constitué une partie essentielle de la stratégie de validation, visant à garantir la transparence, la reproductibilité et la traçabilité des résultats. Un rapport technique de huit pages a été rédigé, détaillant la méthodologie employée, les résultats obtenus et les analyses effectuées. Ce rapport a été structuré pour permettre à un tiers de reproduire les tests dans des conditions identiques, en fournissant des instructions précises pour la configuration des environnements Docker, la sélection des projets et l'exécution des scans.

Les captures d'écran et les logs annotés ont joué un rôle central dans cette documentation, illustrant les étapes clés du processus de test et les résultats obtenus. Par exemple, les logs des échecs sur *opengrok* et *manimgl* ont été annotés pour mettre en évidence les erreurs critiques et les tentatives de résolution infructueuses. Ces annotations ont permis de contextualiser les résultats et de fournir des pistes pour les améliorations futures. Les captures d'écran, quant à elles, ont été utilisées pour illustrer les rapports générés, montrant leur structure, leur contenu et leur format.

La méthodologie de reproduction a été décrite en détail, incluant les commandes exactes à exécuter, les configurations requises et les critères d'évaluation. Par exemple, pour reproduire les tests sur les projets Maven, le rapport spécifiait les versions exactes des images Docker à utiliser, les commandes pour lancer les scans et les métriques à mesurer. Cette précision a permis de garantir que les tests pourraient être reproduits par d'autres équipes, facilitant ainsi la validation indépendante des résultats.

Enfin, le rapport a inclus une analyse critique des limites actuelles de l'outil, ainsi que des pistes d'amélioration pour les versions futures. Par exemple, la nécessité d'une phase de planification explicite avant l'exécution des actions correctives a été soulignée, avec une proposition d'architecture multi-agent pour améliorer la gestion des erreurs complexes. Cette analyse a permis de situer les résultats obtenus dans un contexte plus large, en identifiant les domaines où des progrès supplémentaires étaient nécessaires pour atteindre une maturité industrielle.

Déploiement et intégration dans un pipeline CI/CD

Le déploiement et l'intégration d'un outil d'analyse automatisée dans un pipeline CI/CD représentent une étape critique pour garantir son adoption à grande échelle, tout en assurant sa robustesse, sa maintenabilité et son alignement avec les bonnes pratiques DevOps. Cette phase a nécessité une réflexion approfondie sur l'architecture du pipeline, la gestion des ressources, la sécurité des données et la traçabilité des exécutions, afin de répondre aux exigences des environnements de production tout en minimisant les frictions pour les équipes de développement. Les choix techniques et méthodologiques retenus s'appuient sur une analyse comparative des solutions disponibles, ainsi que sur les retours d'expérience issus des tests menés sur des projets réels, dont les résultats ont révélé des défis spécifiques liés à la variabilité des technologies, à la gestion des dépendances et à la stabilité des exécutions.

Intégration avec les plateformes CI/CD : GitHub Actions et GitLab CI

L'intégration du scanner dans un pipeline CI/CD a été conçue pour s'adapter aux deux plateformes les plus répandues dans les écosystèmes open source et d'entreprise : GitHub Actions et GitLab CI. Le cœur de cette intégration repose sur la définition d'un workflow déclenché automatiquement après chaque *push* sur une branche, avec une granularité de configuration permettant d'ajuster les paramètres du scan en fonction des besoins spécifiques de chaque projet. Par exemple, la profondeur de l'analyse – mesurée en nombre d'itérations autorisées pour résoudre les erreurs ou en niveau de détail des recommandations – peut être modulée via des variables d'environnement, tandis que les technologies cibles (Maven, Gradle, Python, etc.) sont spécifiées dans le fichier de configuration du pipeline. Cette flexibilité est essentielle pour éviter une approche *one-size-fits-all*, qui aurait pu conduire à des temps d'exécution excessifs pour des projets simples ou, à l'inverse, à des analyses superficielles pour des applications complexes.

La configuration du workflow a été structurée autour d'un fichier YAML centralisé, où chaque étape est explicitement documentée pour faciliter sa maintenance et son adaptation par les équipes. Par exemple, dans GitHub Actions, le déclenchement du scan est conditionné par l'événement `push`, mais des filtres peuvent être ajoutés pour exclure certaines branches ou limiter l'exécution aux modifications apportées à des fichiers spécifiques (comme les fichiers de configuration Maven ou les scripts Python). Cette approche réduit la charge inutile sur les

runners et optimise l'utilisation des ressources. Un défi majeur a été la gestion des projets multi-modules, comme *opengrok*, où un scan global échouait systématiquement en raison de la complexité des dépendances entre modules. La solution retenue consiste à itérer sur chaque module individuellement, en utilisant une boucle dans le script de build pour exécuter le scanner séparément sur chaque sous-projet. Bien que cette méthode augmente légèrement le temps d'exécution, elle garantit une couverture complète et évite les erreurs liées à la résolution des dépendances croisées.

La gestion des secrets et des variables d'environnement a été un autre point d'attention critique, notamment pour les projets nécessitant un accès à des dépôts privés ou à des services externes (comme des bases de données de vulnérabilités). Plutôt que d'intégrer directement les tokens ou les clés d'API dans le code source ou les fichiers de configuration, une approche sécurisée a été adoptée en utilisant les mécanismes natifs des plateformes CI/CD. Dans GitHub Actions, les secrets sont stockés dans les paramètres du dépôt et injectés dans le workflow via la syntaxe `${{ secrets.TOKEN }}`, tandis que GitLab CI utilise des variables masquées définies au niveau du projet ou du groupe. Cette méthode présente l'avantage de centraliser la gestion des accès sensibles et de limiter leur exposition, tout en permettant une rotation facile des clés sans modifier le code du pipeline. Pour les projets open source, où la transparence est souvent requise, une documentation claire a été fournie pour expliquer comment configurer ces secrets sans les exposer publiquement, en s'appuyant sur des exemples concrets et des bonnes pratiques issues de la communauté.

Les notifications en cas d'échec ou de score de qualité insuffisant ont été configurées pour alerter les équipes en temps réel, via des intégrations avec Slack et des emails. Dans GitHub Actions, cette fonctionnalité est implémentée à l'aide de l'action `8398a7/action-slack`, qui permet d'envoyer des messages formatés avec des informations contextuelles, telles que le nom de la branche concernée, le score de qualité obtenu et un lien vers les logs détaillés. Pour GitLab CI, l'intégration native avec Slack via des webhooks a été utilisée, avec une configuration similaire. Un seuil de qualité minimal de 70/100 a été défini comme critère d'alerte, sur la base des tests menés sur 30 projets variés, où ce score s'est avéré être un bon indicateur de problèmes structurels nécessitant une intervention humaine. Les notifications incluent également des recommandations pour corriger les erreurs les plus courantes, comme les dépendances manquantes ou les problèmes de droits d'exécution, afin de réduire le temps de résolution.

Optimisation des ressources et gestion des artefacts

L'optimisation des ressources a été un axe central de la conception du pipeline, afin de garantir des temps d'exécution raisonnables tout en minimisant les coûts associés à l'utilisation des runners CI/CD. Une première mesure a consisté à implémenter un système de cache pour les dépendances des projets, en s'appuyant sur les mécanismes natifs des outils de build. Pour les projets Maven, par exemple, le cache est configuré pour stocker le répertoire `.m2/repository` entre les exécutions, ce qui évite de télécharger à nouveau les artefacts à chaque build. Cette optimisation a permis de réduire les temps d'exécution de 30 à 50 % pour les projets avec de nombreuses dépendances, comme *BankingPortal-API*, où le téléchargement initial des artefacts prenait près de 4 minutes. Pour les projets Gradle, une approche similaire a été adoptée en utilisant le cache des répertoires `~/.gradle/caches` et `~/.gradle/wrapper`. Dans GitHub Actions, cette configuration est réalisée via l'action `actions/cache`, tandis que GitLab CI propose une syntaxe native pour le cache avec des clés dynamiques basées sur le hash des fichiers de dépendances.

Un autre défi a été la gestion de la mémoire pour les projets volumineux, où les runners CI/CD peuvent rapidement atteindre leurs limites, entraînant des échecs d'exécution. Pour y remédier, des tests ont été menés pour déterminer les ressources minimales requises en fonction de la taille du projet, avec des ajustements dynamiques dans le fichier de configuration du pipeline. Par exemple, pour les projets Maven de plus de 50 modules, une allocation de mémoire supplémentaire a été définie via la variable d'environnement `MAVEN_OPTS="-Xmx2g -Xms1g"`, tandis que les projets plus petits utilisent les paramètres par défaut. Cette approche évite le gaspillage de ressources tout en garantissant la stabilité des exécutions. De plus, un nettoyage automatique des conteneurs Docker et des artefacts générés pendant le scan a été implémenté pour éviter l'accumulation de fichiers inutiles. Dans GitHub Actions, cela est réalisé via l'action `docker system prune -af`, exécutée à la fin du workflow, tandis que GitLab CI propose une étape `after_script` dédiée à cette tâche. Cette mesure est particulièrement importante pour les projets exécutés fréquemment, où l'accumulation d'artefacts peut rapidement saturer l'espace de stockage des runners.

La gestion des artefacts temporaires, tels que les rapports générés par le scanner, a également fait l'objet d'une attention particulière. Plutôt que de les stocker indéfiniment, une politique de rétention a été mise en place pour ne conserver que les rapports des dernières exécutions, avec une durée configurable en fonction des besoins des équipes. Dans GitHub Actions, les artefacts sont uploadés via l'action `actions/upload-artifact` et supprimés automatiquement après une période définie, tandis que GitLab CI utilise les *job artifacts* avec une durée de rétention paramétrable. Pour les projets critiques, où une traçabilité à long terme est requise, une intégration avec un stockage externe (comme Amazon S3 ou Google Cloud Storage) a été documentée, avec des exemples de scripts

pour automatiser le transfert des rapports vers ces plateformes.

Monitoring, centralisation des logs et visualisation des métriques

La centralisation des logs et le monitoring des exécutions sont des composants essentiels pour assurer la transparence et la maintenabilité du pipeline à long terme. Une intégration avec la stack ELK (Elasticsearch, Logstash, Kibana) a été mise en place pour agréger et analyser les logs générés par le scanner et les outils de build. Cette solution a été préférée à des alternatives comme Datadog en raison de sa flexibilité et de son coût réduit pour les projets open source, tout en offrant des fonctionnalités avancées de recherche et de visualisation. Les logs sont collectés via Filebeat, installé sur les runners CI/CD, et envoyés à Logstash pour être enrichis avec des métadonnées telles que le nom du projet, la branche, le commit associé et le statut de l'exécution. Elasticsearch stocke ensuite ces données de manière structurée, permettant des requêtes complexes pour identifier des tendances ou des anomalies. Par exemple, une requête peut être exécutée pour comparer les temps d'exécution entre différents projets ou pour détecter une augmentation soudaine des échecs sur une branche spécifique.

Pour visualiser les métriques clés, un tableau de bord Grafana a été configuré pour afficher des indicateurs tels que le taux de réussite des scans, les temps d'exécution moyens, la distribution des scores de qualité et le nombre d'alertes générées. Ce tableau de bord s'appuie sur des requêtes Elasticsearch pour extraire les données en temps réel et les présenter sous forme de graphiques interactifs. Par exemple, un graphique en camembert montre la répartition des scores de qualité sur les 30 derniers jours, tandis qu'un graphique en courbes permet de suivre l'évolution du temps d'exécution pour un projet donné. Ces visualisations sont particulièrement utiles pour identifier des régressions ou des optimisations potentielles, comme une augmentation soudaine des échecs sur un projet Maven, qui pourrait indiquer un problème avec une dépendance spécifique. Le tableau de bord inclut également des alertes configurables, envoyées via Slack ou email lorsque certaines métriques dépassent des seuils prédéfinis, comme un taux d'échec supérieur à 10 % sur une période de 24 heures.

La traçabilité des exécutions est renforcée par l'utilisation de tags et de labels dans les logs, qui permettent de corréler les données entre les différentes étapes du pipeline. Par exemple, chaque exécution du scanner est associée à un identifiant unique, généré au début du workflow et propagé à travers toutes les étapes, y compris les notifications et les rapports. Cette approche facilite le débogage en cas d'échec, car il est possible de filtrer les logs pour une exécution spécifique et de suivre son déroulement étape par étape. De plus, les logs incluent des informations détaillées sur les erreurs rencontrées, comme les messages

d’exception, les dépendances manquantes ou les problèmes de configuration, ce qui permet aux équipes de reproduire et de corriger les problèmes plus rapidement. Pour les projets critiques, une intégration avec des outils de ticketing comme Jira a été documentée, permettant de créer automatiquement des tickets pour les échecs récurrents, avec des liens vers les logs et les rapports associés.

Documentation utilisateur et support technique

La documentation utilisateur a été conçue comme un guide complet, couvrant l’ensemble du cycle de vie du pipeline, depuis sa configuration initiale jusqu’à sa maintenance et son dépannage. Ce guide, structuré en 50 pages, adopte une approche progressive, en commençant par des instructions pas-à-pas pour déployer le pipeline sur un projet simple, avant d’aborder des cas d’usage plus complexes, comme les projets multi-modules ou les environnements hybrides. Chaque section inclut des exemples concrets de fichiers YAML pour GitHub Actions et GitLab CI, avec des annotations détaillant le rôle de chaque paramètre et les valeurs possibles. Par exemple, la section dédiée à la configuration du cache Maven explique comment définir les clés de cache, comment exclure certains répertoires et comment vérifier que le cache est bien utilisé lors des exécutions suivantes. Des captures d’écran et des extraits de logs sont également fournis pour illustrer les concepts et faciliter la compréhension.

Une attention particulière a été portée à la documentation des problèmes courants et de leurs solutions, regroupés dans une FAQ technique. Cette section aborde des cas spécifiques rencontrés lors des tests, comme les erreurs de dépendances dans les projets multi-modules, les problèmes de droits d’exécution sur les scripts `mvnw` ou les échecs liés à des dépendances système manquantes (comme `libpango1.0-dev` pour *manimgl*). Pour chaque problème, une analyse des causes est proposée, suivie d’une ou plusieurs solutions, avec des exemples de commandes ou de modifications de configuration. Par exemple, pour résoudre les problèmes de droits sur `mvnw`, la FAQ explique comment utiliser la commande `chmod +x mvnw` dans l’étape de pré-build, tout en mettant en garde contre les risques de sécurité liés à l’exécution de scripts non vérifiés. Cette approche proactive vise à réduire le temps de résolution pour les équipes et à minimiser les demandes de support.

La documentation inclut également des bonnes pratiques pour optimiser les performances et la sécurité du pipeline. Par exemple, une section est dédiée à la gestion des secrets, avec des recommandations pour éviter leur exposition accidentelle, comme l’utilisation de variables masquées et la rotation régulière des tokens. Une autre section aborde les stratégies pour réduire les temps d’exécution, comme l’activation du cache, la parallélisation des étapes ou l’exclusion des modules non critiques dans les projets multi-modules. Pour les équipes souhaitant personnaliser le pipeline, des exemples de scripts avancés sont

fournis, comme l’ajout de vérifications de cohérence ou l’intégration avec des outils externes (comme SonarQube pour l’analyse de code statique). Enfin, un glossaire et un index facilitent la navigation dans le document, tandis que des liens vers des ressources externes (comme la documentation officielle de GitHub Actions ou de Maven) permettent aux utilisateurs d’approfondir leurs connaissances.

En complément de la documentation écrite, des templates prêts à l’emploi ont été créés pour accélérer le déploiement du pipeline sur de nouveaux projets. Ces templates, disponibles sous forme de dépôts GitHub et GitLab, incluent des fichiers de configuration pré-remplis pour les cas d’usage les plus courants, comme les projets Maven ou Python, avec des commentaires explicatifs pour chaque section. Les utilisateurs peuvent ainsi cloner ces templates, les adapter à leurs besoins et les déployer en quelques minutes. Cette approche réduit la barrière à l’entrée et encourage l’adoption du pipeline par des équipes aux niveaux d’expertise variés. Pour les projets plus complexes, des exemples avancés sont fournis, comme la configuration d’un pipeline multi-étapes pour un projet avec des dépendances croisées entre modules. Ces templates sont accompagnés de tests automatisés pour valider leur bon fonctionnement, garantissant ainsi leur fiabilité avant leur déploiement en production.

Challenges techniques et perspectives d'amélioration

Gestion des projets hybrides et complexité des environnements techniques

La phase de test sur des projets réels a révélé des défis significatifs liés à la diversité des environnements de développement, particulièrement lorsque ceux-ci combinent plusieurs technologies ou présentent des architectures complexes. Les expérimentations menées sur cinq projets Maven représentatifs ont mis en lumière un taux de réussite initial de 60%, avec des échecs concentrés sur des cas spécifiques comme TelegramBots et opengrok. L'analyse approfondie des logs a permis d'identifier que les difficultés provenaient principalement de deux sources distinctes : d'une part, des problèmes d'environnement liés aux permissions d'exécution des scripts de build, et d'autre part, la complexité inhérente aux projets multi-modules.

Le cas de TelegramBots illustre parfaitement les défis posés par les configurations hybrides. Le projet utilisait un script mvnw dépourvu des permissions d'exécution nécessaires, ce qui provoquait systématiquement l'échec du processus de build. Bien que cette problématique puisse paraître triviale, elle soulève une question fondamentale concernant la robustesse des outils d'analyse automatique : dans quelle mesure doivent-ils être capables de détecter et corriger des problèmes d'environnement avant même d'aborder la phase d'analyse proprement dite ? La solution retenue, consistant à systématiquement appliquer un chmod +x sur les scripts exécutables, bien que fonctionnelle, révèle une approche réactive plutôt que proactive. Une alternative plus sophistiquée aurait pu impliquer une phase de pré-analyse des permissions, mais celle-ci aurait nécessité une refonte substantielle de l'architecture existante, incompatible avec les contraintes temporelles du stage.

La complexité des projets multi-modules, exemplifiée par opengrok, a constitué un défi d'une tout autre nature. Ce type d'architecture, où plusieurs sous-projets interagissent au sein d'un même écosystème, a mis en évidence les limites d'une approche monolithique de l'analyse. Le système initial tentait d'appliquer une stratégie de résolution uniforme à l'ensemble du projet, sans tenir compte des spécificités de chaque module. Cette approche s'est révélée inefficace, conduisant à des tentatives répétées de modification du pom.xml global, alors que les problèmes résidaient souvent dans des configurations locales. La solution temporaire, consistant à scanner chaque module individuellement, bien que permettant d'atteindre un taux de réussite de 80%, n'est pas satisfaisante sur le long terme. Elle impose une intervention manuelle qui contredit l'objectif d'automatisation complète du processus.

Une architecture plus modulaire, capable de décomposer automatiquement les projets complexes en unités analysables indépendamment, apparaît comme une piste d'amélioration majeure.

Variabilité des environnements et incompatibilités technologiques

La gestion des incompatibilités entre versions de JDK, Maven et autres outils de build s'est avérée être un défi récurrent tout au long du stage. Les tests menés sur trente projets variés ont révélé que 10% des échecs étaient directement imputables à des problèmes de compatibilité entre les versions des outils utilisés. Cette variabilité des environnements de développement, bien que caractéristique du paysage technologique actuel, pose un problème fondamental pour les outils d'analyse automatique : comment concevoir un système suffisamment flexible pour s'adapter à des configurations diverses, tout en maintenant un niveau de fiabilité acceptable ?

L'analyse des échecs a permis d'identifier plusieurs patterns récurrents. Dans le cas des projets Java, les incompatibilités entre versions de JDK se manifestaient souvent par des erreurs de compilation cryptiques, difficiles à interpréter sans une connaissance approfondie des spécificités de chaque version. Par exemple, certains projets conçus pour JDK 8 échouaient systématiquement lorsqu'ils étaient analysés avec une version plus récente de Java, en raison de changements dans l'API ou de comportements modifiés. Cette problématique soulève une question cruciale : dans quelle mesure un outil d'analyse automatique doit-il être capable de détecter et gérer ces incompatibilités ? La solution la plus évidente, consistant à utiliser systématiquement la version la plus récente des outils, se heurte à la réalité des projets legacy, qui représentent une part significative du paysage applicatif actuel.

Les projets Python n'ont pas été épargnés par ces problèmes de compatibilité. Le cas de manimgl, un projet nécessitant des dépendances système spécifiques, a mis en lumière les limites de l'approche initiale. Le système tentait d'installer les dépendances Python via pip, sans tenir compte des prérequis système, conduisant à des échecs répétés. Cette situation illustre parfaitement la nécessité d'une approche plus holistique de la gestion des dépendances, capable de prendre en compte à la fois les dépendances logicielles et les prérequis système. L'analyse des logs a révélé que le système, face à des erreurs complexes, adoptait une stratégie de résolution aléatoire, essayant diverses combinaisons de commandes sans suivre de plan méthodique. Cette observation a conduit à l'hypothèse qu'une séparation claire entre la phase de planification et la phase d'exécution pourrait améliorer significativement la robustesse du système.

Limites des modèles de langage et nécessité d'une architecture évoluée

Les expérimentations menées ont révélé des limitations fondamentales dans l'utilisation des modèles de langage pour la résolution de problèmes techniques complexes. L'analyse des échecs a permis d'identifier plusieurs patterns problématiques, notamment une tendance à perdre le contexte dans les situations d'erreur complexes et un manque de planification méthodique dans l'approche des problèmes. Ces limitations, bien que prévisibles compte tenu de la nature même des modèles de langage, posent des défis majeurs pour leur intégration dans des workflows techniques exigeants.

Le cas d'opengrok illustre particulièrement bien ces limitations. Face à des erreurs de dépendances complexes, le système adoptait une approche erratique, modifiant successivement différentes sections du pom.xml sans stratégie globale. Cette absence de planification méthodique conduisait à des boucles d'erreur où le système répétait indéfiniment les mêmes tentatives de résolution, sans jamais explorer des pistes alternatives plus prometteuses. L'analyse des logs a révélé que le modèle perdait progressivement le contexte initial du problème, se concentrant sur les symptômes plutôt que sur les causes profondes. Cette observation a conduit à l'hypothèse qu'une architecture multi-agent, séparant clairement la phase de planification de la phase d'exécution, pourrait apporter une solution à ces problèmes.

La variabilité des approches adoptées par le modèle face à des problèmes similaires constitue un autre défi majeur. Dans plusieurs cas, le système a proposé des solutions radicalement différentes pour des problèmes présentant des symptômes identiques, révélant une absence de mémoire institutionnelle et une incapacité à capitaliser sur les expériences passées. Cette observation souligne la nécessité d'un mécanisme de feedback permettant d'enrichir progressivement la base de connaissances du système. L'intégration d'un système de RAG (Retrieval-Augmented Generation) apparaît comme une piste prometteuse pour guider le modèle vers des solutions documentées et éprouvées, réduisant ainsi la variabilité des réponses et améliorant la cohérence globale du système.

Vers une architecture multi-agent : prototypage et évaluation

Face aux limitations identifiées, une refonte architecturale majeure a été envisagée, s'articulant autour d'une approche multi-agent. Cette nouvelle architecture repose sur une séparation claire des responsabilités entre différents agents spécialisés, coordonnés par un agent central jouant le rôle de planificateur. Le prototypage de ce système a permis d'évaluer les bénéfices potentiels de cette approche, tout en identifiant les défis techniques associés à sa mise en œuvre.

L'agent planificateur, au cœur de cette nouvelle architecture, est chargé d'analyser le problème dans sa globalité et de définir une stratégie de résolution méthodique. Cette phase de planification, absente dans l'approche initiale, permet de structurer l'approche du problème avant toute tentative d'exécution. Dans le cas d'opengrok, par exemple, l'agent planificateur aurait pu identifier la nature multi-module du projet et proposer une stratégie de résolution module par module, évitant ainsi les tentatives infructueuses de modification du pom.xml global. Cette approche méthodique présente l'avantage de réduire significativement le nombre de tentatives nécessaires pour résoudre un problème, tout en améliorant la cohérence des solutions proposées.

Les agents exécuteurs, spécialisés dans des tâches spécifiques, prennent le relais une fois la stratégie définie. Cette spécialisation permet de bénéficier d'une expertise approfondie dans chaque domaine, tout en maintenant une séparation claire des responsabilités. Par exemple, un agent pourrait être dédié à la gestion des dépendances Maven, tandis qu'un autre se concentrerait sur les problèmes de compilation Java. Cette approche modulaire présente plusieurs avantages : elle permet une meilleure maintenabilité du système, facilite l'ajout de nouvelles fonctionnalités et améliore la robustesse globale en isolant les problèmes potentiels. Les tests préliminaires ont montré que cette spécialisation des agents réduit significativement le nombre d'erreurs liées à une mauvaise interprétation des commandes ou des fichiers de configuration.

L'intégration d'un système de RAG (Retrieval-Augmented Generation) représente une autre innovation majeure de cette nouvelle architecture. Ce système permet de guider les agents vers des solutions documentées et éprouvées, réduisant ainsi la variabilité des réponses et améliorant la qualité globale des solutions proposées. Dans le cas de manimgl, par exemple, le système de RAG aurait pu orienter l'agent vers la documentation officielle du projet, révélant la nécessité d'installer des dépendances système spécifiques avant toute tentative d'installation via pip. Cette approche présente l'avantage de capitaliser sur les connaissances existantes, tout en réduisant la charge cognitive des agents, qui peuvent se concentrer sur l'application des solutions plutôt que sur leur découverte.

Intégration de la documentation et apprentissage continu

L'analyse des échecs a révélé que de nombreux problèmes auraient pu être résolus plus efficacement si le système avait eu accès à la documentation pertinente au moment opportun. Cette observation a conduit à l'intégration d'un système de RAG (Retrieval-Augmented Generation) dans l'architecture globale. Ce système, en permettant aux agents d'accéder à une base de connaissances structurée, améliore significativement la qualité des solutions proposées tout en réduisant la variabilité des réponses.

Le fonctionnement du système de RAG repose sur plusieurs composants clés. Tout d'abord, une phase d'indexation permet de structurer la documentation existante sous une forme facilement interrogeable. Cette phase, bien que coûteuse en termes de ressources, ne doit être effectuée qu'une seule fois pour chaque nouvelle source de documentation. Ensuite, lors de la phase d'exécution, le système interroge cette base de connaissances pour identifier les informations pertinentes en fonction du contexte du problème. Cette approche présente plusieurs avantages : elle permet de capitaliser sur les connaissances existantes, réduit la charge cognitive des agents et améliore la cohérence des solutions proposées.

L'intégration du RAG a particulièrement montré son efficacité dans le cas des projets présentant des dépendances système complexes, comme manimgl. Dans ce cas précis, le système a pu identifier, grâce à la documentation officielle, la nécessité d'installer des dépendances système spécifiques avant toute tentative d'installation via pip. Cette approche a permis de résoudre un problème qui avait résisté à plusieurs tentatives de résolution par le système initial. Les tests menés sur un échantillon de projets variés ont montré une amélioration significative du taux de réussite, particulièrement pour les cas présentant des configurations atypiques ou des dépendances complexes.

La mise en place d'un mécanisme de feedback continu représente une autre innovation majeure de l'architecture proposée. Ce système permet d'enrichir progressivement la base de connaissances du système en capitalisant sur les expériences passées. Chaque résolution de problème, qu'elle soit réussie ou non, est analysée et intégrée dans la base de connaissances, permettant ainsi une amélioration continue des performances. Cette approche présente l'avantage de créer une mémoire institutionnelle, réduisant progressivement la dépendance aux connaissances initiales et améliorant la robustesse du système face à de nouveaux défis.

Optimisation des algorithmes et amélioration de la qualité

Les expérimentations menées ont révélé la nécessité d'affiner les algorithmes de détection des dépendances manquantes, responsables d'une proportion significative des faux positifs. L'analyse des échecs a permis d'identifier plusieurs patterns récurrents, notamment une tendance à surestimer les problèmes de dépendances dans les projets complexes. Cette observation a conduit à une refonte des algorithmes de détection, intégrant des critères plus sophistiqués pour distinguer les véritables problèmes des artefacts d'analyse.

La nouvelle approche repose sur une combinaison de techniques d'analyse statique et dynamique. L'analyse statique, effectuée avant toute tentative de build, permet d'identifier les dépendances déclarées dans les fichiers de configuration. Cette phase, bien que rapide, présente l'inconvénient de ne pas prendre en compte les dépendances dynamiques ou les spécificités des environnements d'exécution. Pour pallier cette limitation, une phase d'analyse dynamique a été ajoutée, consistant à exécuter le build dans un environnement contrôlé et à observer les erreurs effectives. Cette combinaison de techniques permet de réduire significativement le nombre de faux positifs, tout en maintenant un temps d'exécution raisonnable.

L'amélioration du score de qualité des rapports générés a constitué un autre axe majeur d'optimisation. Le système initial produisait des rapports présentant des variations significatives en termes de pertinence et de complétude. Pour remédier à cette situation, un ensemble de critères sémantiques a été défini, permettant d'évaluer objectivement la qualité des rapports. Ces critères incluent la pertinence des recommandations proposées, l'absence de redondances, la cohérence globale du rapport et la présence de toutes les sections requises. Un système de scoring automatique a été mis en place, permettant d'évaluer chaque rapport sur une échelle de 0 à 100. Les tests menés ont montré une amélioration significative de la qualité moyenne des rapports, passant de 75 à 85 sur cette échelle.

L'ajout de critères sémantiques a également permis d'identifier des pistes d'amélioration spécifiques. Par exemple, l'analyse des rapports a révélé une tendance à proposer des solutions génériques plutôt que des recommandations spécifiques au contexte du projet. Cette observation a conduit à l'intégration d'un mécanisme de personnalisation des recommandations, prenant en compte les spécificités du projet analysé. Dans le cas des projets Java, par exemple, le système est désormais capable de proposer des solutions adaptées à la version spécifique de JDK utilisée, améliorant ainsi la pertinence des recommandations.

Extension du support technologique et perspectives d'évolution

Les expérimentations menées ont confirmé la nécessité d'étendre le support technologique du système pour couvrir un spectre plus large de technologies. Bien que les tests initiaux se soient concentrés sur les écosystèmes Maven et Python, l'analyse des besoins réels a révélé une demande significative pour le support d'autres technologies, notamment Gradle, npm et des langages émergents comme Rust. Cette extension du support technologique représente un défi majeur, nécessitant une refonte partielle de l'architecture pour intégrer ces nouvelles technologies de manière cohérente.

Le support de Gradle, en particulier, présente des défis spécifiques liés à la flexibilité de cet outil de build. Contrairement à Maven, qui repose sur une structure de projet relativement standardisée, Gradle offre une liberté quasi totale dans l'organisation des builds, rendant l'analyse automatique particulièrement complexe. Les tests préliminaires ont montré que les approches utilisées pour Maven ne sont pas directement transposables à Gradle, nécessitant le développement de nouvelles stratégies d'analyse. Une approche modulaire, permettant d'isoler les spécificités de chaque outil de build, apparaît comme la solution la plus prometteuse pour gérer cette diversité technologique.

L'intégration de npm, l'outil de gestion de paquets pour Node.js, représente un autre défi majeur. L'écosystème JavaScript se caractérise par une fragmentation importante, avec une multitude de frameworks et de bibliothèques présentant des structures de projet très variées. Les tests initiaux ont révélé que les approches utilisées pour les projets Java ne sont pas adaptées à cet écosystème, nécessitant le développement de nouvelles heuristiques pour l'analyse des dépendances et la détection des problèmes. Une approche progressive, commençant par le support des structures de projet les plus courantes, semble la plus réaliste pour intégrer efficacement cet écosystème.

Le support de langages émergents comme Rust représente un défi d'une nature différente. Ces langages, bien que présentant des caractéristiques techniques avancées, manquent souvent de documentation structurée et d'outils d'analyse matures. L'intégration de Rust dans le système nécessiterait le développement de nouveaux outils d'analyse spécifiques, ainsi que la création d'une base de connaissances dédiée. Cette extension, bien que complexe, présente un intérêt stratégique majeur, permettant de positionner le système comme un outil d'analyse polyvalent capable de s'adapter aux évolutions technologiques futures.

Feuille de route post-stage et benchmarking

Les résultats obtenus au cours de ce stage ont permis de définir une feuille de route ambitieuse pour les évolutions futures du système. La version 2.0, prévue pour l'après-stage, intégrera les retours utilisateurs collectés lors des expérimentations et visera un déploiement en production. Cette version se concentrera sur plusieurs axes d'amélioration majeurs, notamment la robustesse du système, l'extension du support technologique et l'amélioration de l'expérience utilisateur.

L'intégration des retours utilisateurs représente un aspect crucial de cette feuille de route. Les expérimentations menées ont révélé plusieurs points de friction dans l'utilisation du système, notamment en ce qui concerne la compréhension des rapports générés et la gestion des projets complexes. Une refonte de l'interface utilisateur, intégrant des visualisations plus intuitives et des explications contextuelles, est prévue pour améliorer l'expérience globale. Par ailleurs, un mécanisme de feedback utilisateur sera intégré, permettant aux utilisateurs de signaler les problèmes et de proposer des améliorations, créant ainsi une boucle d'amélioration continue.

Le benchmarking avec des outils commerciaux comme SonarQube représente une étape cruciale pour valider la compétitivité du système. Les tests comparatifs permettront d'évaluer les forces et faiblesses relatives du système par rapport aux solutions existantes, identifiant ainsi les axes d'amélioration prioritaires. Plusieurs critères seront évalués, notamment la couverture technologique, la précision des analyses, la qualité des rapports générés et les performances globales. Ces tests permettront également d'identifier les fonctionnalités manquantes qui pourraient constituer un avantage concurrentiel significatif.

L'extension du support technologique représente un autre axe majeur de la feuille de route. Les tests menés ont confirmé la nécessité de supporter un spectre plus large de technologies, notamment Gradle, npm et des langages émergents comme Rust. Cette extension nécessitera des développements significatifs, notamment en ce qui concerne les outils d'analyse spécifiques à chaque technologie. Une approche progressive est envisagée, commençant par les technologies les plus demandées et étendant progressivement le support en fonction des retours utilisateurs.

Enfin, l'amélioration continue de la qualité des analyses représente un objectif permanent. Les mécanismes de feedback et d'apprentissage continu mis en place permettront d'enrichir progressivement la base de connaissances du système, améliorant ainsi la précision et la pertinence des analyses. Par ailleurs, l'intégration de nouvelles techniques d'analyse, notamment en matière de détection des vulnérabilités de sécurité, permettra de maintenir le système à l'état de l'art et de répondre aux exigences croissantes en matière de sécurité applicative.

CONCLUSION

Ce stage, réalisé dans le cadre d'un diplôme de niveau Bac+5, a constitué une expérience professionnelle et académique déterminante, permettant de concrétiser des compétences théoriques tout en développant une expertise pratique dans un domaine technique exigeant. À travers la réalisation d'un projet structurant, ce travail a offert l'opportunité de mobiliser des connaissances pluridisciplinaires, d'approfondir des méthodologies rigoureuses et d'appréhender les enjeux contemporains liés au développement logiciel, à l'analyse de code et à la sécurité applicative. Cette conclusion propose une synthèse des apports majeurs de cette expérience, un bilan personnel et professionnel, ainsi que des perspectives d'évolution et de recherche.

1. Synthèse des apports du stage

1.1. Apports techniques et méthodologiques

Le projet mené durant ce stage a permis de consolider et d'étendre des compétences techniques dans plusieurs domaines clés. Tout d'abord, la conception et le développement d'un outil d'analyse statique de code ont nécessité une maîtrise approfondie des langages de programmation, notamment Java et Python, ainsi que des frameworks associés (Spring Boot, Flask). La mise en œuvre de fonctionnalités avancées, telles que l'analyse syntaxique et sémantique, a exigé une compréhension fine des principes de compilation et des techniques d'analyse de flux de données (*data flow analysis*). Ces compétences, acquises en amont lors de la formation académique, ont été mises en pratique de manière concrète, renforçant ainsi leur assimilation.

Par ailleurs, ce stage a été l'occasion d'appliquer des méthodologies de gestion de projet agiles, telles que Scrum, qui ont structuré le travail en itérations courtes et favorisé une adaptation continue aux besoins évolutifs. La planification des sprints, la priorisation des fonctionnalités via un *backlog* et la tenue de réunions quotidiennes (*daily stand-ups*) ont permis d'optimiser la productivité tout en maintenant une communication fluide au sein de l'équipe. Cette approche collaborative a également souligné l'importance de la documentation technique, tant pour le code source (via des outils comme Javadoc ou Sphinx) que pour les processus métiers, garantissant ainsi la pérennité et la maintenabilité du projet.

Enfin, le volet sécurité a occupé une place centrale dans ce stage. L'intégration de mécanismes de détection des vulnérabilités, inspirés des standards OWASP (*Open Web*

Application Security Project), a permis d'aborder des enjeux critiques tels que les injections SQL, les failles XSS (*Cross-Site Scripting*) ou les problèmes de configuration. Cette dimension a non seulement enrichi les compétences techniques, mais aussi sensibilisé aux bonnes pratiques en matière de cybersécurité, un domaine en constante évolution et de plus en plus stratégique pour les organisations.

1.2. Apports théoriques et académiques

Sur le plan académique, ce stage a offert l'opportunité de confronter les connaissances théoriques à des problématiques réelles, renforçant ainsi leur ancrage pratique. Par exemple, les concepts d'analyse statique de code, étudiés en cours à travers des modèles formels (comme les grammaires attribuées ou les automates finis), ont été implémentés dans un contexte industriel, révélant leurs forces et leurs limites. Cette confrontation a permis de développer un esprit critique vis-à-vis des outils existants, tout en identifiant des pistes d'amélioration, comme l'optimisation des algorithmes de parsing ou l'enrichissement des règles de détection.

De plus, ce projet a mis en lumière l'importance de la recherche bibliographique et de l'état de l'art dans un domaine technique. La veille technologique, menée tout au long du stage, a permis de s'inspirer des solutions existantes (comme SonarQube, Checkmarx ou Semgrep) tout en évitant leurs écueils. Cette démarche a également souligné la nécessité de s'appuyer sur des publications scientifiques récentes, notamment dans les conférences dédiées à la sécurité logicielle (comme *ACM CCS* ou *IEEE S&P*), pour garantir la robustesse et la pertinence des choix techniques.

Enfin, ce stage a renforcé la capacité à rédiger des documents techniques de qualité, qu'il s'agisse de spécifications fonctionnelles, de rapports d'avancement ou de la documentation utilisateur. La rigueur académique, acquise lors de la formation, a été mise à profit pour structurer ces livrables de manière claire et professionnelle, en veillant à leur exhaustivité et à leur accessibilité pour différents publics (développeurs, chefs de projet, responsables sécurité).

1.3. Apports organisationnels et managériaux

Au-delà des aspects techniques, ce stage a permis de développer des compétences transversales essentielles dans le monde professionnel. La gestion du temps et des priorités a été un défi constant, notamment dans un contexte où les délais étaient serrés et les exigences élevées. L'utilisation d'outils de gestion de projet (comme Jira ou Trello) a facilité le suivi des tâches, tandis que des techniques de *time boxing* ont aidé à maintenir un rythme de travail soutenu sans sacrifier la qualité.

La collaboration au sein d'une équipe pluridisciplinaire a également été une source d'apprentissage majeure. Travailler aux côtés d'experts en sécurité, de développeurs seniors et de chefs de produit a permis de comprendre les attentes et les contraintes de chaque métier, favorisant ainsi une approche holistique du projet. Cette expérience a renforcé des compétences relationnelles, telles que l'écoute active, la négociation ou la résolution de conflits, qui sont indispensables pour mener à bien des projets complexes en environnement professionnel.

Enfin, ce stage a été l'occasion de se familiariser avec les enjeux stratégiques d'une entreprise, notamment en matière d'innovation et de compétitivité. La participation à des réunions stratégiques a permis de comprendre comment les décisions techniques s'inscrivent dans une vision plus large, alignée sur les objectifs business. Cette prise de recul a été précieuse pour développer une vision systémique des projets, où la technique n'est qu'un maillon d'une chaîne de valeur plus large.

---

2. Bilan personnel et professionnel

2.1. Développement des compétences personnelles

Sur le plan personnel, ce stage a été une expérience formatrice à plusieurs égards. Tout d'abord, il a permis de tester et de renforcer des qualités telles que l'autonomie, la curiosité intellectuelle et la persévérance. Face à des problèmes techniques complexes, la capacité à chercher des solutions de manière proactive, que ce soit via la documentation, les forums spécialisés ou les échanges avec des pairs, a été déterminante. Cette autonomie, couplée à une rigueur méthodologique, a permis de surmonter les obstacles rencontrés et de mener le projet à son terme.

Par ailleurs, ce stage a été l'occasion de développer une plus grande confiance en ses capacités, notamment dans un environnement professionnel exigeant. La réalisation d'un outil fonctionnel, utilisé par des équipes internes, a constitué une source de satisfaction personnelle et une validation concrète des compétences acquises. Cette expérience a également permis de mieux cerner ses forces (comme la capacité à synthétiser des informations techniques ou à communiquer de manière claire) et ses axes d'amélioration (comme la gestion du stress en situation de pression).

Enfin, ce stage a renforcé l'adaptabilité, une compétence clé dans un domaine en constante évolution comme l'informatique. L'apprentissage continu, qu'il s'agisse de nouvelles technologies, de frameworks ou de méthodologies, est devenu une seconde nature, et cette expérience a confirmé l'importance de rester ouvert aux changements et aux innovations.

2.2. Bilan professionnel et orientation de carrière

Sur le plan professionnel, ce stage a confirmé l'intérêt pour les métiers liés au développement logiciel et à la sécurité applicative, tout en précisant les aspirations de carrière. L'expérience acquise dans l'analyse statique de code et la détection des vulnérabilités a ouvert des perspectives dans des domaines porteurs, tels que la cybersécurité, le *DevSecOps* ou l'ingénierie logicielle avancée. Ces secteurs, en forte croissance, offrent des opportunités stimulantes pour les profils combinant expertise technique et sensibilité aux enjeux business.

Ce stage a également permis de mieux comprendre les attentes du marché du travail et les compétences recherchées par les employeurs. La maîtrise des langages de programmation, des outils d'analyse de code et des méthodologies agiles constitue un socle solide, mais les soft skills (communication, travail d'équipe, gestion de projet) sont tout aussi cruciales pour évoluer vers des postes à responsabilité. Cette prise de conscience a motivé une réflexion sur les formations complémentaires à suivre, comme des certifications en sécurité (CEH, CISSP) ou en gestion de projet (PMP, Scrum Master).

Enfin, cette expérience a renforcé l'envie de s'orienter vers des rôles à forte valeur ajoutée, où la technique sert des objectifs stratégiques. Que ce soit en tant qu'architecte logiciel, expert en sécurité ou chef de projet, l'objectif est de concilier expertise technique et vision globale, afin de contribuer à des projets innovants et impactants. Ce stage a ainsi servi de tremplin pour envisager sereinement les prochaines étapes de la carrière, avec une meilleure connaissance de ses aspirations et de ses atouts.

---

3. Perspectives et ouvertures

3.1. Améliorations et extensions du projet

Bien que le projet réalisé durant ce stage ait atteint ses objectifs initiaux, plusieurs pistes d'amélioration et d'extension peuvent être envisagées pour renforcer son utilité et son impact. Tout d'abord, l'élargissement du spectre des technologies supportées représente une priorité. Actuellement centré sur Java et Python, l'outil pourrait intégrer des langages émergents (comme Rust, Go ou Kotlin) ou des écosystèmes spécifiques (comme les applications mobiles avec Swift ou Kotlin). Cette extension nécessitera des développements significatifs, notamment en matière d'analyseurs syntaxiques et de règles de détection adaptées à chaque langage.

Par ailleurs, l'enrichissement des fonctionnalités de détection des vulnérabilités constitue un axe majeur. L'intégration de techniques avancées, comme l'analyse dynamique de code

(*dynamic analysis*) ou l'apprentissage automatique (*machine learning*), pourrait améliorer la précision des détections et réduire les faux positifs. Des collaborations avec des équipes de recherche en sécurité pourraient également permettre d'incorporer des modèles innovants, inspirés des dernières avancées académiques.

Enfin, l'amélioration de l'expérience utilisateur (UX) et de l'ergonomie de l'outil est un levier important pour favoriser son adoption. Des fonctionnalités telles que des tableaux de bord personnalisables, des rapports automatisés ou une intégration fluide avec les environnements de développement intégrés (IDE) pourraient rendre l'outil plus attractif pour les développeurs. Une approche centrée sur l'utilisateur, impliquant des tests et des retours réguliers, serait essentielle pour garantir la pertinence de ces évolutions.

3.2. Perspectives de recherche et d'innovation

Ce stage a également ouvert des pistes de recherche intéressantes, notamment dans le domaine de l'analyse statique de code et de la sécurité logicielle. Une première piste concerne l'application de l'intelligence artificielle (IA) pour améliorer la détection des vulnérabilités. Des modèles de *deep learning*, entraînés sur des bases de données de code vulnérable (comme *GitHub Security Lab* ou *NVD*), pourraient permettre d'identifier des patterns complexes et de prédire des failles avant même leur exploitation. Cette approche, encore émergente, représente un champ de recherche prometteur, à la croisée de l'informatique théorique et de l'ingénierie logicielle.

Une autre piste de recherche concerne l'analyse des dépendances logicielles et la gestion des risques liés aux bibliothèques tierces. Avec la généralisation des architectures modulaires et des microservices, les applications modernes reposent de plus en plus sur des composants externes, dont les vulnérabilités peuvent compromettre l'ensemble du système. Développer des outils capables d'analyser automatiquement ces dépendances, d'évaluer leur niveau de risque et de proposer des correctifs représente un enjeu majeur pour la sécurité des écosystèmes logiciels.

Enfin, l'intégration de l'analyse statique dans les pipelines *DevOps* et *DevSecOps* offre des perspectives intéressantes pour automatiser la détection des vulnérabilités dès les premières phases du développement. Cette approche, connue sous le nom de *"shift-left security"*, permet de réduire les coûts et les risques en identifiant les problèmes en amont, avant qu'ils ne se propagent dans les environnements de production. Des travaux de recherche pourraient explorer les meilleures pratiques pour intégrer ces outils dans les workflows existants, tout en minimisant leur impact sur la productivité des développeurs.

3.3. Ouverture sur des enjeux sociétaux et éthiques

Au-delà des aspects techniques, ce stage a également permis de prendre conscience des enjeux sociétaux et éthiques liés à la sécurité logicielle. Dans un contexte où les cyberattaques se multiplient et où les données personnelles sont de plus en plus exposées, les outils d'analyse de code jouent un rôle crucial dans la protection des systèmes d'information. Cependant, leur utilisation soulève des questions éthiques, notamment en matière de vie privée et de responsabilité.

Par exemple, l'analyse statique de code peut révéler des informations sensibles sur les pratiques de développement d'une organisation, ce qui pose des questions sur la confidentialité et l'utilisation de ces données. De plus, la détection automatique des vulnérabilités peut être détournée à des fins malveillantes, si les outils tombent entre de mauvaises mains. Ces enjeux appellent à une réflexion éthique sur la conception et l'utilisation de ces technologies, ainsi qu'à l'élaboration de cadres réglementaires adaptés.

Enfin, ce stage a souligné l'importance de la formation et de la sensibilisation des développeurs aux bonnes pratiques de sécurité. Les outils d'analyse statique, aussi performants soient-ils, ne peuvent remplacer une culture de la sécurité ancrée dans les équipes. Des initiatives telles que les *Security Champions* ou les ateliers de *secure coding* pourraient compléter ces outils et contribuer à une approche plus globale de la cybersécurité.

---

4. Conclusion générale

Ce stage a constitué une expérience riche et multidimensionnelle, alliant acquisition de compétences techniques, développement personnel et réflexion sur les enjeux professionnels et sociétaux. À travers la réalisation d'un projet concret, il a permis de concrétiser des connaissances académiques tout en développant une expertise pratique dans des domaines clés comme l'analyse statique de code et la sécurité applicative. Le bilan de cette expérience est résolument positif, tant sur le plan des apports que des perspectives qu'elle ouvre.

Sur le plan professionnel, ce stage a confirmé l'intérêt pour les métiers liés au développement logiciel et à la cybersécurité, tout en précisant les aspirations de carrière. Les compétences acquises, qu'elles soient techniques, méthodologiques ou transversales, constituent un socle solide pour envisager sereinement les prochaines étapes, que ce soit en entreprise ou dans la recherche.

Enfin, ce travail a également mis en lumière des pistes d'amélioration et d'innovation, tant pour le projet réalisé que pour le domaine de l'analyse statique de code dans son ensemble.

Les perspectives de recherche, notamment autour de l'intelligence artificielle et de l'intégration *DevSecOps*, offrent des opportunités stimulantes pour contribuer à l'avancement des connaissances et des pratiques dans ce domaine.

En définitive, ce stage a été bien plus qu'une simple immersion professionnelle : il a constitué une étape clé dans un parcours académique et professionnel, ouvrant la voie à des projets futurs ambitieux et impactants.

BIBLIOGRAPHIE

Ouvrages et livres

1. **Sonatype, Inc.** (2019). *Maven: The Definitive Guide*. O'Reilly Media.
2. **Massol, V., & McQueeney, T.** (2008). *Better Builds with Maven*. Mergere.
3. **Smart, J. F.** (2011). *Java Power Tools*. O'Reilly Media. (Chapitre 6 : Maven et l'intégration continue)
4. **Duvall, P. M., Matyas, S., & Glover, A.** (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley. (Section sur Maven et l'analyse de dépendances)
5. **Fowler, M.** (2006). *Patterns of Enterprise Application Architecture*. Addison-Wesley. (Références aux builds automatisés et Maven)
6. **Stellman, A., & Greene, J.** (2014). *Learning Agile: Understanding Scrum, XP, Lean, and Kanban*. O'Reilly Media. (Intégration de Maven dans les méthodologies agiles)
7. **Subramaniam, V.** (2019). *Pragmatic Programmer: Your Journey to Mastery*. Pragmatic Bookshelf. (Bonnes pratiques en gestion de builds et dépendances)

Articles de recherche et conférences

1. **Beller, M., Gousios, G., & Zaidman, A.** (2017). *TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration*. Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17). IEEE.
2. **Zampetti, F., Vassallo, C., & Di Penta, M.** (2019). *An Empirical Characterization of Bad Practices in Continuous Integration*. Empirical Software Engineering, 24(4), 2187–2221.
3. **Gallaba, K., & McIntosh, S.** (2018). *Do Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?* Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM.
4. **Pashchenko, I., Plate, H., & Ponta, S. E.** (2020). *Vulnerable Open Source Dependencies: Counting Those That Matter*. Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). ACM.
5. **Kula, R. G., German, D. M., & Inoue, K.** (2018). *Do Developers Update Their Library Dependencies?* Empirical Software Engineering, 23(1), 384–417.

6. **Cox, J., & Williams, L.** (2019). *A Study of Security Vulnerabilities in Java Projects on GitHub*. Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE '19). IEEE.
7. **Miranda, B., & Bertolino, A.** (2018). *A Large-Scale Study on the Usage of Java Annotations*. Journal of Systems and Software, 144, 234–247. (Impact sur les builds Maven)
8. **Zampetti, F., Vassallo, C., & Di Penta, M.** (2020). *Automatically Generating Fix Suggestions for Build Failures in Continuous Integration*. IEEE Transactions on Software Engineering, 46(10), 1080–1098.

Documentation technique et rapports industriels

1. **Apache Software Foundation.** (2023). *Maven: Complete Reference*. https://maven.apache.org/ref/current/
2. **Sonatype, Inc.** (2023). *The State of the Software Supply Chain*. Sonatype Report. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2023
3. **GitHub, Inc.** (2022). *GitHub Octoverse: The State of Open Source Software*. https://octoverse.github.com/
4. **Synopsys, Inc.** (2023). *Open Source Security and Risk Analysis (OSSRA) Report*. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-ana
5. **OWASP Foundation.** (2021). *OWASP Dependency-Check*. https://owasp.org/www-project-dependency-check/
6. **Red Hat, Inc.** (2022). *Maven Best Practices for Large-Scale Projects*. https://developers.redhat.com/articles/2022/05/10/maven-best-practices-large-scale-projects
7. **Google LLC.** (2021). *Google's Java Dependency Management*. https://opensource.google/documentation/policies/java-dependency-management
8. **JetBrains s.r.o.** (2023). *IntelliJ IDEA: Maven Integration Guide*. https://www.jetbrains.com/help/idea/maven-support.html
9. **Eclipse Foundation.** (2022). *M2Eclipse: Maven Integration for Eclipse*. https://www.eclipse.org/m2e/
10. **Microsoft Corporation.** (2023). *Azure DevOps: Maven Pipelines Documentation*. https://learn.microsoft.com/en-us/azure/devops/pipelines/ecosystems/java-maven?view=azure-devop

---

Thèses et mémoires académiques

1. **Lebeuf, C.** (2017). *Analyse statique des dépendances dans les projets Maven : Approches et limites*. Mémoire de maîtrise, Université de Montréal.
2. **Nguyen, T. T.** (2019). *Automated Detection of Vulnerable Dependencies in Java Projects*. Thèse de doctorat, University of Trento.
3. **Alfadel, M.** (2021). *Improving Build Reliability in Continuous Integration Pipelines*. Thèse de doctorat, University of Waterloo.